FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Analysis of Load Balancing Options for AutoPas' Sliced Traversal

Vincent Fischer

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation and Analysis of Load Balancing Options for AutoPas' Sliced Traversal

Implementierung und Analyse von Lastbalanzierungsmöglichkeiten für das Sliced Traversal von AutoPas

Author:	Vincent Fischer
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	September 15th 2020

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

Munich, September 15th 2020 Vincent Fischer

Acknowledgements

I would like to thank all people who supported me while writing this thesis, first and foremost my advisor, Fabio Gratl. Thank you for introducing me to the fascinating field of molecular dynamics in an excellent lab course last semester and for our weekly meetings during this project. I would not have been able to solve many of the problems I encountered without your insight. Thank you also to my dear friends Matt and Martin, as well as to my parents, who proofread this thesis.

Abstract

AutoPas is a free and open-source library written in C++ providing multiple efficient algorithms for particle simulations. In order to fully exploit modern processors, it is necessary to distribute the computational load evenly among multiple threads. One of the shared-memory parallelization approaches implemented in AutoPas cuts the simulation domain into multiple equally large slices and assigns one slice to each thread. However, for inhomogeneous particle distributions, this does not distribute the computational load equally among the threads.

In this thesis, multiple variants of sliced traversal for different particle containers were implemented, which aid in balancing the computational load among the threads. The new traversals were benchmarked using two different particle simulations, which were run on the Haswell architecture based CoolMUC-2 cluster. The results show that each of the new traversal options outperforms the original unbalanced sliced traversal for the same container for at least one scenario. As AutoPas uses autotuning to determine the best traversal for the scenario, this results in an absolute improvement.

Zusammenfassung

AutoPas ist eine in C++ geschriebene, freie und quelloffene Softwarebibliothek, die mehrere effiziente Algorithmen für Partikelsimulationen bereitstellt. Um moderne Prozessoren voll auszunutzen, ist es notwendig, die Rechenlast gleichmäßig auf mehrere Threads zu verteilen. Einer der in AutoPas implementierten Ansätze zur shared-memory Parallelisierung zerschneidet die Simulationsdomäne in mehrere gleich große Slices und teilt jedem Thread ein Slice zu. Für inhomogene Partikelverteilungen teilt dies jedoch die Rechenlast nicht gleichmäßig auf die Threads auf.

In dieser Bachelorarbeit wurden mehrere Varianten dieses "sliced traversal" für verschiedene Partikelcontainer implementiert, welche dazu dienen, die Rechenlast besser zu verteilen. Die neuen Traversals wurden mittels zwei verschiedener Partikelsimulationen bewertet, welche auf dem auf der Haswell Architektur basierenden CoolMUC-2 Cluster durchgeführt wurden. Die Ergebnisse zeigen, dass jedes der neuen Traversals in mindestens einem Szenario performanter ist, als das ursprüngliche unbalanzierte sliced Traversal. Da AutoPas autotuning verwendet um das für das Szenario am besten geeignete traversal zu bestimmen, ist dies nur vorteilhaft.

Contents

Acknowledgements **vii**

Abstract **ix**

Zusammenfassung **xi**

I. Introduction and Background 1

1. Introduction **2**

2. Theoretical Background **3**
- 2.1. Intermolecular Forces 3
- 2.2. Improving upon the Run-Time of Direct Sum Computation 4
  - 2.2.1. Linked Cells 4
  - 2.2.2. Verlet Lists 4
  - 2.2.3. Verlet Lists Cells 5
  - 2.2.4. Verlet Cluster Lists 5
  - 2.2.5. Newton 3 Optimization 5
- 2.3. Algorithms for Parallel Traversal of Cell-Based Containers 6
  - 2.3.1. Base Steps 6
  - 2.3.2. Coloring Traversals 6
  - 2.3.3. Sliced Traversal with Locks 7

3. Overview of AutoPas' Code Structure **8**

II. Implementation of Different Load Balancing Options 9

4. Implementing a Version of Sliced Traversal for Verlet Cluster Lists **10**

5. Adapting Sliced Traversal to Allow Load Balancing **11**
- 5.1. Dynamic Scheduling 11
  - 5.1.1. Sliced Traversal with Two Colors 11
- 5.2. Heuristic-Based Balancing 11
  - 5.2.1. Load Estimation Heuristics 13

III. Analysis 15

6. Introduction to MD-Flexible 16

7. Comparing the Performance of the New Traversals 17
- 7.1. Simulation of Elongated Steinmetz Solids 17
  - 7.1.1. Theoretical Analysis 17
  - 7.1.2. Evaluating the Balancing Algorithm 19
- 7.2. Ostwald Ripening 22
  - 7.2.1. Per Iteration Run-Time Comparison 23
  - 7.2.2. Evaluating Load Estimation Heuristics 24

IV. Conclusion 28

8. Summary 29

9. Future Work 30

V. Appendix 31

A. Measurement Data 32

B. Simulation Config Files 33
- B.1. Steinmetz Simulation 33
- B.2. Ostwald Ripening Simulation 34

Bibliography 37

Part I.

Introduction and Background

1. Introduction

Molecular dynamics simulations play a significant role in modern science. They are heavily used in some branches of computational biochemistry and medicine, such as protein folding [Mia+15] and drug research. A recent example of this is testing the effectiveness of existing drugs against the SARS-CoV-2 virus [KSP20]. Many of these simulations include a very large number of particles, which makes solving their dynamic behavior analytically unfeasible. Instead, numerical methods are invoked.

In theory, the pairwise interactions between all particles have to be calculated leading to quadratic run-time. For short-ranged potentials, however, a cutoff can be introduced without sacrificing accuracy. To further increase the simulation speed parallel computation can be utilized.

AutoPas is a free and open-source node-level performance library written in C++, which provides efficient algorithms for solving arbitrary N-body simulations [Gra+19]. It utilizes OpenMP for shared memory parallelization. For short-ranged potentials, it provides linked cells and verlet list based containers for the efficient traversal of particle pairs. One of the possible parallelization strategies known as "sliced traversal" cuts the simulation domain into multiple equally large slices along its longest axis. One slice gets assigned to each thread. Locks are placed on the layers near slice boundaries to prevent race conditions. This works well for homogeneous particle distributions but fails to equally distribute the computational load among the threads if the particles are not evenly distributed.

In this thesis, three different variations of sliced traversal are implemented. The first two are based on OpenMPs dynamic scheduling option. Instead of creating one slice per thread, as many slices as possible are created and assigned to the threads dynamically. One of these two options uses locks to prevent race conditions, as does the original sliced traversal. The other instead uses an alternating coloring on the slices. Each color is then processed separately. The third approach stays at one slice per thread but tries to estimate the load beforehand and size the slices accordingly. To this end, two different load estimation heuristics were introduced, which can be chosen from using a tunable option.

Finally, the performance increase compared to the unbalanced sliced traversal was measured for two different simulations. In addition, the effectiveness of the balancing algorithm for the heuristic-based traversals was analyzed.

2. Theoretical Background

2.1. Intermolecular Forces

By Newton's second law of motion, we know that the center of mass of molecules must move according to their initial velocity and the forces acting on them. The motion can be approximately described by eqs. (2.1) and (2.2), where $\mathbf{v}$ is the velocity of a point particle, $\mathbf{F}$ is the force exerted on it and $\mathbf{r}$ is its position in three-dimensional Euclidean space.

$$\mathbf{v}(t) = \mathbf{v}_0 + \int_0^t \frac{\mathbf{F}(t')}{m} \mathrm{d}t' \tag{2.1}$$

$$\mathbf{r}(t) = \mathbf{r}_0 + \int_0^t \mathbf{v}(t') \, \mathrm{d}t' \tag{2.2}$$

The forces arise from a pairwise interaction potential; for the purpose of molecular dynamics simulations, we will be using the Lennard-Jones potential, which approximates the sum of several attractive and repulsive physical effects. The potential between particles i and j is described by [Len24]:

$$U_{ij} = 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] \tag{2.3}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between the two molecules. ϵ_{ij} is the minimum value that U_{ij} can take and therefore represents the depth of the potential well. σ_{ij} is the finite distance where $U_{ij} = 0$. The potential reaches its minimum ϵ_{ij} at $2^{\frac{1}{6}}\sigma_{ij}$. This is the point where attractive and repulsive forces are in equilibrium. Both constants can be calculated from molecule specific constants using a set of combining rules. A common example is to use the arithmetic mean for σ_{ij} and the geometric mean for ϵ_{ij}. These are known as the Lorentz-Berthelot rules, and are based on equations for noble gas molecules postulated by H. A. Lorentz [Lor81] and D. Berthelot [Ber98].

$$\epsilon_{ij} = \sqrt{\epsilon_i \epsilon_j} \qquad \sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} \tag{2.4}$$

Figure 2.1 shows the potential for $\sigma_i = \sigma_j = 0.5$ and $\epsilon_i = \epsilon_j = 1.5$.

The force exerted by particle i on particle j can then be derived by taking the gradient with respect to $\mathbf{r}_j$

$$\mathbf{F}_{ij} = -\nabla_j U_{ij} \tag{2.5}$$

The total force for molecule j can then be calculated by summing over all of the pairwise forces:

$$\mathbf{F}_j = \sum_{i \neq j} \mathbf{F}_{ij} \tag{2.6}$$

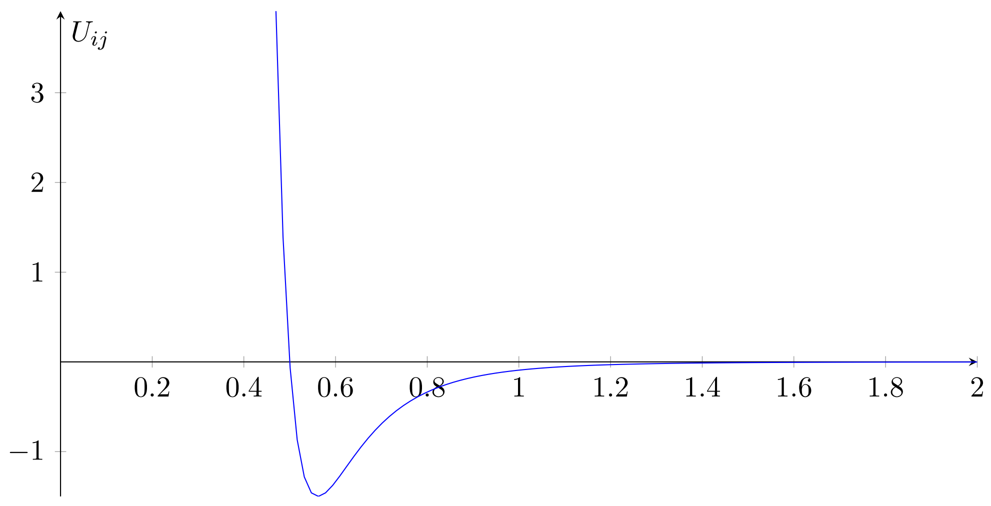


Figure 2.1.: Lennard-Jones potential visualized for $\sigma_i = \sigma_j = 0.5$ and $\epsilon_i = \epsilon_j = 1.5$.

2.2. Improving upon the Run-Time of Direct Sum Computation

The simplest approach to calculate the updated forces in each time step is summing the pairwise forces over all particle pairs. It has quadratic runtime. However, as the Lennard-Jones potential quickly drops off to zero with increasing r_{ij}, it suffices to consider particle pairs which are within a certain cutoff distance c of each other (for example $c := 2.5 \cdot \sigma_{ij}$, but may be different according to the scenario). On its own this doesn't cut down on the runtime though, as it is still necessary to calculate the distance between each particle pair. The data structures described in the following chapters aid in limiting the number of neighbors that need to be considered for each particle.

2.2.1. Linked Cells

For this algorithm, the domain is split into a 3D grid of evenly spaced cells. The particles are then sorted into these cells according to their position in space. Assuming the cell side length is equal to the cutoff length, it then suffices to consider the particle pairs that are in the same cell or in neighboring cells. If the cell side length is smaller than the cutoff length more cells will need to be considered. Assuming there exists a finite upper bound on the number of particles in each cell, this effectively reduces the runtime to $\mathcal{O}(n)$.

2.2.2. Verlet Lists

The Linked Cells algorithm has to consider all neighboring cells of the cell containing the current particle. This covers a volume of $(3 \cdot c)^3 = 27c^3$. The particles that need to be considered are however all contained within a sphere with radius c and thus volume $\frac{4\pi}{3}c^3$, which is only about 16% of the covered volume. The Verlet Lists approach, proposed by Loup Verlet in 1967 [Ver67], aims to further reduce the number of particles that need to be considered by building a list of neighboring particles every n time steps. One list is built for each particle with references to all particles within a distance of $c + s$. The skin depth s has to be chosen in such a way, that no particle will traverse a distance greater than $\frac{s}{2}$ in n time steps. This ensures that if two particles which are not considered to be neighbors when the lists get rebuilt are moving towards each other at maximum speed, they will also not get within

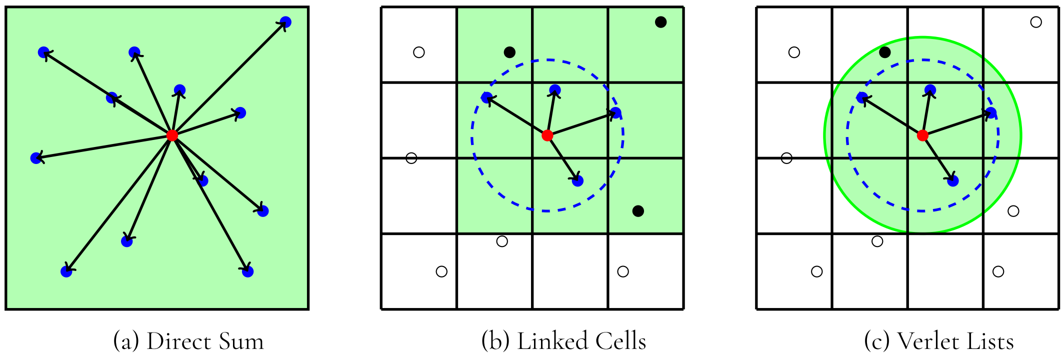


Figure 2.2.: Visualization of Direct Sum, Linked Cells, and Verlet Lists. Interactions between the red particle and all blue particles are calculated. For particles within the green area, the distance to the red particle has to be calculated. The dotted blue circle represents the cutoff radius.

distance c of each other until the next rebuild. The sum of cutoff and skin depth will be referred to as interaction length. The fraction of volume within the cutoff sphere to the volume covered by the traversal is then increased to $\frac{c^3}{(c+s)^3}$. A comparison of Direct Sum, Linked Cells, and Verlet Lists is visualized in Figure 2.2.

Note that while the original algorithm used the direct sum algorithm to rebuild the neighbor lists, in AutoPas an underlying Linked Cells container is used to further improve performance. This also provides a natural way of parallelization by assigning different cells to different threads.

2.2.3. Verlet Lists Cells

The standard Verlet Lists container provided by AutoPas stores the neighbor lists for each particle in one single list. The Verlet Lists Cells container (VLC) improves upon this by storing the neighbor lists of each particle within the cell the particle is currently in. This means list entries may need to be moved to a neighboring cell if a particle crosses the boundary, but it provides better parallelization support.

2.2.4. Verlet Cluster Lists

As Verlet Lists store a list of neighbors for each particle they have a greater memory requirement than the Direct Sum or Linked Cells algorithms. The Verlet Cluster Lists container (VCL) aims to reduce this overhead. The domain is split into towers in the x and y directions. Each tower contains equally sized clusters of particles that are close together. Instead of creating neighbor lists for every particle, every cluster stores a list of neighboring clusters. Two clusters are neighbors if one contains a particle that is within a distance of $c + s$ of a particle in the other cluster. This optimization of the Verlet Lists approach was first implemented for the software GROMACS [Abr+15].

2.2.5. Newton 3 Optimization

Newton's third law of motion states that when one body exerts a force on a second body, the second body also exerts a force of equal magnitude but opposite direction on the first body [New87]. For the purpose of molecular dynamics, this means it suffices to calculate the force between two particles i and j only once and add it to the total force exerted on i and subtract it from the total force exerted on j, effectively cutting the number of calculations in half. This optimization can be applied to all of the algorithms described in the previous sections.

2.3. Algorithms for Parallel Traversal of Cell-Based Containers

When using both multiple threads for the simulation and the optimization based on Newton's third law of motion, it is necessary to introduce some kind of synchronization to prevent race conditions. Otherwise, two threads could simultaneously try to update the force exerted on the same particle.

2.3.1. Base Steps

The base step of a traversal defines which particle interactions are calculated for each base cell. Without the Newton 3 optimization, all particle pairs where at least one particle is within the base cell have to be considered, which leads to a base step covering 27 cells. With Newton 3 optimization enabled, only half the number of neighboring cells need to be considered, as the other interactions are covered by applying the base step to the other cell. This base step covers only 14 cells, however, in order to be able to tesselate it, 18 cells have to be locked. For traversals of the Linked Cells container, we can further reduce the number of cells per base step to 8 if we don't require that all particle pairs contain at least one particle within the base cell. Instead, some of the interactions between diagonally neighboring cells are moved to different base steps.

These base steps will be referred to as c27, c18 and c08 respectively. Figure 2.3 shows a comparison of how the interactions are divided among the base steps for the latter two variants.

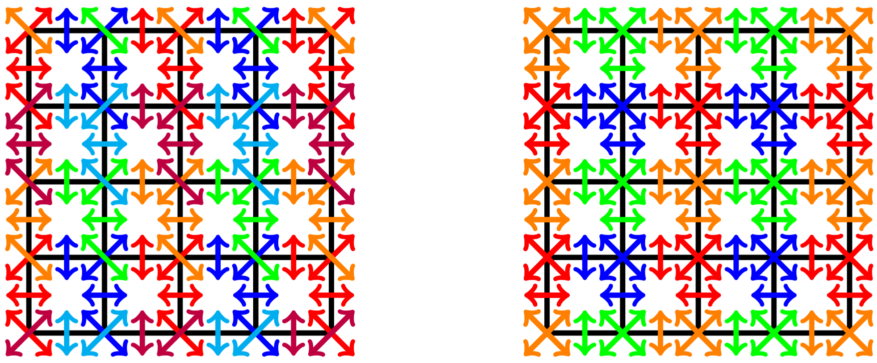


Figure 2.3.: comparison of c18 base step (left) and c08 base step (right). The colors mark the base cell to which the interaction is grouped.

2.3.2. Coloring Traversals

Coloring traversals arise naturally from the base steps described above. Each cell gets assigned a color in such a way that

1. no two neighboring cells have the same color.
2. the base step applied to two cells of the same color will never process interactions with the same third cell.

For example for the c18 and c08 base steps as visualized in Figure 2.3 each cell can be colored according to its most often occurring arrow color.

As described in Algorithm 1 the colors are then processed sequentially, but within each color, the base step can be applied to each cell in parallel.

Algorithm 1: Colored Traversal

```
1 for c in colors do
2     for cell in cells with color c do in parallel
3         processBaseStep(cell)
```

Note that cells with equal colors are usually spaced in regular intervals in all three dimensions, so the inner loop does not need to iterate over every cell, but use a stepsize with a color specific starting point to iterate over only the cells with that color. The key point to remember here is that processing one color can only start when the previous color is finished. In other words, all threads have to be joined after the inner loop. This, of course, causes some overhead every time, which is the reason we want to minimize the number of colors.

2.3.3. Sliced Traversal with Locks

This traversal algorithm uses locks to prevent race conditions instead of colors. The domain is cut into multiple slices along its longest dimension. The slices are then processed in parallel. Every slice locks its first layer of cells (or multiple layers if the cell side length is shorter than the interaction length). The lock is released, when the first layer is finished. When the last layer is reached, the starting layer of the next slice is locked. Each slice has to be at least two layers thick. Otherwise, that slice would immediately lock both its own starting layer and the starting layer of the next slice, which prevents the two slices from being processed in parallel. Algorithm 2 describes this in greater detail.

Algorithm 2: Sliced Traversal

```
1  for slice < numSlices do in parallel
2      for layer < sliceThickness[slice] do
3          if layer == 0 and slice != 0 then
4              locks[slice].lock()
5          if layer == sliceThickness[slice]-1 and slice < numSlices-1 then
6              locks[slice+1].lock()
7          for cell in currentLayer do
8              processBaseStep(cell)
9          if layer == 0 and slice != 0 then
10             locks[slice].unlock()
11         if layer == sliceThickness[slice]-1 and slice < numSlices-1 then
12             locks[slice+1].unlock()
```

3. Overview of AutoPas' Code Structure

In order to describe the changes made to AutoPas, it is first necessary to understand how the library is structured. The main interface to AutoPas is the *AutoPas* class. It takes a template parameter describing the particle type. The class provides the method *iteratePairwise()* which applies a pairwise functor given as a template parameter to every particle pair in the domain. Multiple configurations for the iteration can be chosen from. A configuration consists of a container for the particles, a traversal algorithm, a data layout, and whether to use Newton's third law to cut down on calculations. One container exists for each of the data structures described in Section 2.2. Multiple traversals can exist for every container. Their job is mainly to provide an efficient way to parallelize the pairwise iteration for the force calculation. While similarly working traversals can be implemented for multiple containers, they have to be implemented in separate classes for each container. The data layout can be either an "Array of Structures" (AoS) or a "Structure of Arrays" (SoA). Not all combinations of these options are applicable. For example traversals for the *VerletListsCells* container currently only support the AoS data layout.

The configuration used for an iteration is chosen by an autotuner. In regular intervals, it tries out different configurations, limited by a set of allowed choices for each option. Afterward, the best configuration is used for the next n iterations. The configurations tried in a specific block of tuning iterations are chosen by a search strategy, the simplest of which is the full search which simply measures the performance of all possible combinations of the allowed options. Some options, such as the cell side length, support infinite sets of allowed choices, which is not supported by this search strategy for obvious reasons. If an infinite set is supplied, other search strategies, such as a Bayesian Cluster Search, may be used, which can cope with these. Each chosen configuration is run multiple times within one block of tuning iterations. This helps compensate for random fluctuations and also ensures that only a minority of the runs will be affected by container- or neighbor list rebuilds.

Part II.

Implementation of Different Load Balancing Options

4. Implementing a Version of Sliced Traversal for Verlet Cluster Lists

Previously sliced traversal was implemented for the *LinkedCells* and *VerletListsCells* containers. Both versions inherit most of their functionality from the common base class *SlicedBasedTraversal*, which provides methods for finding and slicing the longest dimension, and iterating over the particle pairs. The latter function requires a base step to be passed as a parameter, which when applied to all cells in a container calculates all pairwise forces that are in range. For the *LinkedCells* container, the c08 base step is used. For the *VerletListsCells* container the base step simply iterates over the neighbor list of each particle in a cell. This is equivalent to a c18 base step. In the *VerletClusterLists* container the towers are interpreted as cells. This means there is only one cell in the z-dimension, and slicing is only possible in the other two dimensions. In order to adapt sliced traversal to this container, a new subclass of *SlicedBasedTraversal* had to be implemented, which provides an appropriate base step. The container already provides the two functions *traverseCluster(cluster)* and *traverseClusterPair(cluster1, cluster2)* to iterate over particle pairs within one cluster or two neighboring clusters respectively. These were used to construct the base step described by Algorithm 3. It iterates over every cluster in a tower and calls *traverseCluster(cluster)* on it, as well as *traverseClusterPair(cluster, neighbor)* on it and each of its neighbors.

Algorithm 3: Base step for Verlet Cluster Lists

```
1 for cluster in tower do
2     traverseClusterPair(cluster)
3     for neighbor in cluster.neighbors do
4         traverseClusterPair(cluster, neighbor)
```

5. Adapting Sliced Traversal to Allow Load Balancing

The sliced traversal algorithm provided by AutoPas previously created one slice for every OpenMP thread, with all slices being the same thickness. If not enough layers are present, fewer sliced will be created. While this works well for scenarios with relatively even particle distributions, it does not evenly distribute the load among the threads in inhomogeneous scenarios. In order to solve this problem, two different approaches were taken and implemented in AutoPas. The first is based on the dynamic scheduling option of OpenMP and is split into two variants. The second uses heuristics to estimate the load beforehand. All of the traversals described in the following sections inherit from a common base class. A class diagram illustrating the relevant dependencies is shown in fig. 5.1.

5.1. Dynamic Scheduling

Previously AutoPas' sliced traversal used OpenMP's static scheduling option. OpenMP however also provides a dynamic scheduling option, which assigns tasks to threads dynamically at runtime. This approach simply replaces static with dynamic scheduling. For balancing purposes many small tasks work better than a few large ones, as there are more options to divide them among the threads and the imbalance among the threads is limited by the longest task. For this reason, the calculation of slice thicknesses was modified to create as many slices as possible while respecting the minimum slice thickness. From now on this traversal will be referred to as "dynamic sliced traversal".

5.1.1. Sliced Traversal with Two Colors

Creating the maximum number of slices possible results in half of the layers having locks on them, which could potentially lead to a significant slowdown. For comparison purposes, another version of sliced traversal was implemented, which does not use locks. Instead, the slices are colored in an alternating fashion. Each color is separately processed. This results in a hybrid of the two algorithms described in Section 2.3. Since a lot of the functionality of this traversal and the original sliced traversal are identical, this was refactored into a common base class *SlicedBasedTraversal*. The original *SlicedBasedTraversal* was renamed to *SlicedLockBasedTraversal*. The new sliced traversal with colors is provided by the class *SlicedC02BasedTraversal*.

5.2. Heuristic-Based Balancing

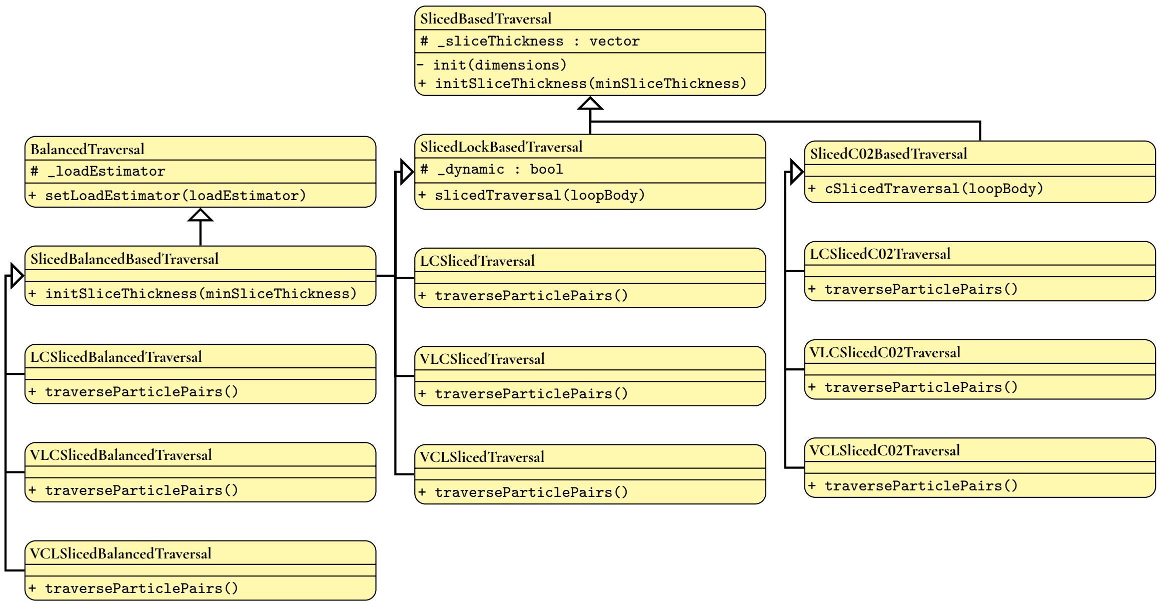
The idea of the second approach, as opposed to the two dynamic variants described in Section 5.1, is to use a heuristic to estimate the number of interactions for each layer. The slice thicknesses can then be chosen, to give each slice an equal fraction of the total computational load. The time required to calculate the heuristic should be low compared to the time required to calculate one iteration of force updates. Otherwise, the potential speedup gained by better distributing the load could be negated by the additional time required for this optimization. Different options for this heuristic will be discussed in Section 5.2.1.

To implement this feature in AutoPas a new subclass of *SlicedLockBasedTraversal* was introduced, named *SlicedBalancedBasedTraversal*. It overrides the classes *initTraversal()* method, which is responsible for setting up the slice thicknesses. The class also inherits from the new class *BalancedTraversal* which provides a member representing the load estimation function as well as the method *setLoadEstimator(EstimationFunction)*. This method has to be called before *initTraversal()*. The calculation of load thicknesses is described in Algorithm 4. The while loop on line 11 increases the thickness of the current

Algorithm 4: Calculation of slice thicknesses

```
1  for $i = 0$ ; $i < numLayers$ do
2      loads[i] = estimateLoadForLayer(i)
3  for $i \geq 1$ ; $i < numLayers$ do
4      loads[i] += loads[i-1]
5  remainingLoad = loads[numLayers - 1]
6  totalThickness = 0
7  avg = remainingLoad / numSlices
8  for $s < numSlices$ do
9      thickness = minSliceThickness
10     currentSliceLoad = loads[totalThickness + thickness - 1] - loads[totalThickness]
11     while totalThickness + thickness $<$ numLayers and currentSliceLoad $<$ avg do
12         currentSliceLoad = loads[totalThickness + thickness - 1] - loads[totalThickness]
13         increasedSliceLoad = loads[totalThickness + thickness] - loads[totalThickness]
14         if abs(avg - currentSliceLoad) $<$ abs(avg - increasedSliceLoad) then
15             break
16         else
17             thickness += 1
18     if totalThickness + thickness $>$ numLayers then
19         sliceThickness[s-1] += numLayers - totalThickness
20         numSlices = s
21         break
22     sliceThickness[s] = thickness
23     totalThickness += thickness
24     remainingLoad = loads[numLayers - 1] - loads[totalThickness]
25     remainingSlices = numSlices - s - 1
26     avg = remainingLoad / remainingSlices
```

slice until the total estimated load for that slice is as close to the average remaining load per slice as possible. Lines 18 - 21 handle the case that by setting the thickness of the current slice to the minimum possible slice thickness (see line 9), the total number of layers was already exceeded. Recall that the minimum slice thickness is defined by the ratio of interaction length to cell length. In this case, the number of slices is decreased and the remaining layers added to the last slice.



SlicedBasedTraversal
# _sliceThickness : vector
- init(dimensions)
+ initSliceThickness(minSliceThickness)
BalancedTraversal
# _loadEstimator
+ setLoadEstimator(loadEstimator)
SlicedLockBasedTraversal
# _dynamic : bool
+ slicedTraversal(loopBody)
SlicedC02BasedTraversal
+ cSlicedTraversal(loopBody)
SlicedBalancedBasedTraversal
+ initSliceThickness(minSliceThickness)
LCSlicedTraversal
+ traverseParticlePairs()
LCSlicedC02Traversal
+ traverseParticlePairs()
LCSlicedBalancedTraversal
+ traverseParticlePairs()
VLCSlicedTraversal
+ traverseParticlePairs()
VLCSlicedC02Traversal
+ traverseParticlePairs()
VLCSlicedBalancedTraversal
+ traverseParticlePairs()
VCLSlicedTraversal
+ traverseParticlePairs()
VCLSlicedC02Traversal
+ traverseParticlePairs()
VCLSlicedBalancedTraversal
+ traverseParticlePairs()

Figure 5.1.: Class Diagramm showing the different sliced traversal variations.

5.2.1. Load Estimation Heuristics

Different algorithms were implemented to estimate the load for a specific cuboid region within the domain. In order to choose between them, a new tunable option was added to AutoPas and integrated into the different tuning strategies.

The first estimation algorithm is trivial and simply returns 1 for every region. It serves multiple purposes: Firstly to preserve the functionality of the original sliced traversal. This is sensible not only as a baseline for runtime comparisons but also since more complicated calculations should not be necessary if the particle distribution is relatively even. Secondly, the tuning algorithms implemented in AutoPas do not support some traversals requiring additional options. The "none"-heuristic solves this by being the only supported option for all other traversals.

A very intuitive way to estimate the load is to count the number of interactions that need to be calculated for each particle within the region. This would take excessively long for the Linked Cells container, but for the Verlet Lists Cells container, a very good approximation is to add up the lengths of the neighbor lists. Due to the skin depth, this is a slight overestimation, but the correlation with the actual value is sufficient for balancing purposes. The same approach can be used for the Verlet Cluster Lists container. While the neighbor lists here contain references to the neighboring clusters instead of to particles, the estimation still works, as each cluster contains the same number of particles. This heuristic will be referred to as "Neighbor List Length" (NLL).

The final estimation algorithm assumes that most particles that are within one cell will have interactions with each other. Interactions with neighboring cells are ignored. The heuristic simply sums up the squared number of particles for each cell in the region. This is not as accurate as the previous heuristic but has the benefit of being applicable to the Linked Cells container. It is also faster to calculate for the Verlet Lists Cells container if the number of particles is significantly greater than the number of cells. One should also note that the assumptions made are only sensible if the interaction length and the cell length are equal. For this reason, this algorithm is not available for the Verlet Cluster Lists container, as each cell occupies the complete z-dimension of the domain. This heuristic will be referred to as "Squared Particle per Cell" (SPpC)

Part III.

Analysis

6. Introduction to MD-Flexible

MD-Flexible is a simple particle simulation program built on top of the AutoPas library. Among other useful options, it provides object generators, which create 3D shapes filled with particles, a velocity-scaling thermostat to control the total kinetic energy in the system, and the option to load particles from a checkpoint. The configuration of a simulation consists of a YAML file specifying firstly the different objects to be simulated and secondly the options that are passed to AutoPas. The latter include, but are not limited to the interaction potential that is used via the functor option, the cell size for Linked Cells and Verlet Lists Cells containers, the allowed containers and traversal options that may be used, and the number and frequency of tuning iterations. The current state of the simulation can be written out to VTK files at regular intervals. By setting the log level to "debug" it is also possible to gain detailed information about each iteration of the simulation, such as the complete configuration used, including the traversal, and how long it took. For the different variations of sliced traversal, the time spent in each specific slice is also included in the debug log. This makes performance analysis and comparison of the different balancing options possible.

7. Comparing the Performance of the New Traversals

In the following sections, the performance of the new traversal options will be evaluated using two different scenarios. For this purpose, we will be using the version of sliced traversal balanced with the "none"-heuristic as a baseline, as it results in identical behavior to the original sliced traversal, and differs only in how the slice thicknesses are calculated.

7.1. Simulation of Elongated Steinmetz Solids

The first scenario was specifically designed to test the balancing algorithm. It consists of an elongated Steinmetz solid[1] made up of 110239 particles. The particles are arranged in a series of layers, each layer consisting of a square of particles arranged in a grid. Figure 7.1 shows a visualization of this object. A listing of the python script used to generate the objects for md-flexible can be seen in Listing B.1.

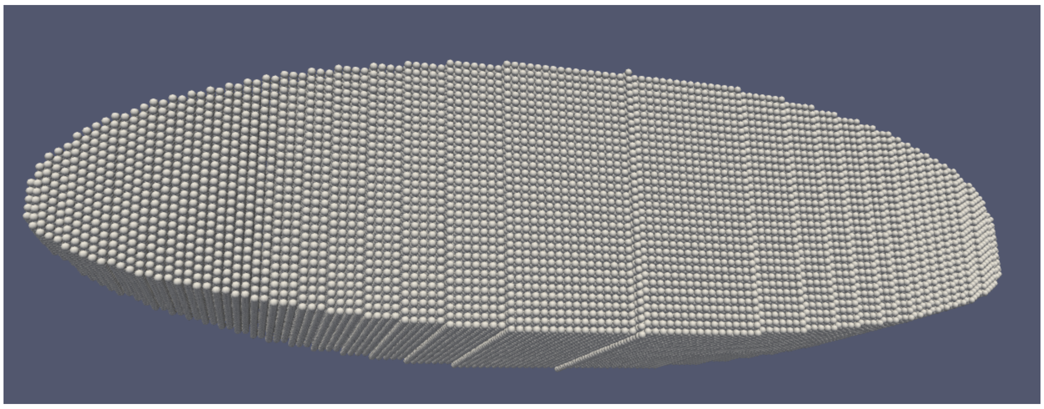


Figure 7.1.: Visualization of the elongated Steinmetz solid

7.1.1. Theoretical Analysis

As the Steinmetz solid is constructed in such a way, to have a uniform particle density, each particle has roughly the same amount of neighbors, apart from those at the very edge. Due to the size of the simulation, these should however only make up a small fraction of all the particles and can be safely ignored. When iterating with sliced traversal, the load on each slice should therefore be proportional to the fraction of volume of the Steinmetz solid covered. If l is the length of the solid and w its width, the intersection area A when cutting through a plane normal to its longest axis at a distance x from one end

[1]A Steinmetz solid is the intersection of two cylinders of equal radius rotated to each other by 90°

is given by the following equation

$$A\left(x\right) = w^2 \cdot \left(1 - \left(\frac{2x - l}{l}\right)^2\right) \tag{7.1}$$

The volume between two planes at a and b is therefore given by the integral

$$\int_a^b A\left(x\right) \mathrm{d}x = \int_a^b w^2 \cdot \left(1 - \left(\frac{2x - l}{l}\right)^2\right) \mathrm{d}x = \tag{7.2}$$

$$= \int_a^b w^2 \cdot \left(1 - \frac{4x^2 - 4lx + l^2}{l^2}\right) \mathrm{d}x = \tag{7.3}$$

$$= \left[w^2 \cdot \left(x - \frac{\frac{4}{3}x^3 - 2lx^2 + l^2 x}{l^2}\right)\right]_a^b = \tag{7.4}$$

$$= w^2 \cdot \left(\frac{2}{l}\left(b^2 - a^2\right) - \frac{4\left(b^3 - a^3\right)}{3l^2}\right) \tag{7.5}$$

For this scenario, four threads were used. With the heuristic set to none the balanced sliced traversal creates four equally large slices which correspond to the following volumes:

$$V_0 = w^2 \cdot \left(\frac{2}{l}\left(\left(\frac{l}{4}\right)^2 - 0^2\right) - \frac{4\left(\left(\frac{l}{4}\right)^3 - 0^3\right)}{3l^2}\right) = \frac{5}{48} \cdot w^2 l \tag{7.6}$$

$$V_1 = w^2 \cdot \left(\left(\frac{2}{l}\left(\frac{l}{2}\right)^2 - \left(\frac{l}{4}\right)^2\right) - \frac{4\left(\left(\frac{l}{2}\right)^3 - \left(\frac{l}{4}\right)^3\right)}{3l^2}\right) = \frac{11}{48} \cdot w^2 l \tag{7.7}$$

$$V_2 = w^2 \cdot \left(\left(\frac{2}{l}\left(\frac{3l}{4}\right)^2 - \left(\frac{l}{2}\right)^2\right) - \frac{4\left(\left(\frac{3l}{4}\right)^3 - \left(\frac{l}{2}\right)^3\right)}{3l^2}\right) = \frac{11}{48} \cdot w^2 l \tag{7.8}$$

$$V_3 = w^2 \cdot \left(\left(\frac{2}{l}l^2 - \left(\frac{3l}{4}\right)^2\right) - \frac{4\left(l^3 - \left(\frac{l}{34}\right)^3\right)}{3l^2}\right) = \frac{5}{48} \cdot w^2 l \tag{7.9}$$

As the total iteration time is limited by the longest calculating thread this means we can expect threads calculating the inner Slices to be active for nearly the complete iteration time, while the outer slices are finished after only $\frac{5}{11} \approx 45\%$ of the total time. We can also calculate where the optimal slice boundaries should be, by setting the slice volume to one-quarter of the total volume and solving for b. It suffices to calculate the first boundary. The rest can be determined by symmetry to be at $b_1 = \frac{l}{2}$ and $b_2 = l - b_0$. a can therefore be set to 0. This leads to $b_0 = \left(0.5 - sin\left(\frac{\pi}{18}\right)\right) \cdot l \approx 0.33 \cdot l$ and a volume of $\frac{w^2 l}{6}$, which is about 27% smaller than the largest slice without balancing. This represents an upper bound for the possible speedup gained by balancing. Note that while the dynamic scheduling algorithms described in Section 5.1 do not work on the basis of optimizing the slice thickness, they still have the effect of minimizing the total idle time of all threads, so the speedup for those also has the same upper bound.

7.1.2. Evaluating the Balancing Algorithm

The simulations discussed in this section were run on the CoolMUC2 Cluster using 4 threads. The data layout was fixed to AoS, as this is the only one supported by the Verlet Lists Cells traversals. The simulation was run 50 times for each configuration. Iterations containing a container or neighbor list rebuild were discarded, as they would skew the results. The balanced sliced traversal using the "none" heuristic was used as a baseline to judge the effectiveness of different balancing options. Before we compare the runtime of all the algorithms, we will compare the fraction of time each thread is active for the heuristic-based traversals with the theoretical values calculated in Section 7.1.1. Figure 7.2 shows the time spent in each slice of the balanced sliced traversal using the "none" heuristic as fractions of the total iteration time for each of the three supported containers.

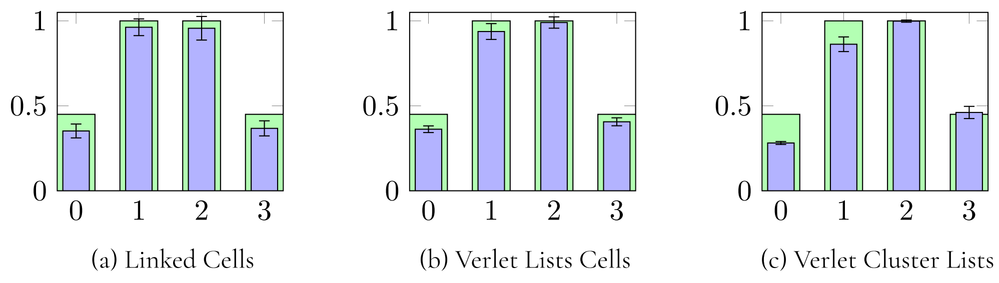


Figure 7.2.: Time spent in different slices as fractions of the total iteration time for all three different containers using the "none" heuristic. The theoretical values calculated in Section 7.1.1 are shown in green. See Table A.2 for the exact values.

The measured time spent in each of the slices is a bit lower than predicted by eqs. (7.6) to (7.9), The exact values are listed in Table A.2. For the outer two slices, this difference can be explained by two different factors: Firstly a layer of halo cells is automatically inserted at both ends of the domain. Each slice gets assigned 18 layers, but for the outer two slices, only 17 of those layers contain particles. Secondly, the outer slices contain a greater proportion of particles, that are at the edge of the solid and thus have fewer neighbors. This was not taken into account when calculating the theoretical load per slice, thus it overestimates the load at the edge. The larger variation in the first slice using the Verlet Cluster Lists container may be a result of uneven slicing. For the first two containers, the combination of the length of the solid and cell side length was chosen to result in a layer count divisible by four. The Verlet Cluster Lists container, however, calculates its own optimal tower side length. In this case, it resulted in the last slice being one layer thicker than the first three. The last two slices, therefore, have a greater combined load than the first two.

When using one of the other two heuristics, the ideal result would be all threads being active for the complete amount of time. However, there will always be some overhead, as calculating the load estimation also requires some time. This can be seen in Figures 7.3 and 7.4, as none of the time fractions reach 100%.

Using the SPpC heuristic the domain was cut into slices of lengths 24, 12, 12, and 24, for both the Linked Cells and Verlet Lists Cells containers. Accounting for the halo cells at both ends the slice boundaries then lie at $0.33 \cdot l$, $0.50 \cdot l$ and $0.67 \cdot l$. This closely matches the ideal slicing points calculated in Section 7.1.1.

The NLL heuristic gives similar results, with slice thicknesses 23, 13, 13 and 23 for the Verlet Lists Cells container, which corresponds to slice boundaries at $0.31 \cdot l$, $0.50 \cdot l$, and $0.69 \cdot l$. This is not as close

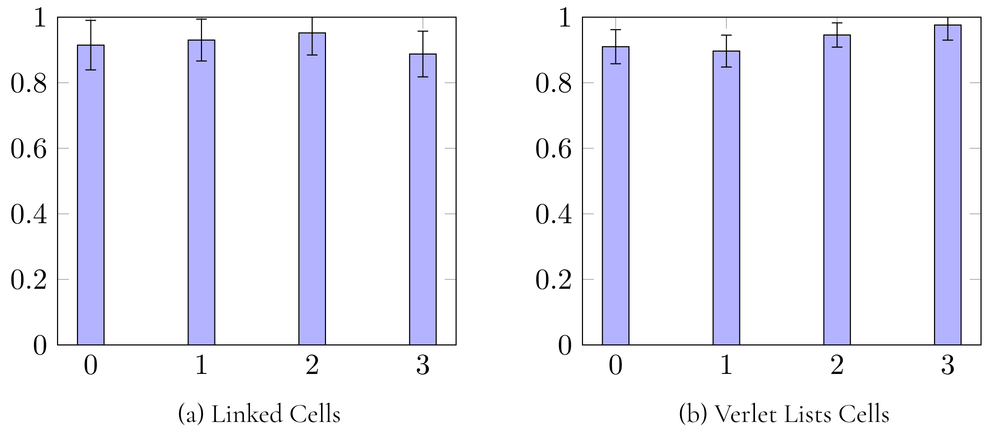


Figure 7.3.: Time spent in different Slices as Fractions of the total iteration time for the Linked Cells and Verlet Lists Cells containers using the SPpC heuristic

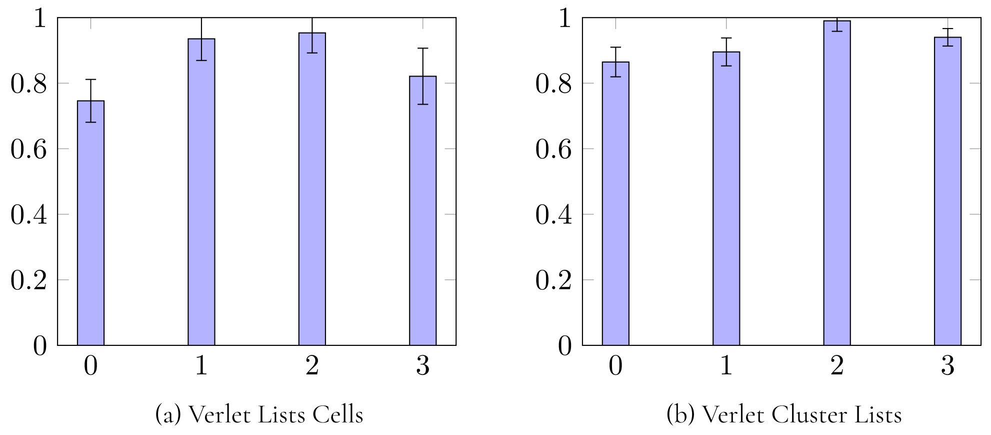


Figure 7.4.: Time spent in different Slices as Fractions of the total iteration time for the Verlet Lists Cells and Verlet Cluster Lists containers using the NLL heuristic

to the theoretically optimal boundary locations as those achieved using the SPpC heuristic. Thus, SPpC uses thread time more efficiently than NLL in this scenario, as can be seen when comparing Figures 7.3 and 7.4. From this, we can expect that SPpC also results in a faster runtime for this container. For the Verlet Cluster Lists container, the slice thicknesses using the NLL heuristic were 20, 9, 10, and 18. This corresponds to slice boundaries at $0.33 \cdot l$, $0.49 \cdot l$, and $0.67 \cdot l$

Having confirmed that using heuristics to estimate and balance the load improves the thread utilization, we can now take a look at the runtime performance gained as a result. Figures 7.5 to 7.7 show the average runtimes for one iteration of the simulation using different containers and balancing approaches, including the two dynamic approaches described in Section 5.1.

Balancing using the SPpC heuristic works well for both supported containers, resulting in average speedups of 29% for the Linked Cells Container and 30% for the Verlet Lists Cells container. This is more than was theoretically predicted to be possible, but can be explained as the imbalance using no heuristic was also underestimated, as mentioned above, resulting in more room for improvement.

The NLL heuristic was slightly worse for the Verlet Lists Cells container, achieving only an average

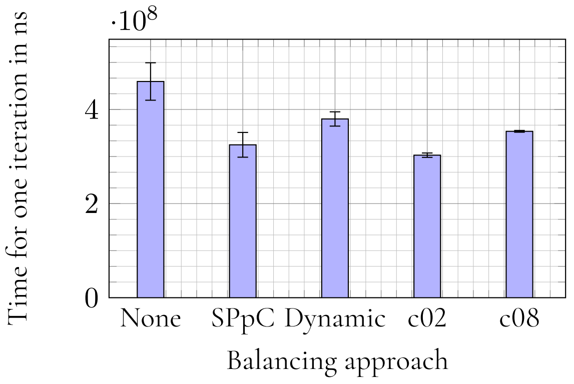


Figure 7.5.: Runtime comparison for the different traversals of the Linked Cells container. The c08 coloring traversal is also included here, to serve as a comparison to the c02 based sliced traversal. The exact values are listed in Table A.1.

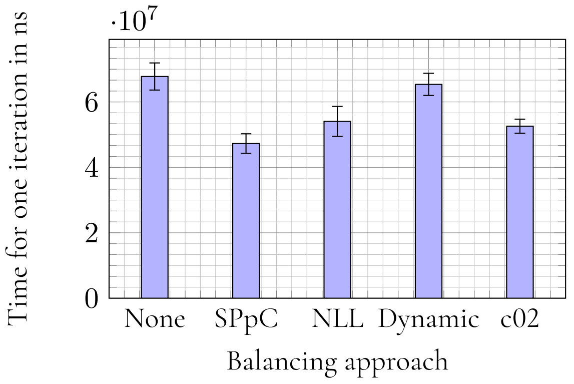


Figure 7.6.: Runtime comparison for the different traversals of the Verlet Lists Cells container.

speedup of 20%. This matches the prediction made from the measurements of thread time utilization. For the Verlet Cluster Lists container, it achieves a speedup of 30%. The larger performance increase for this container compared to the Verlet Lists Cells container can be explained by the fact that the unbalanced sliced traversal for this container already had worse thread time utilization. It should also be noted that due to the way the clusters are built, the NLL heuristic does not work exactly the same for both containers.

The dynamic approach also works well for both the Linked Cells and Verlet Cluster Lists containers, achieving speedups of 17% and 20% respectively. However, it has hardly any effect for the Verlet Lists Cells container. The average speedup there is 3.6%, but may vary from -4.2% to 11%. The dynamic sliced traversal using colors instead of locks, on the other hand, works well for all three containers. The relative speedups for Linked Cells, Verlet Lists Cells and Verlet Cluster Lists are respectively 34%, 22% and 32%. For the Linked Cells and Verlet Cluster Lists containers this makes it the fastest traversal for this specific scenario, however balancing using the SPpC heuristic is still better for the Verlet Lists Cells container. For the Linked Cells container, this traversal also outperforms the c08 traversal by 6.7%. This performance increase is most likely due to fewer OpenMP barriers, as discussed in Section 2.3.2. The average measured times for all configurations are also listed in Table A.1.

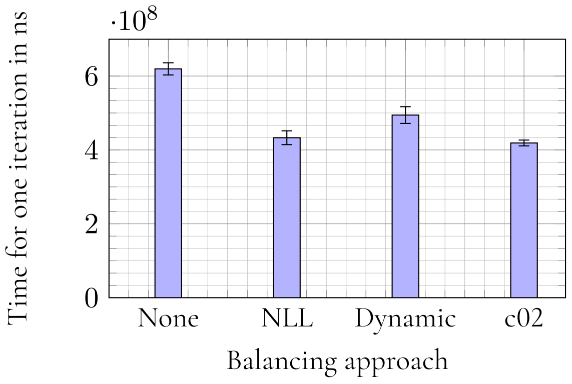


Figure 7.7.: Runtime comparison for the different traversals of the Verlet Cluster Lists container.

7.2. Ostwald Ripening

For the second simulation, the domain was filled with a grid of $640 \times 80 \times 80$ particles. In the first step, 100000 equilibration iterations were run. At this point, the domain contains a homogeneous distribution of particles. The particle density and temperature were chosen to result in a pressure between the triple point and critical point, as well as a temperature above the liquid-gas coexistance line. After the equilibration the temperature of the simulation was dropped to the point where gaseous and liquid phases coexist. This causes the particles to condense into clusters. As they grow, they are pulled together by surface tension and form a minimal surface (in this case a gyroid). This process is known as Ostwald ripening [Ost85]. Figure 7.8 shows the necessary temperature change for this to occur in a phase diagram. Snapshots of the simulation at regular intervals can be seen in Figure 7.9. The result is a continuous increase of inhomogeneity within the simulation domain.

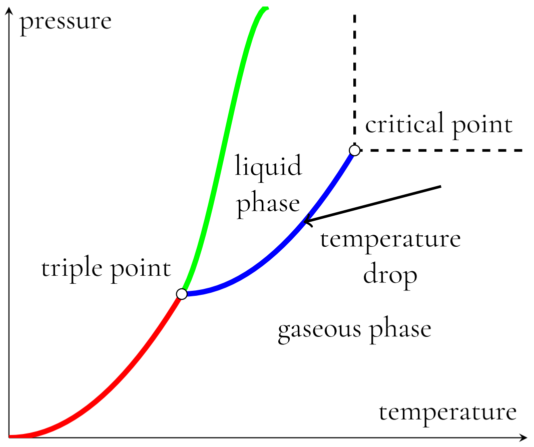
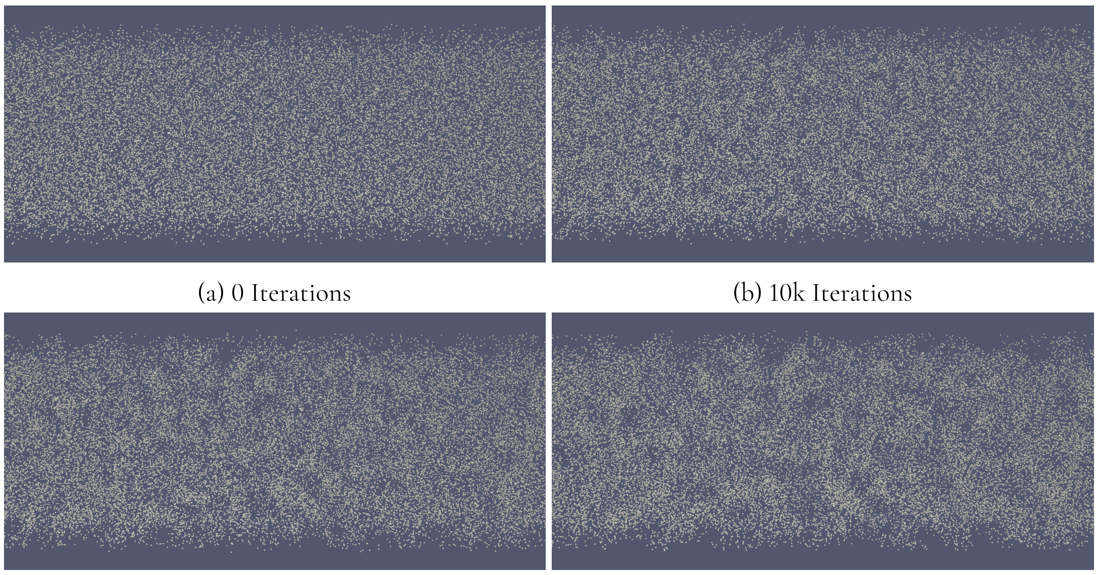


Figure 7.8.: Ostwald Ripening occurs when cooling according to the black arrow in this generic phase diagram.

The second part of the simulation was run for 30000 iterations using the different variations of sliced traversal. The tuning frequency was set to 1000. The complete configuration files used for both steps of the simulation are printed in Listings B.2 and B.3. Using the measured times collected in the tuning iterations, the performance of each traversal over the course of the simulation was analyzed. This



(a) 0 Iterations (b) 10k Iterations

(c) 20k Iterations (d) 30k Iterations

Figure 7.9.: Section of the Ostwald Ripening Simulation after 0, 10k, 20k and 30k Iterations.

simulation could not be run using the Verlet Cluster Lists container, as its slow region iterator makes it unsuitable for such a large scenario.

7.2.1. Per Iteration Run-Time Comparison

Figures 7.10 and 7.11 show the per iteration runtime for the Linked Cells and Verlet Lists Cells traversals respectively. For all of the traversals, this runtime increases as the simulation progresses. This is due to the fact that the average number of neighbors of a particle increases as they move closer to form the clusters, thus causing more forces requiring computation. For the Linked Cells container, the unbalanced sliced traversal, the SPpC balanced version, and the c02-sliced traversal all start out with very similar iteration times. The c02-sliced traversal increases in iteration time faster than the unbalanced traversal, but by 20k iterations both traversals again take comparably long. The sliced traversal balanced using the SPpC heuristic, on the other hand, gains a significant advantage over the unbalanced version. The run-time during the last block of tuning iterations was 3.6% faster. This is to be expected, as the balancing makes little difference for the homogeneous state in the beginning, but compensates for the changing load distribution, as the inhomogeneity increases. The dynamic scheduled sliced traversal is consistently slower than the unbalanced version, taking between 4% and 11% more time per iteration. This is likely due to the fact that the average slice thickness here is 2 layers as opposed to 12 for the heuristic-based traversals (including the unbalanced option). The probability of having to wait for a lock to be released is thus increased. The c02-sliced traversal creates even more slices, but as no locks are used, it is still more performant in this case. Comparing the c02-sliced traversal to the c08 traversal, c02 is 14% faster in early iterations but approaches the run-time of c08 in later iterations. A possible explanation is that in the early iterations, the lower number of colors of the c02-based sliced traversal gives it an advantage, whereas c08 allows for better balancing in an increasingly inhomogeneous domain due to its smaller block size of 8 cells as opposed to an entire slice.

While the traversals for the Verlet Lists Cells container also have increasing iteration times, this

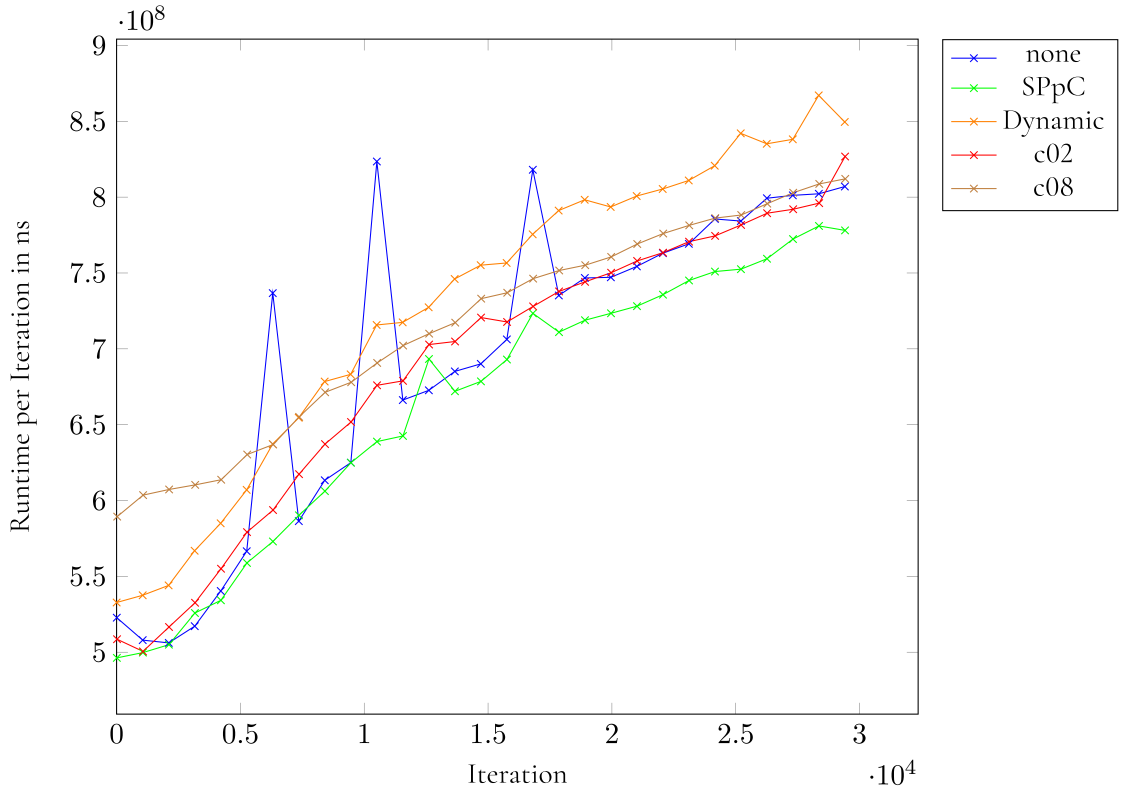


Figure 7.10.: Per iteration runtime of the different Linked Cells Traversals using AoS data layout.

increase is less significant than for the Linked Cells traversals. While the Linked Cells traversals gained between 38% and 63% of the iteration time at the beginning of the simulation, for the Verlet Lists Cells traversals this was only between 9% and 27%. This is likely due to the larger volume that has to be checked for neighbors for the Linked Cells container, as discussed in Section 2.2.2. The unbalanced traversal is clearly slower than all other versions. However, due to the large fluctuations all traversals for this container have, it is difficult to quantify this speedup exactly. Averaged over all of the iterations the c02-based sliced traversal has the largest speedup, taking 21% less time for one iteration. With a speedup of 18%, the dynamic scheduled version has a slight advantage over the two heuristic-based approaches. Using the NLL heuristic the speedup is 15% and using the SPpC heuristic it is 14%.

7.2.2. Evaluating Load Estimation Heuristics

As for the Steinmetz simulation, the imbalance of the loads on the different slices for the heuristic-based balancing approaches was analyzed. In the first simulation, we could simply differentiate between inner and outer slices. In this case, however, 28 threads were used for the simulation. To quantify the imbalance we will instead calculate the relative standard deviation of the time spent on processing each slice. As can be seen in fig. 7.12 both traversals for the Linked Cells container start out with very similar imbalance. However, there is a very slight increase in imbalance over the course of the simulation for the unbalanced sliced traversal, whereas using the SPpC heuristic there is a slight drop. Using ideal balancing the imbalance should stay constant at 0. However, multiple factors prevent us from reaching this ideal. Firstly the heuristics give us only an estimate for the actual load. Secondly, even if we had

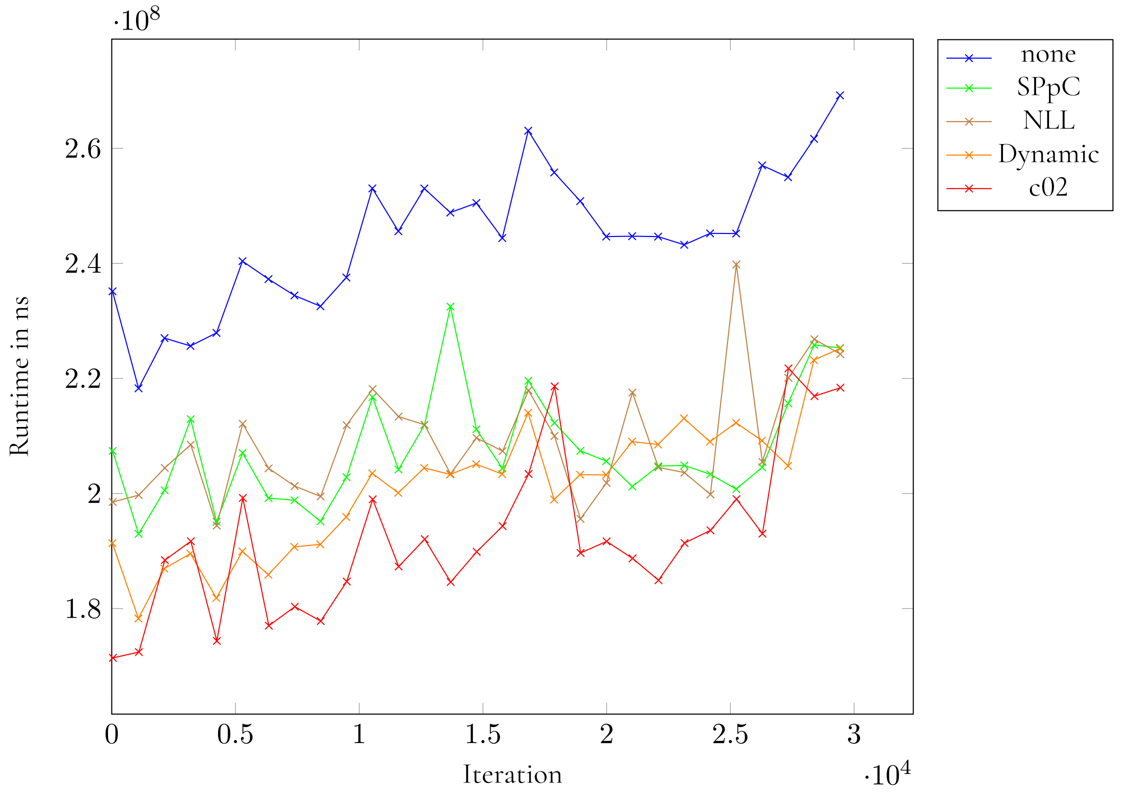


Figure 7.11.: Per iteration runtime of the different Verlet Lists Cells Traversals

accurate load predictions, the greedy balancing algorithm described in Section 5.2 and more specifically by Algorithm 4 does not guarantee optimal slicing. Finally, even with an optimal balancing algorithm we still are only able to slice at discrete intervals. The drop in imbalance for the SPpC traversal, while not specifically expected, can therefore nonetheless be explained. The greater imbalance of the unbalanced traversal by the end of the simulation explains the longer iteration time seen in Figure 7.10. The three spikes seen in the iteration runtime of the unbalanced traversal can also be predicted from this graph, the cause for this momentary imbalance is however unknown.

The same measurement was also performed for the heuristic-based traversals for the Verlet Lists Cells container. As with the runtime measurement, there are very large fluctuations, as can be seen in Figure 7.13. Contrary to the expectation, however, the imbalance decreases slightly for all traversal options, including the unbalanced version. Averaged over all iterations of the simulation, the imbalance using the two different heuristics is however still lower. The unbalanced traversal has an average imbalance of 12.5% whereas using the SPpC heuristic it drops to 11.2% and using the NLL heuristic to 10.6%, which explains why the NLL performs very slightly better for this simulation.

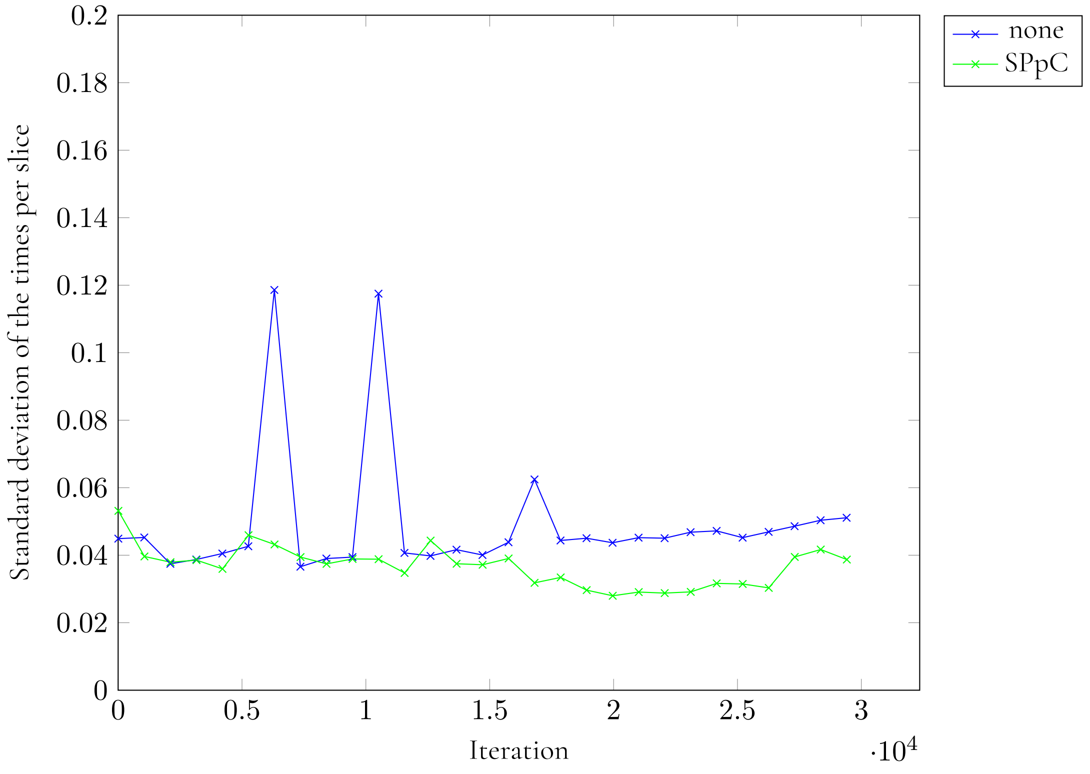


Figure 7.12.: Load imbalance for the different Linked Cells heuristic based Traversals

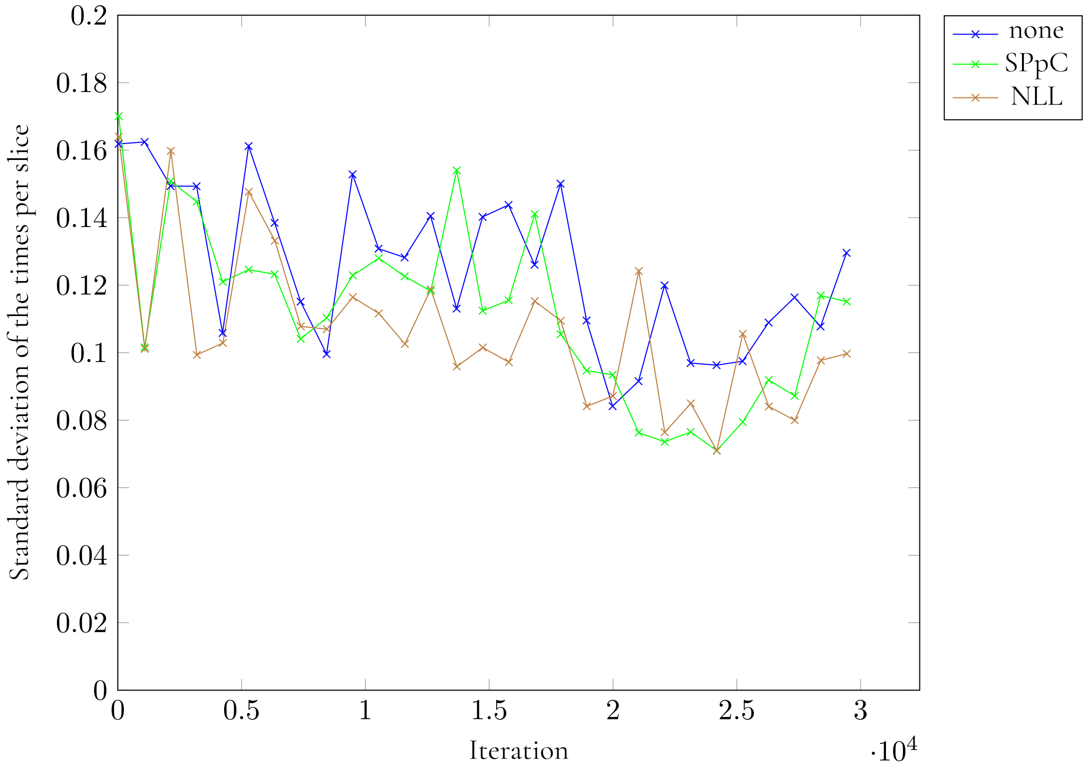


Figure 7.13.: Load imbalance for the different Verlet Lists Cells heuristic-based traversals

Part IV.

Conclusion

8. Summary

In this thesis, multiple new variations of sliced traversal were implemented. To reuse as much code as possible, *SlicedBasedTraversal* was split into four classes. The original class now only contains some common functionality such as finding the longest dimension and calculating slice thicknesses. The latter is done in such a way to create as many slices as possible. *SlicedLockBasedTraversal* provides the original traversal algorithm, using locks on the starting layers of each slice. *SlicedC02BasedTraversal* implements a different traversal algorithm, where odd and even-numbered slices are processed separately in the same fashion as in the various coloring traversals. This eliminates the need for locks on the slice boundaries. *SlicedBalancedBasedTraversal* inherits from *SlicedLockBasedTraversal*, but overrides the calculation of slice thicknesses. A load estimation function inherited from a second superclass is used to create one slice for each thread with approximately equal loads. Each of these three traversals was specialized for the Linked Cells, Verlet Lists Cells, and Verlet Cluster Lists containers. Two load estimation heuristics were implemented based on the number of particles within a cell and the length of neighbor lists. Additionally, a constant heuristic was implemented which provides the behavior of the original unbalanced sliced traversal. The heuristic was added as a tunable option to the different search strategies. Overall this adds 36 new configurations to AutoPas.

The performance of each new traversal was tested and compared against the unbalanced sliced traversal using two different simulations run on the CoolMUC-2 cluster. It was shown that each of the new traversals can outperform the unbalanced sliced traversal of the same container for some scenarios. The time required for one iteration was decreased by up to 34% for very inhomogeneous scenarios. For the Linked Cells container, the c02-sliced traversal was also shown to be faster than the c08 traversal in some situations. The results have also confirmed that using a load estimation heuristic can improve the load balancing across multiple threads.

9. Future Work

There is still a lot of room for improvement in the implementation of the heuristic-based balancing approaches. For one the only heuristic available for the Linked Cells container is SPpC, which makes assumptions that only hold for a cell size factor of 1. It would therefore be useful to create more heuristics that also work for other values. The balancing algorithm used to calculate the slice thicknesses also has its shortcomings, as it does not guarantee optimal balancing. Nor does it take into account how the load is spread within one slice, which can have an effect on the probability of having to wait for a lock to be released. It would be possible to replace the current algorithm or to have several different options that can be selected from either at compile-time or at runtime.

There are also still aspects of the data collected during the Ostwald ripening simulation that remain unexplained. For one, the three spikes seen in the imbalance and iteration runtime of the unbalanced traversal. The first step in diagnosing their cause should be to print out statistics about the domain at each iteration. The already present SPpC heuristic could be used for this purpose. Another option would be writing out vtk files at every iteration, but these would likely contain more information than necessary and take up vastly more disk space. This analysis should elucidate any random inhomogeneity in the domain. The magnitude of the peaks, however, makes this explanation seem unlikely. The best way to proceed in that case would be to measure the time spent in different parts of the iteration in greater detail. Currently, only the total time, the time for each slice, and the time needed for calculating the loads of each layer are measured. Instead of just measuring the total time for each slice, the time spent in each layer, or even each cell could be measured.

The same measurements could also aid in locating the source of the large fluctuations of the iteration runtime for the Verlet Lists Cells container. This in turn is likely necessary to explain the unexpected decrease of load imbalance during the Ostwald ripening simulation for this container.

Part V.

Appendix

A. Measurement Data

Container	Balancing Approach	T_{Total}	Speedup
LC	None	459.4 ms	0%
LC	SPpC	324.9 ms	29.3%
LC	Dynamic	379.8 ms	17.3%
LC	c02	302.8 ms	34.1%
LC	c08	353.5 ms	23.1%
VLC	None	67.8 ms	0%
VLC	SPpC	47.3 ms	30.2%
VLC	NLL	54.1 ms	20.2%
VLC	Dynamic	65.4 ms	3.6%
VLC	c02	52.6 ms	22.4%
VCL	None	619.4 ms	0%
VCL	NLL	433 ms	30.1%
VCL	Dynamic	494.3 ms	20.2%
VCL	c02	418.9 ms	32.4%

Table A.1.: Average iteration time of the Steinmetz simulation for each configuration.

Container	Heuristic	T_0/T_{Total}	T_1/T_{Total}	T_2/T_{Total}	T_3/T_{Total}
LC	None	35.2%	96.2%	95.7%	36.7%
LC	SPpC	91.5%	93%	95.2%	88.8%
VLC	None	36.2%	93.7%	99%	40.6%
VLC	SPpC	91%	89.7%	94.6%	97.6%
VLC	NLL	74.6%	93.5%	95.3%	82.1%
VCL	None	28.1%	86.3%	99.9%	46.1%
VCL	NLL	86.5%	89.5%	99%	94%

Table A.2.: Fractions of time spent in all slices of the Steinmetz simulation for different Configurations

B. Simulation Config Files

B.1. Steinmetz Simulation

```python
#!/usr/bin/python3

import yaml
import math

gridDict = dict()

width = 70
length = 141
stretch = length/width

oldSpacing = 0
x = 0

for i in range(1, length):

    n = math.floor((((length/2) ** 2 - (length/2-i) ** 2)) ** 0.5 / stretch)
    spacing = 1
    x = x + min(spacing, oldSpacing)
    offset = (width - (n-1) * spacing) / 2

    cube = {
        "particles-per-dimension": [1, n, n],
        "particle-spacing": spacing,
        "bottomLeftCorner": [x , offset, offset],
        "velocity": [0,0,0],
        "particle-type": i,
        "particle-epsilon": 1,
        "particle-sigma": 1,
        "particle-mass": 1
        }

    oldSpacing = spacing
    gridDict[i] = cube

conf = {
    "Objects": {
        "CubeGrid": gridDict
        }
    }

print(yaml.dump(conf, default_flow_style=None))
```

Listing B.1: Python script used to create Steinmetz config

B.2. Ostwald Ripening Simulation

```
functor                         :  Lennard-Jones (12-6)
cutoff                          :  2.5
deltaT                          :  0.00182367
iterations                      :  100000
periodic-boundaries             :  true
Objects:
  CubeGrid:
    0:
      particles-per-dimension   :  [640, 80, 80]
      particle-spacing          :  1.5
      bottomLeftCorner          :  [0, 0, 0]
      velocity                  :  [0, 0, 0]
      particle-type             :  0
      particle-epsilon          :  1
      particle-sigma            :  1
      particle-mass             :  1
thermostat:
  initialTemperature            :  1.4
  targetTemperature             :  1.4
  deltaTemperature              :  2
  thermostatInterval            :  10
  addBrownianMotion             :  true
vtk-write-frequency             :  100000
vtk-filename                    :  ostwald_equilibration

verlet-rebuild-frequency        : 10
verlet-skin-radius              : 0.3
container                       : [VerletListsCells]
selector-strategy               : Fastest-Absolute-Value
tuning-strategy                 : active-harmony
newton3                         : [enabled]
```

Listing B.2: Config file for the Ostwald Ripening equilibration step

```
functor                         :  Lennard-Jones (12-6)
cutoff                          :  2.5
# these are the box size values for a equilibration with 640x80x80 and spacing 1.5
box-min                         :  [-0.75, -0.75, -0.75]
box-max                         :  [959.25, 119.25, 119.25]
deltaT                          :  0.00182367
iterations                      :  30000
periodic-boundaries             :  true
thermostat:
  initialTemperature            :  0.7
  targetTemperature             :  0.7
  deltaTemperature              :  2
  thermostatInterval            :  10
  addBrownianMotion             :  false
vtk-filename                    :  vtk/ostwald
vtk-write-frequency             :  30000
checkpoint                      :  ostwald_equilibration_100000.vtk

container                       : [LinkedCells, VerletListsCells]
verlet-rebuild-frequency        : 10
verlet-skin-radius              : 0.3
selector-strategy               : Fastest-Absolute-Value
data-layout                     : [AoS]
traversal                       : [lc_c08, lc_sliced, lc_sliced_balanced, lc_sliced_c02, vlc_sliced,
     vlc_sliced_balanced, vlc_sliced_c02]
tuning-strategy                 : full-search
tuning-interval                 : 1000
tuning-samples                  : 5
tuning-max-evidence             : 10
newton3                         : [enabled]
cell-size                       : [1]
```

Listing B.3: Config file for the Ostwald Ripening simulation

List of Figures

2.1. Lennard-Jones Potential 4
2.2. Visualization of Direct Sum, Linked Cells, and Verlet Lists 5
2.3. c18 and c08 base steps visualized in 2D 6

5.1. Class diagramm 13

7.1. Visualization of the elongated Steinmetz solid 17
7.2. Steinmetz baseline 19
7.3. Steinmetz slice times for SPpC Heuristic 20
7.4. Steinmetz slice times for NLL Heuristic 20
7.5. Linked Cells runtime comparison 21
7.6. Verlet Lists Cells runtime comparison 21
7.7. Verlet Cluster Lists runtime comparison 22
7.8. Ostwald Ripening phase diagramm 22
7.9. Ostwald Ripening 23
7.10. Linked Cells spinodal decompostion runtime 24
7.11. Verlet Lists Cells spinodal decompostion runtime 25
7.12. Linked Cells spinodal decompostion slice imbalance 26
7.13. Verlet Lists Cells Ostwald ripening slice imbalance 27

List of Tables

A.1. Average iteration time of the Steinmetz simulation 32
A.2. Fractions of time spent in all slices of the Steinmetz simulation 32

Bibliography

[Abr+15] Mark James Abraham et al. “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers”. In: *SoftwareX* 1-2 (2015), pp. 19–25.

[Ber98] Daniel Berthelot. “Sur le mélange des gaz”. In: *Comptes rendus hebdomadaires des séances de l’Académie des sciences* 126 (1898), pp. 1703–1706.

[Gra+19] Fabio Gratl et al. “AutoPas: Auto-Tuning for Particle Simulations”. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), pp. 748–757.

[KSP20] Yogesh Kumar, Harvijay Singh, and Chirag N. Patel. “In silico prediction of potential inhibitors for the Main protease of SARS-CoV-2 using molecular docking and dynamics simulation based drug-repurposing”. In: *Journal of infection and public health* 13 (9 2020), pp. 1210–1223.

[Len24] John Edward Lennard-Jones. “On the Determination of Molecular Fields. II. From the Equation of State of a Gas”. In: *Proceedings of the Royal Society of London* 106.738 (1924), pp. 463–477.

[Lor81] Hendrik Antoon Lorentz. “Ueber die Anwendung des Satzes vom Virial in der kinetischen Theorie der Gase”. In: *Annalen der Physik* 248.1 (1881).

[Mia+15] Yinglong Miao et al. “Accelerated molecular dynamics simulations of protein folding”. In: *Journal of Computational Chemistry* 36.20 (2015), pp. 1536–1549.

[New87] Isaac Newton. *Philosophiae Naturalis Principia Mathematica*. Vol. 1. 1687.

[Ost85] Wilhelm Ostwald. *Lehrbuch der Allgemeinen Chemie*. Vol. 1. 1885.

[Ver67] Loup Verlet. “Computer “experiments” on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules”. In: *Physical Review* 159.4 (1967), pp. 98–103.