TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNCAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Fast Particle Buffering in AutoPas: Reducing Neighbor List Rebuild Time

Xhulia Jasimi

TUM SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

TECHNCAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Fast Particle Buffering in AutoPas: Reducing Neighbor List Rebuild Time

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

Abstract

Contents

1. Introduction

1.1. Section

Citation test [12]. This is the introduction of the thesis.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 1.1.: An example for a source code listing.

2. Theoretical 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 [11] [1]

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 accurately capturing diverse interactions.

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. [7]

Lennard-Jones Potential (LJ)

The Lennard-Jones 12-6 potential is a widely used function in molecular dynamics simulations to model interactions between particles, such as atoms or molecules. 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. [16]

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. [9]

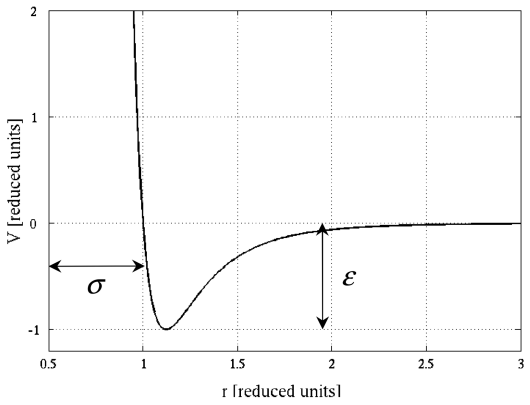


Figure 2.1.: Lennard-Jones Potential [5]

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 [6]. 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 halves the number of force calculations, significantly improving computational efficiency. In AutoPas, this optimization is implemented as the Newton3-optimization [7].

3. Technical Background

3.1. 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. This adaptability is particularly useful for large-scale or dynamic simulations where optimal configurations may shift over time. Although AutoPas is primarily intended for node-level simulations, however it can be integrated into distributed systems. The library also includes example applications, such as the md-flexible framework, which will be used in the course of this thesis [8].

3.1.1. 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. In contrast, the SoA layout separates particle properties into individual arrays, with each array containing data for all particles, enabling 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. The choice of data layout depends on the specific simulation requirements, as each has trade-offs in terms of memory access patterns and computational performance [8].

3.1.2. 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 3.1a, 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 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 [7]. 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. 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 3.1b.

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 $O(n)$ [10]. Furthermore, Linked Cells improve cache efficiency by storing particles within the same cell contiguously in memory. Nonetheless, there is still overhead, since additional particles need to be evaluated to determine if they can be considered for force calculations.

Verlet Lists

The Verlet list method extends the Linked Cells algorithm by precomputing and storing neighbor lists or verlet lists for each particle. These lists include all particles within a cutoff radius r_c plus an additional buffer zone defined by the verlet skin factor s, resulting in a search radius of $r_c \cdot s$, illustrated with yellow in Figure 3.1c. This buffer ensures that fast-moving particles do not enter or leave the cutoff region unnoticed, allowing the neighbor lists to be reused for multiple iterations.

Constructing these lists involves evaluating only particles in nearby cells, improving runtime efficiency. During each iteration, distance checks are performed only for the stored neighbors, achieving a complexity of $O(N)$ [17]. While smaller buffer sizes reduce unnecessary calculations, they require more frequent updates to the lists, making the choice of s crucial for performance.

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 [7].

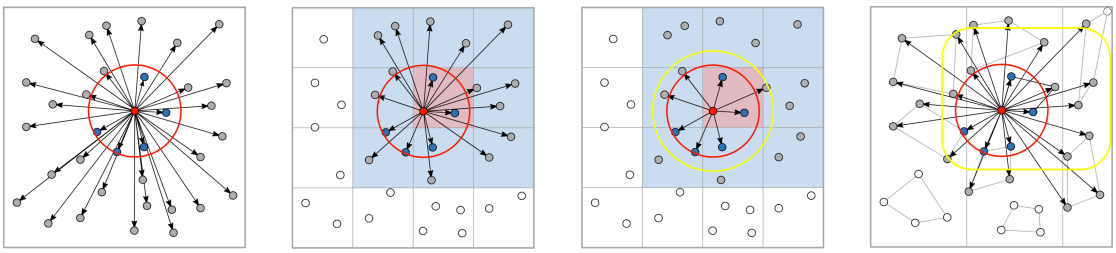
Verlet Cluster Lists

Verlet Cluster Lists are built upon regular Verlet Lists by grouping particles into clusters rather than maintaining individual neighbor lists for each particle Figure 3.1d. 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 $\frac{1}{M}$ [7], 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.

3.1.3. 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 [8].

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



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

Figure 3.1.: Neighbor identification algorithms [7]

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.

3.1.4. Base Steps in AutoPas

c01 Base Step

The c01 base step uses only a single color, as illustrated in Figure 3.2a, 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 3.2b). 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 3.2c). This adjustment allows for better parallel processing and improved cache utilization.

3.1.5. 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 [7].

`lc_c08`

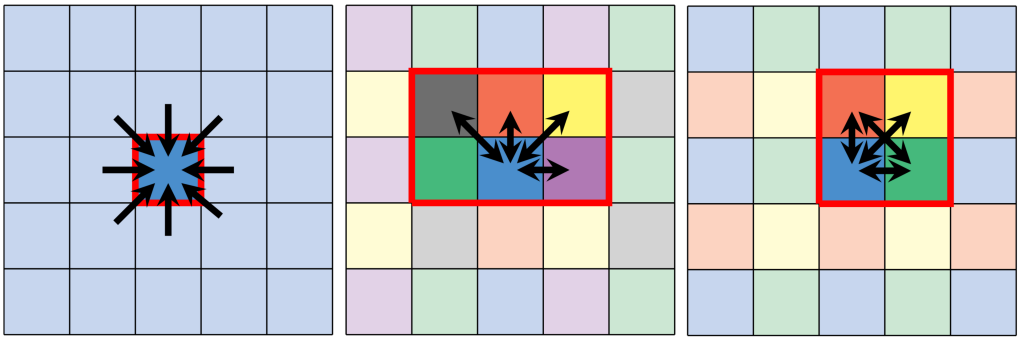
This traversal utilizes `c08` as its base step, and it is designed for the Linked Cells container. It divides a 2D domain into four colors and a 3D domain into eight colors. The dependency of the number of colors C on the number of dimensions D follows $C = 2^D$ [7].

`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. [4]

`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. [4]



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

Figure 3.2.: Base Steps Approaches [13]

3.1.6. 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 [8].

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.

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 [2]. 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 [3]. 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 [15]. 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.

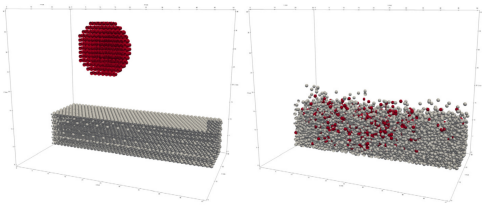


Figure 5.1.: Falling Drop [7]

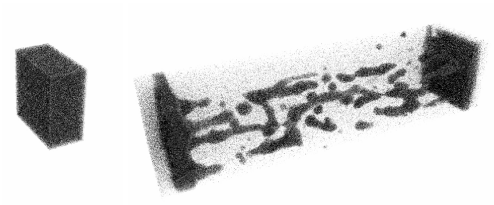


Figure 5.2.: Exploding Liquid [15]

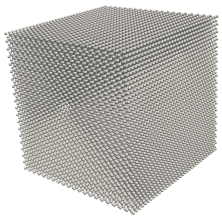


Figure 5.3.: Constant Cube

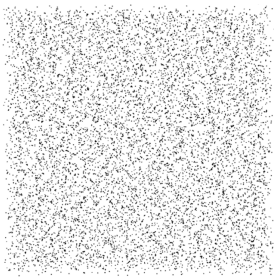


Figure 5.4.: Equilibration [14]

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. **??**

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

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

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.5.5), and neighbor list rebuild times per iteration (Fig.5.6), as well as the number of particles stored in the buffer per iteration (Fig.5.7).

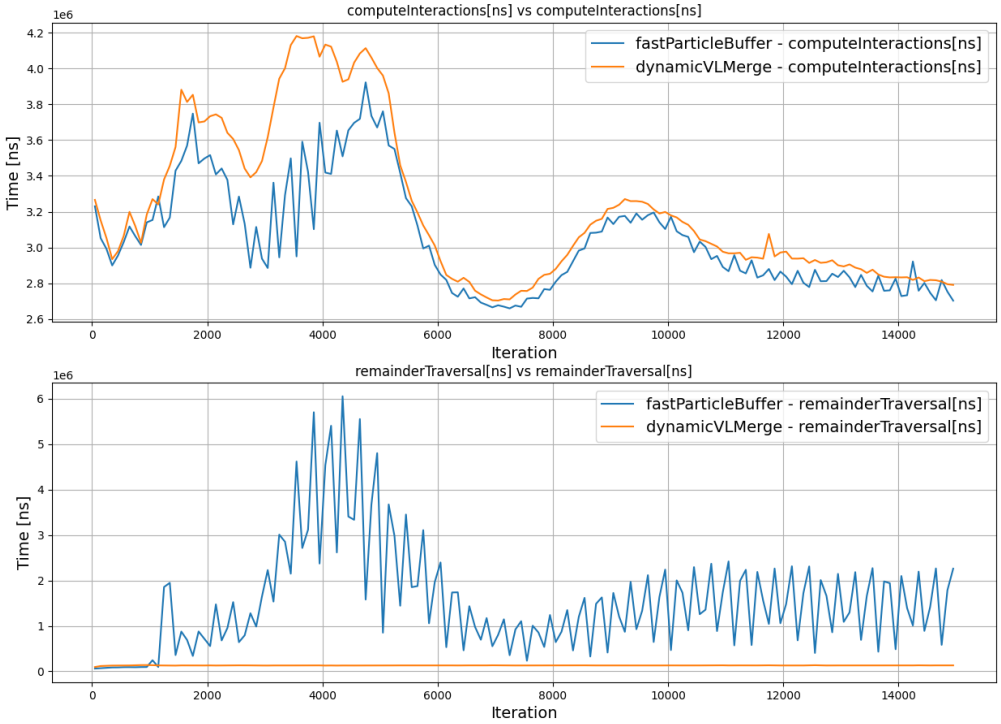


Figure 5.5.: Compute interactions versus Remainder traversal Falling Drop Vlc_c08

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

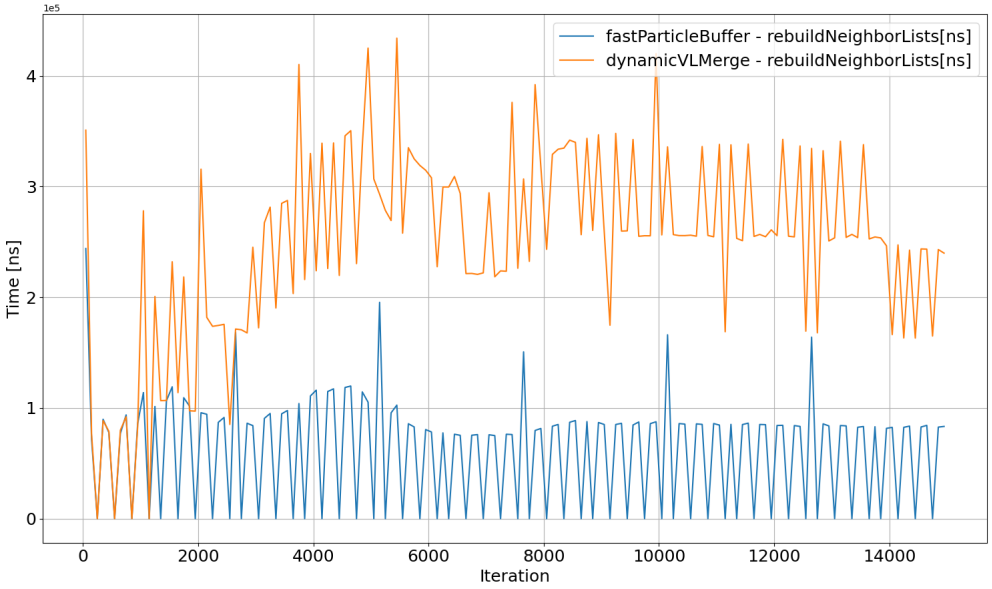


Figure 5.6.: Time spent rebuilding neighbor lists in Falling Drop Vlc_c08

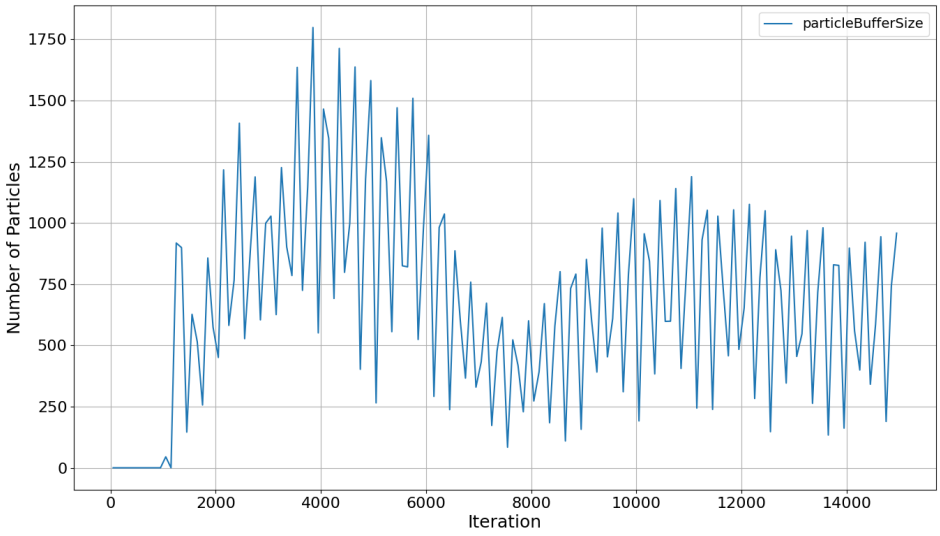


Figure 5.7.: Buffer Size throughout Iterations in Falling Drop Vlc_c08

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.5.6). 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` (Fig.A.2c) traversal. 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 did not provide sufficient insight into the performance of the fast particle branch. The results of these initial tests are presented in the appendix [**link appendix**].

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.

[**Note: Include** `computeInteractions` **graphs for** `vcl_c08` **and** `vcl_c06`]

In contrast, the `vcl_c06` traversal demonstrated similar performance for both branches. In some cases, the fast particle buffer was 1–2 seconds faster. This can be explained by the runtime differences visible in the graphs below. For `vcl_c06`, `computeInteractions` required nearly four times more runtime than `vcl_c08` (note the difference in scales: `vcl_c06` has an ordinate axis in $e7$, while `vcl_c08` is in $e6$). However, the remainder traversal times were comparable between the two. This outcome arises because the fast particle buffer, implemented as a vector of vectors, is not influenced by the container or traversal type. Despite the improved results in this specific experiment, no significant overall performance improvement was observed. Note that these graphs use a window average of 100 to reduce noise.

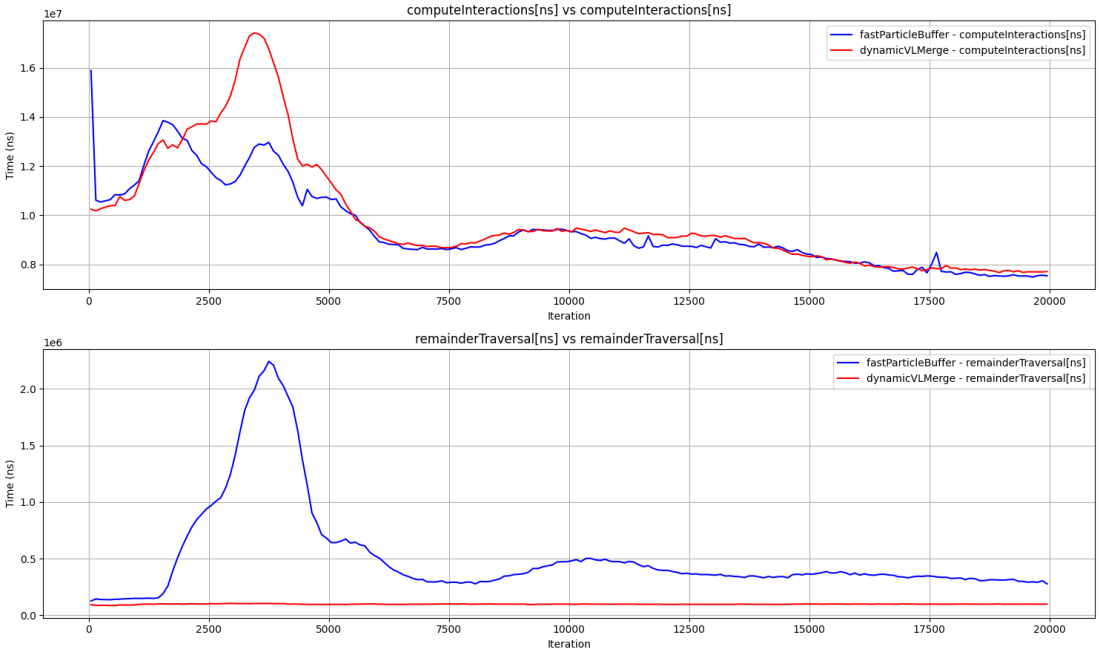


Figure 5.8.: Compute Interactions and Remainder Traversal in falling drop vclc06

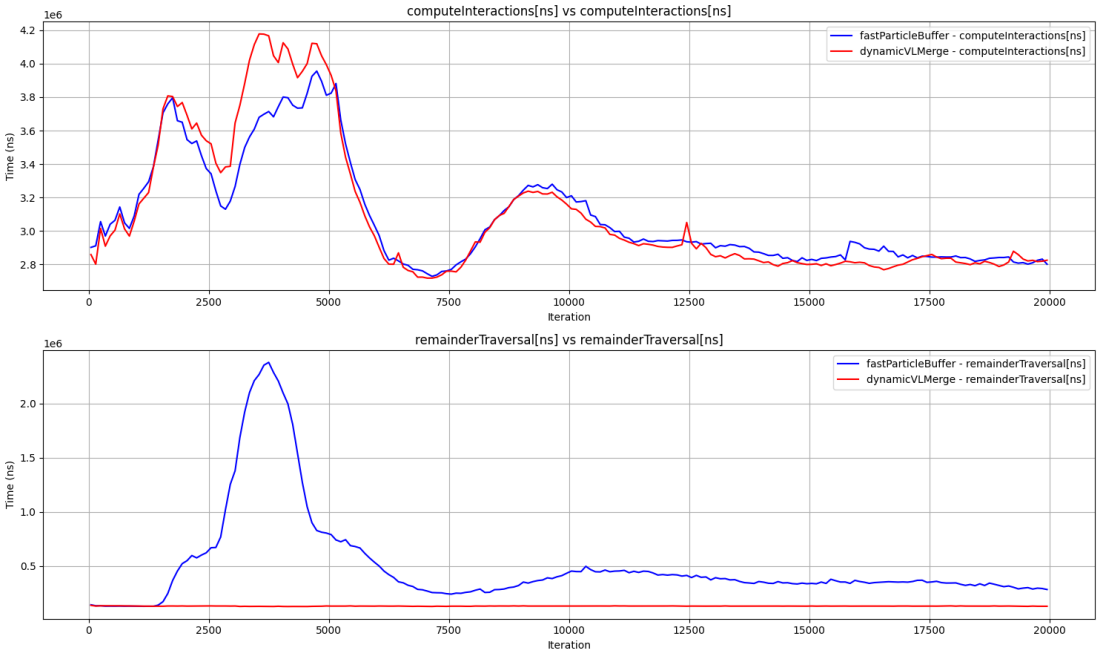


Figure 5.9.: Compute Interactions and Remainder Traversal in falling drop vlcc08

Exploding Liquid Scenario: The results for this scenario followed a pattern similar to Falling Drop. The performance trends of the traversals were comparable between the two scenarios. However, anomalies were observed in the behavior of `vlc_c08` and `vcl_c06`, as indicated in the graphs [**link Exploding Liquid graphs**]. Missing lines in the graphs suggest that the experiment failed for certain iteration counts in both branches. Although there were spikes in `vlc_c08`, where the fast particle buffer branch appeared to outperform the Dynamic VL Merge, the anomalies in the experiments render these results unreliable. Since both branches failed under certain conditions, these results cannot be considered valid. The `lc_c08` traversal, however, exhibited stable behavior with no failed runs.

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. [**Why is the performance so poor? Adjust the reasoning for choosing Constant Velocity Cube, as there is still significant particle interaction.**]

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. In the graph below, the runtime scale for remainder traversal is $e8$, while `computeInteractions` is on the $e7$ scale, showing a clear difference in time consumption.

The issue lies in the difference in optimization between the buffer and the container. Containers and traversals are designed to be efficient and parallelized, allowing particles to be processed quickly. In contrast, the fast particle buffer is a simple vector of vectors and is not optimized in the same way, leading to much higher computation times for remainder traversal when many particles are stored in the buffer.

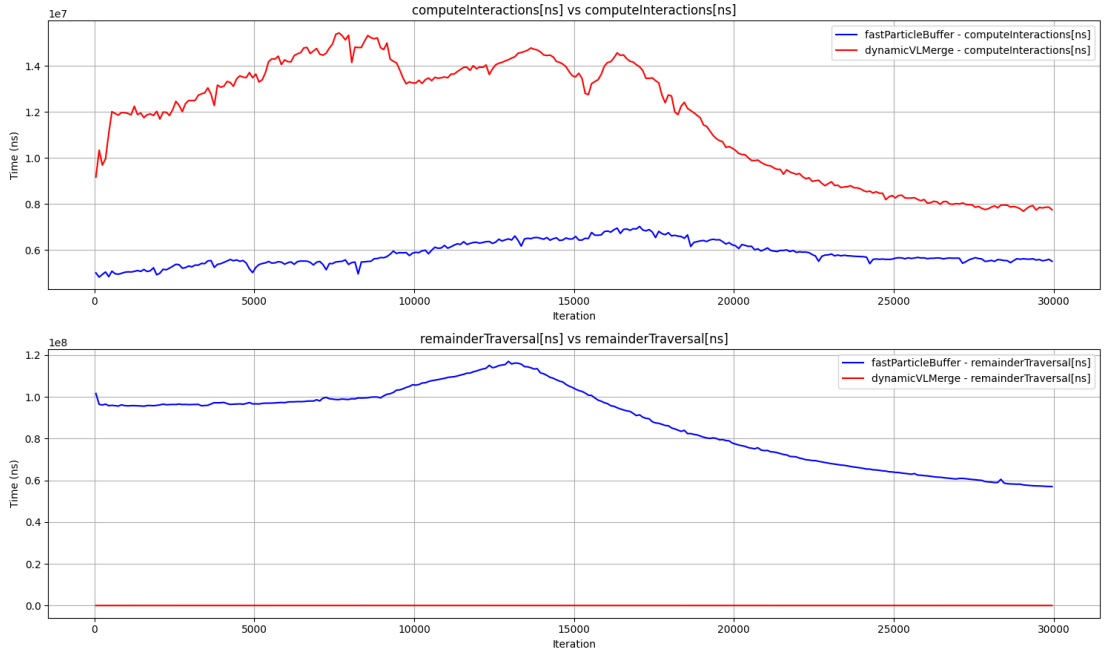


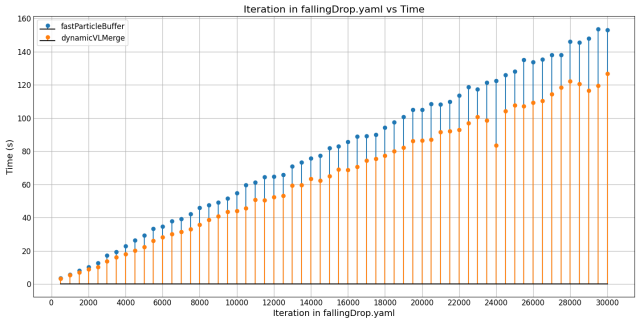
Figure 5.10.: Compute Interactions and Remainder Traversal in Constant Velocity Cube `vlp_c08`.

5.3.2. Percentage Experiments

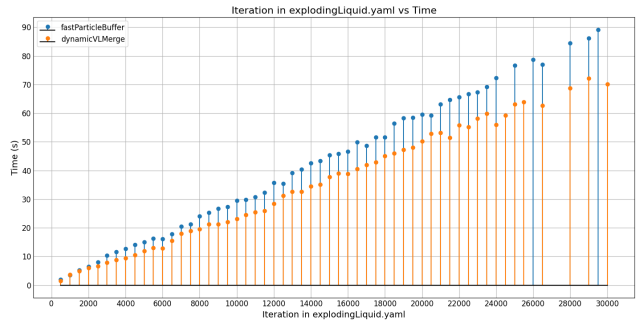
5.3.3. Spinodal Decomposition Equilibration

Given that the observed performance speedup in earlier experiments ranged between only 1 to 2 seconds, it raises the question of whether these results were merely coincidental or indicative of actual improvement. To address this, percentage-based experiments were conducted for the fourth scenario, spinodal decomposition equilibration. The hypothesis was that, due to the highly dynamic nature of this scenario and its substantial number of particles (4 million), there would be noticeable performance improvements when using the particle buffer.

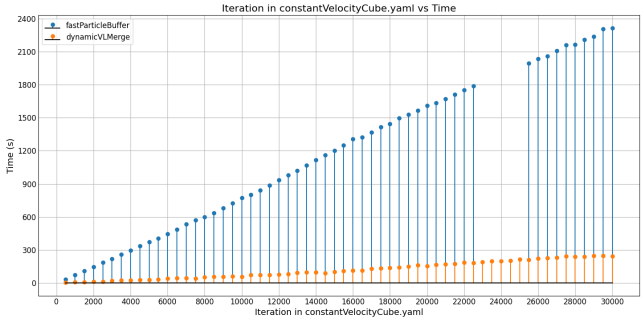
From prior frequency experiments, it was observed that the particle buffer performed better at smaller frequencies. Therefore, this experiment focused on frequencies ranging from 10 to 50 with a step size of 10. Additionally, as established earlier, smaller thresholds tended to yield better results. Considering the large particle count in spinodal decomposition, 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



(a) Frequency vs Time for Falling Drop



(b) Frequency vs Time for Exploding Liquid



(c) Frequency vs Time for Constant Velocity Cube

Figure 5.11.: Comparison of Frequency vs Time for Different Simulations

to 40,000 and 200,000 particles, respectively, which exceed the total particle count of the other scenarios.

Results for `vlc_c08`**:** This scenario was particularly demanding, with an average runtime of 9 hours and 59 minutes for the Dynamic VL Merge branch. 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.

Overall, the fastest runtime for this experiment was achieved with the fast particle buffer using a 5% threshold at frequency 30, taking 9 hours and 33 minutes. The second-best runtime was achieved by the fast particle buffer with a 1% threshold, which took 6 minutes longer (9 hours and 39 minutes). The lowest runtime for the Dynamic VL Merge branch was observed at frequency 40, with a runtime of 9 hours and 47 minutes. Across all frequencies and branches, the speedup achieved by the fast particle buffer, compared to the fastest Dynamic VL Merge result, was 14 minutes, focusing on achieving the fastest overall simulation runtime regardless of frequency.

Results for `lc_c08`**:** Using LinkedCellsReferences with traversal `lc_c08`, 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 the 5% threshold, a performance improvement of 2.3% (20 minutes) was observed at frequency 30. For all other frequencies, the runtime ranged from -0.95% to 0.29% compared to the Dynamic VL Merge branch, indicating negligible differences.

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.

Summary: The results demonstrate that performance improvements using the fast particle buffer are highly scenario- and configuration-dependent. While significant

speedups were observed at specific thresholds and frequencies, the benefits diminished or reversed at higher frequencies or with larger thresholds. The experiment highlights the importance of fine-tuning parameters to achieve optimal performance, especially in highly dynamic scenarios with large particle counts.

Spinodal Decomposition with Increased Temperature

To further amplify the number of fast particles and make the experiment more dynamic, the temperature of the system in the spinodal decomposition equilibration scenario was increased. This adjustment resulted in molecules with higher kinetic energy, thereby increasing their speed. The focus of this experiment was on frequencies 10, 20, and 30. The YAML file used for this setup can be found in the appendix [**link YAML file in the appendix**].

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 threshold of 1%, which achieved a 47-minute improvement compared to the fastest version of the Dynamic VL Merge branch at frequency 30.

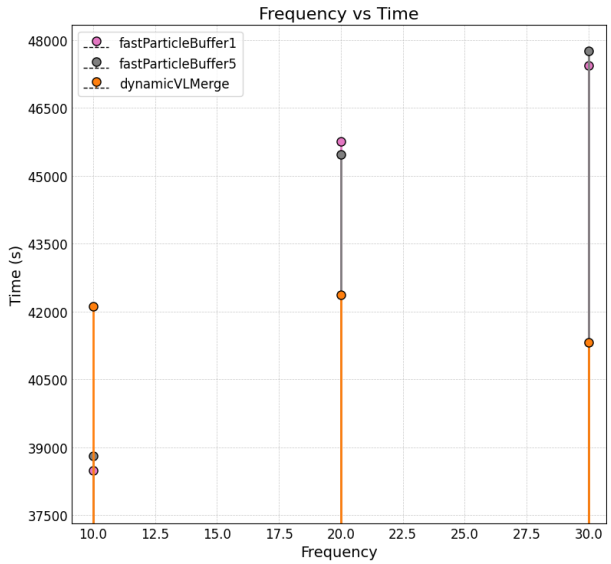


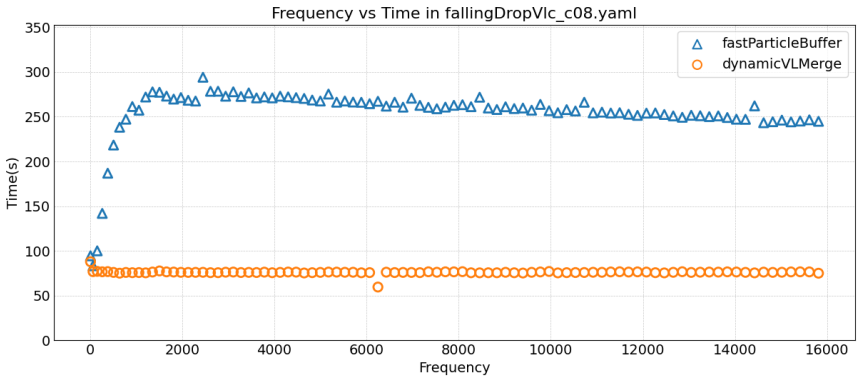
Figure 5.12.: Iterations vs Time for Equilibration vlc_c08 [**CHANGE GRAPH: include newer frequencies**]

5.4. Checkpoint Experiments

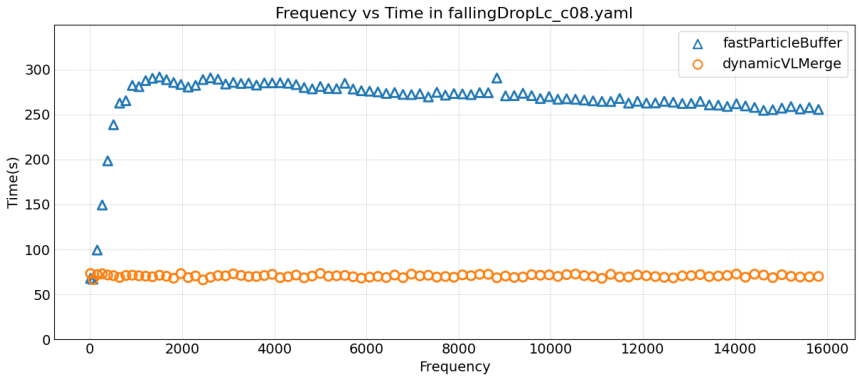
5.5. Hardware differences

6. Future Work

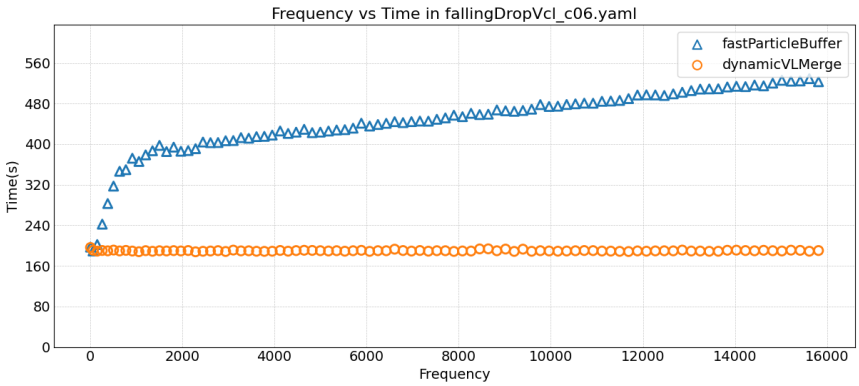
7. Conclusion



(a) Falling Drop vlc_c08

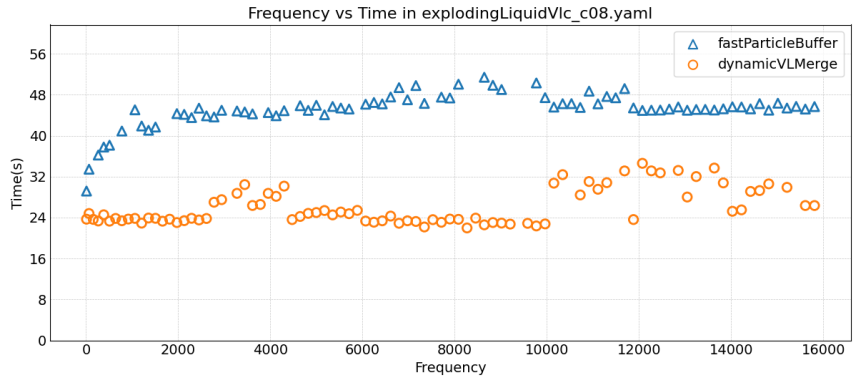


(b) Falling Drop lc_c08

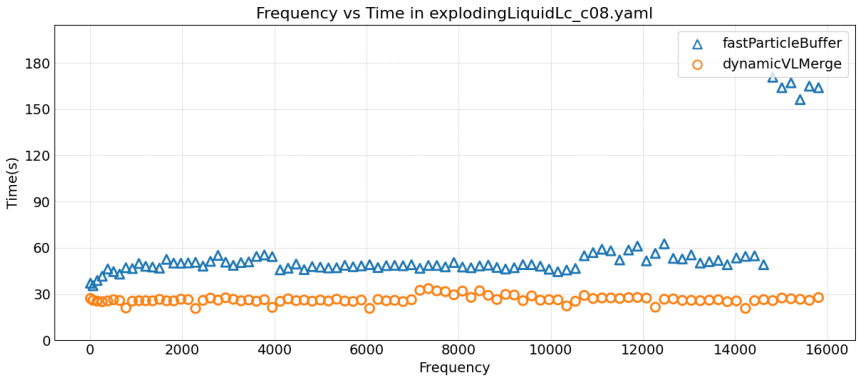


(c) Falling Drop vcl_c06

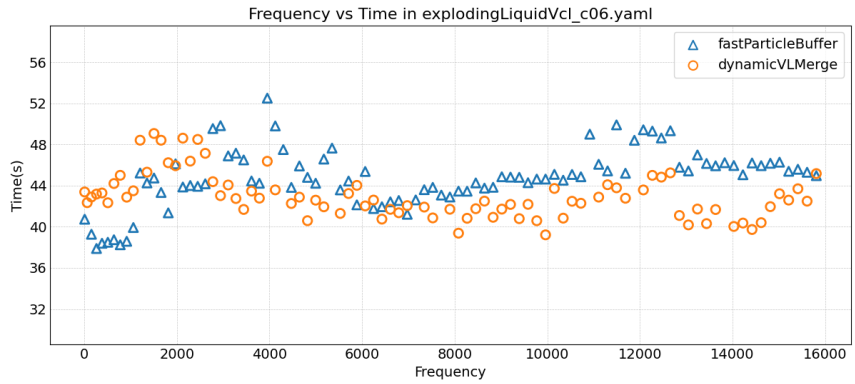
Figure A.1.: Comparison of Frequency vs Time for Falling Drop Experiments



(a) Exploding Liquid vlc_c08

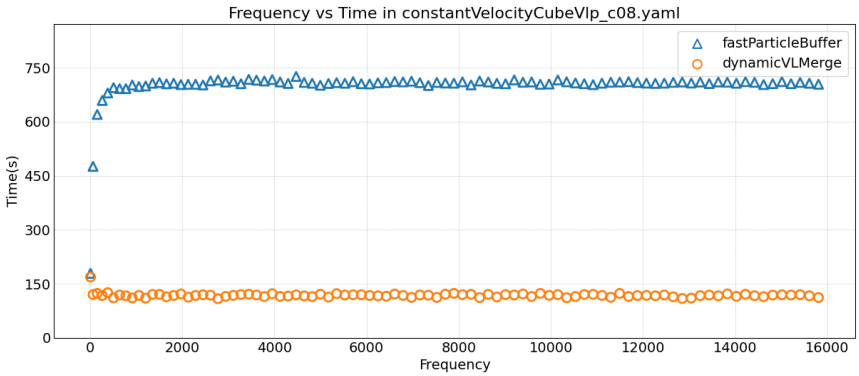


(b) Exploding Liquid lc_c08

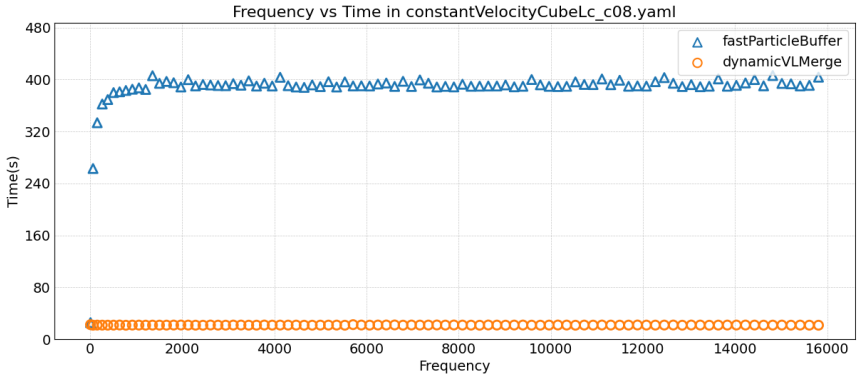


(c) Exploding Liquid vcl_c06

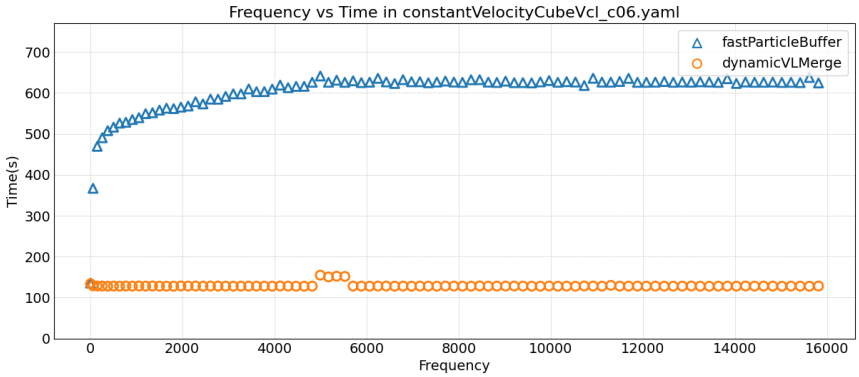
Figure A.2.: Comparison of Frequency vs Time for Exploding Liquid Experiments



(a) Constant Velocity Cube vlp_c08

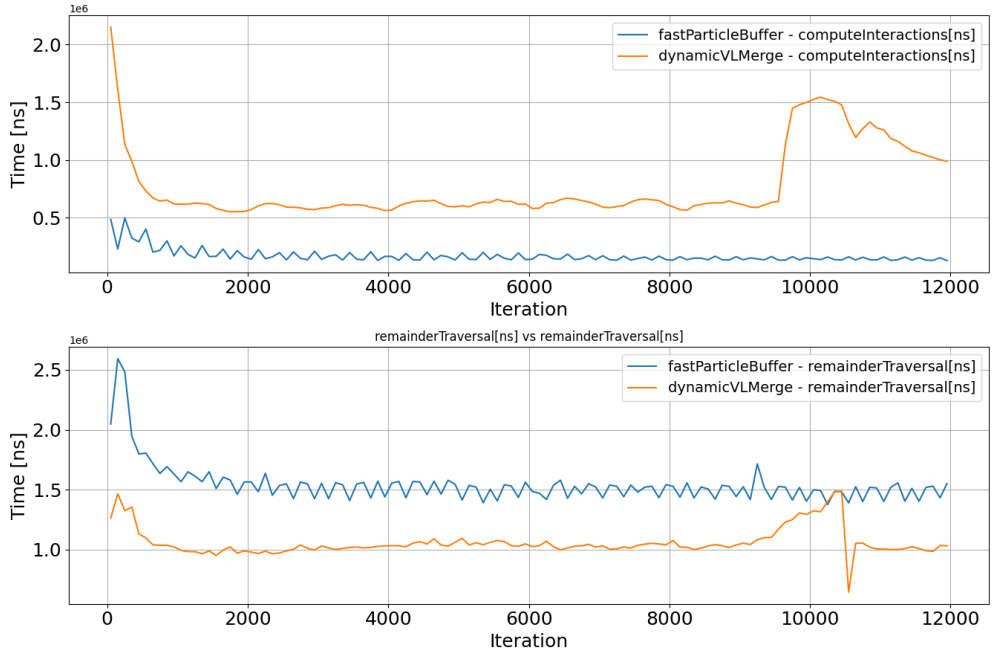


(b) Constant Velocity Cube lc_c08

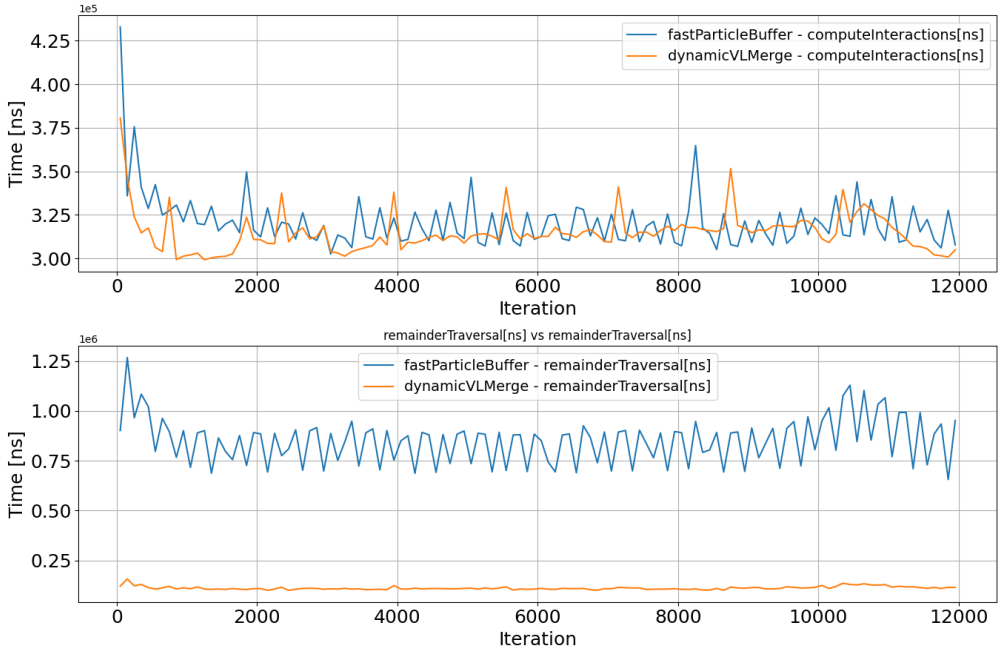


(c) Constant Velocity Cube vcl_c06

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



(a) Exploding Liquid Vcl_c06



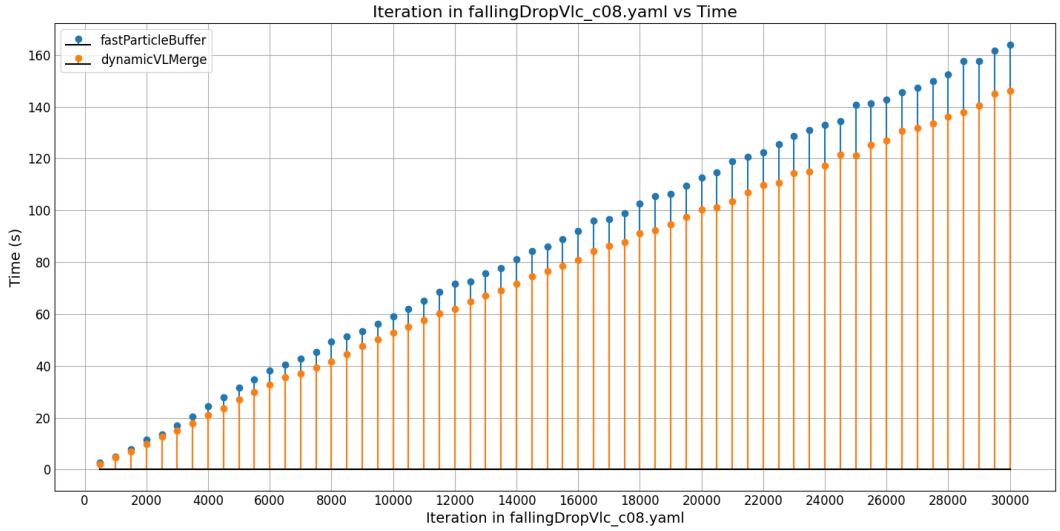
(b) Exploding Liquid Vlc_c08

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

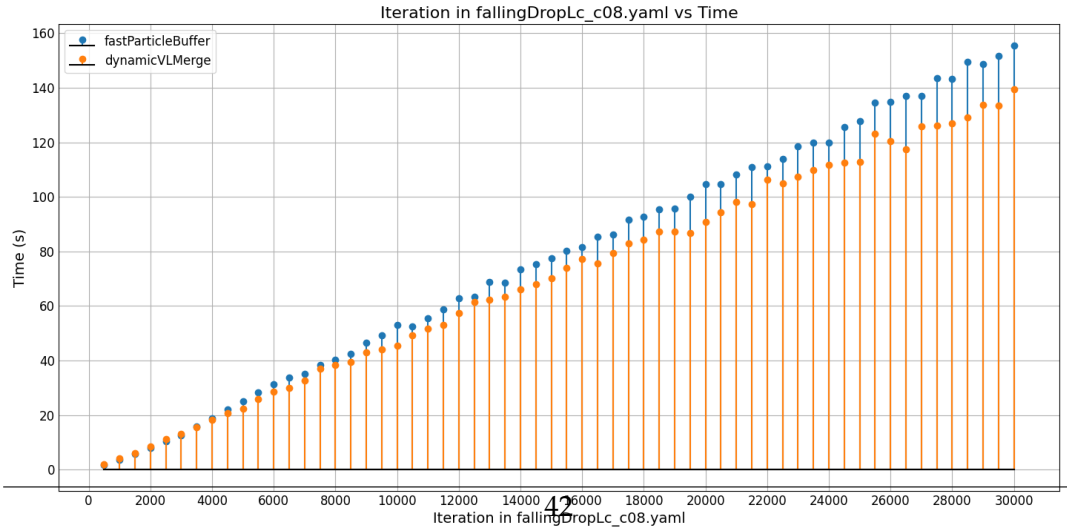
A. Appendix

A.1.

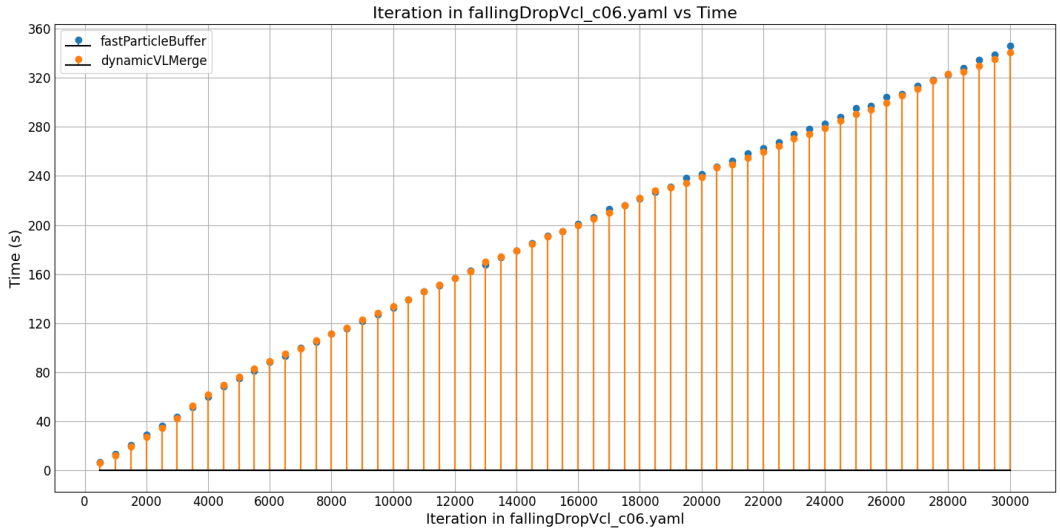
A.2.



(a) Iterations vs Time for Falling Drop vlc_c08

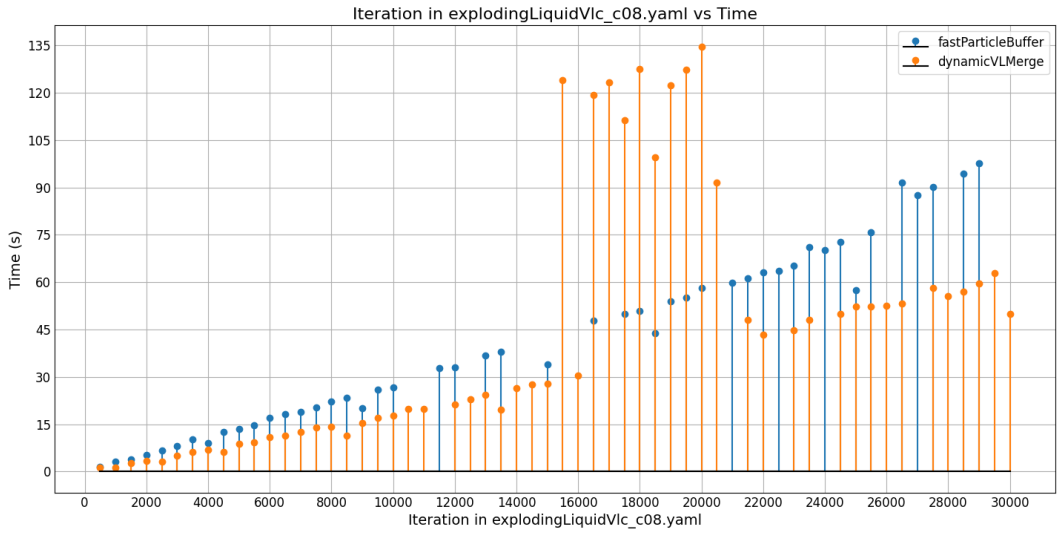


(b) Iterations vs Time for Falling Drop lc_c08

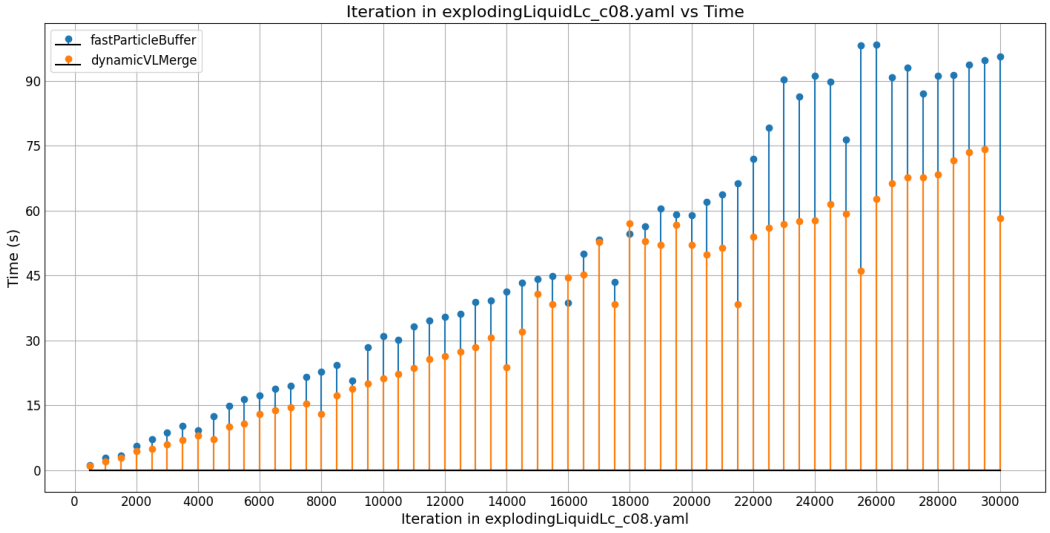


(c) Iterations vs Time for Falling Drop vcl_c06

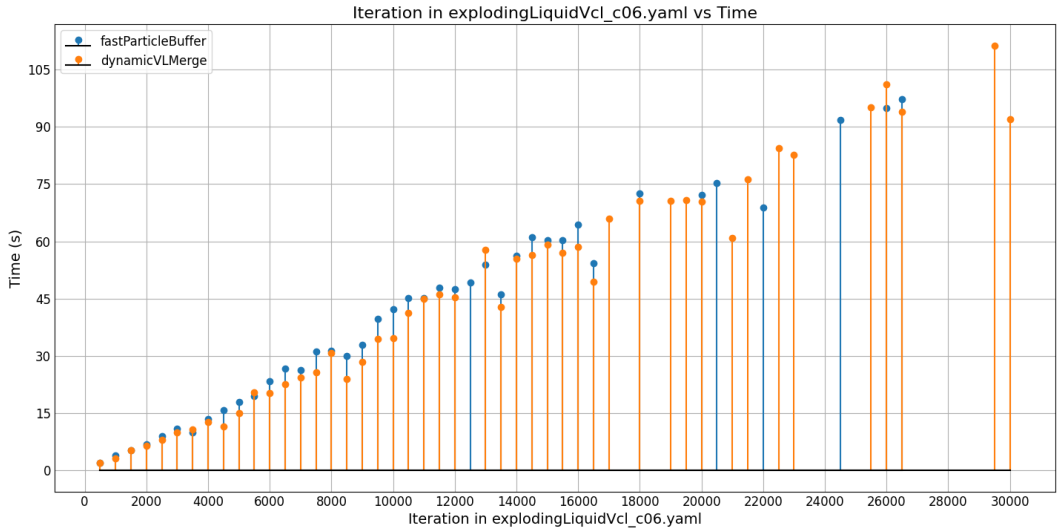
A.3.



(a) Iterations vs Time for Exploding Liquid vlc_c08

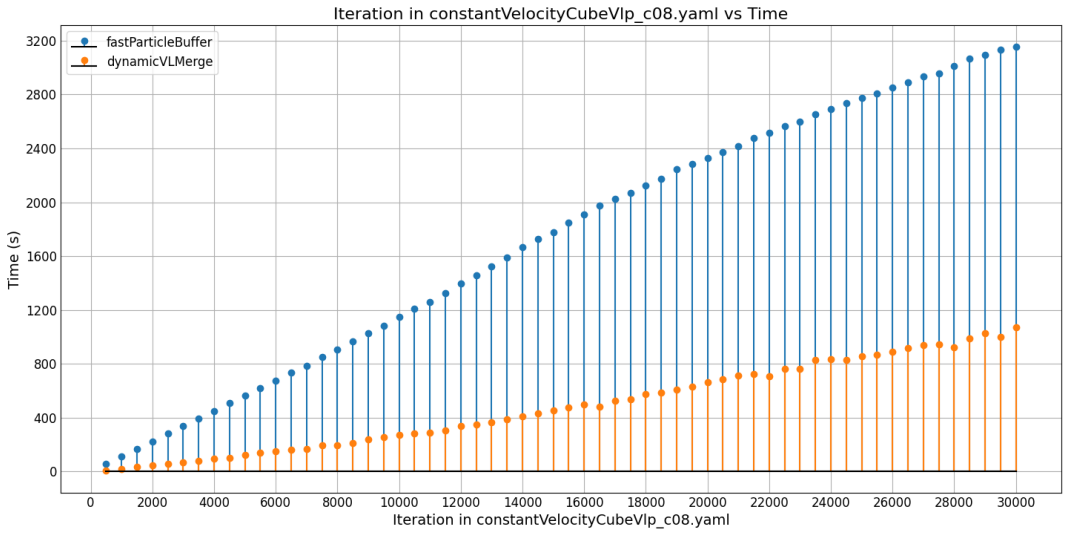


(b) Iterations vs Time for Exploding Liquid lc_c08

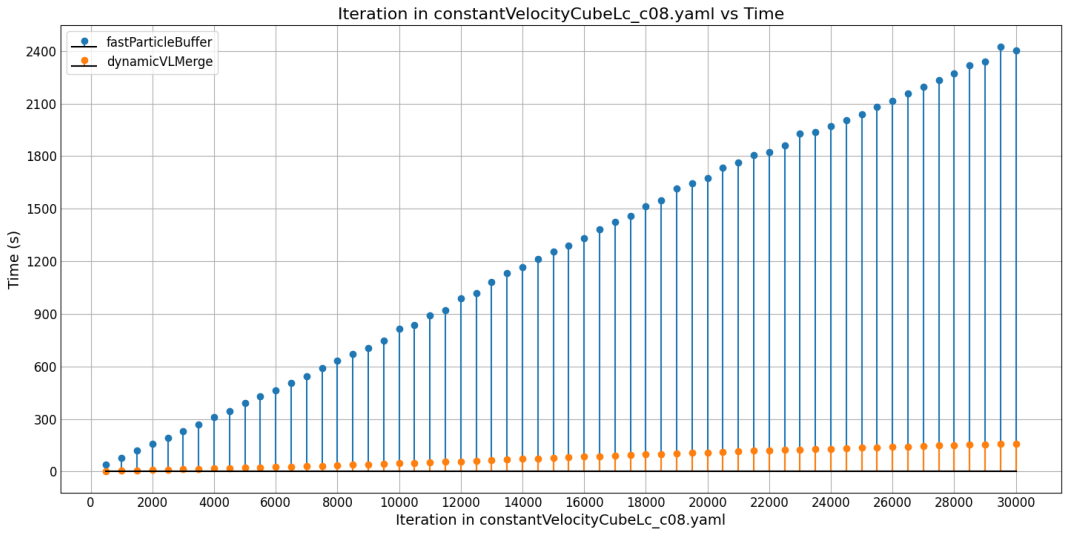


(c) Iterations vs Time for Exploding Liquid vcl_c06

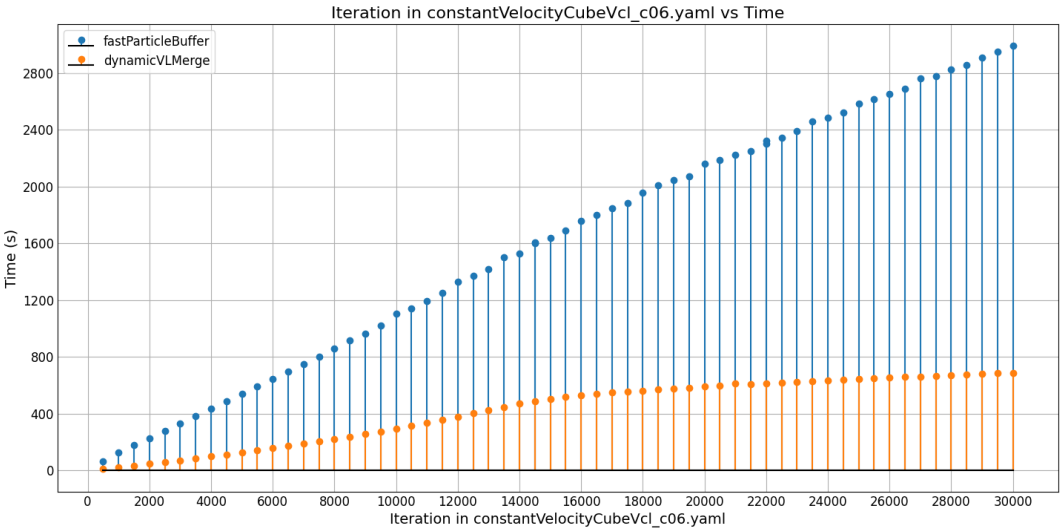
A.4.



(a) Iterations vs Time for Constant Velocity Cube vlp_c08

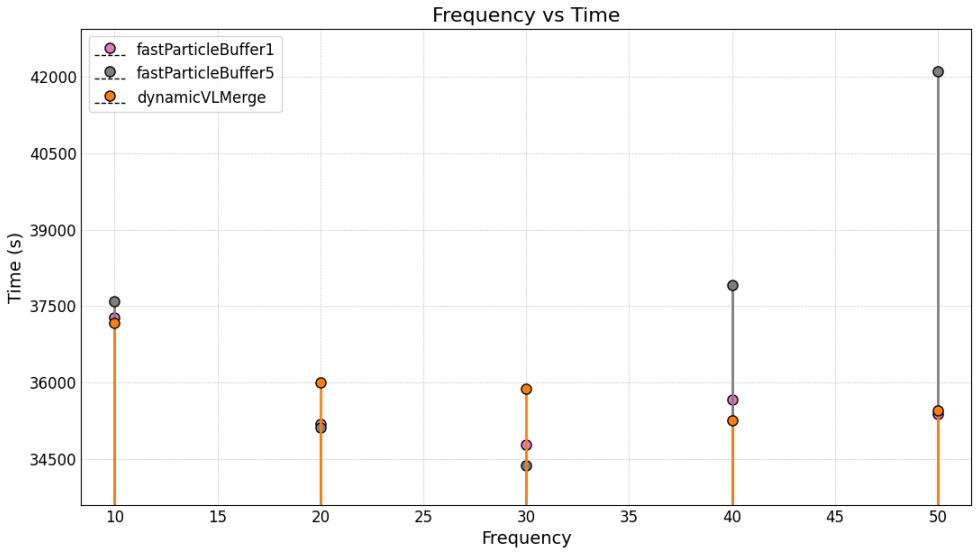


(b) Iterations vs Time for Constant Velocity Cube lc_c08

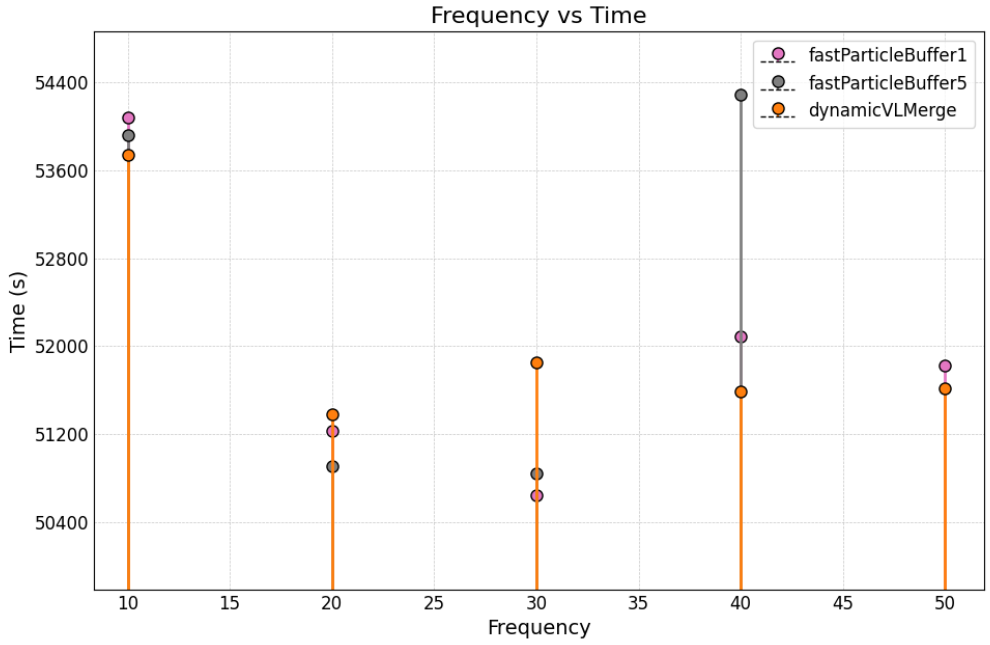


(c) Iterations vs Time for Constant Velocity Cube vcl_c06

A.5.



(a) Iterations vs Time for Spinodal Decomposition vlc_c08



(b) Frequency vs Time for Spinodal Decomposition lc_c08

List of Figures

List of Tables

Bibliography

[1] 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.

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

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

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

[5] 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.

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

[7] 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.

[8] 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 Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 748–757.

[9] 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 Mathematical and Physical Character* 106.738 (1924), pp. 463–477.

[10] M. G. S. Knapek and G. Zumbusch. *Numerical Simulation in Molecular Dynamics, Numerics, Algorithms, Parallelization, Applications*. 2007.

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

[12] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

[13] 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 Applied Mathematics* 433 (2023), p. 115278.

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

[15] 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.

[16] 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.

[17] 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.