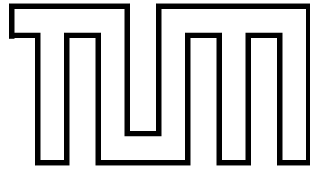


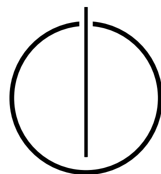
FAKULTÄT FÜR INFORMATIK

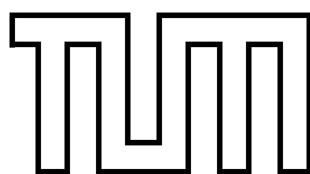
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Enabling Massive Parallelism for the AutoPas Demonstrator MD-Flexible using Adaptive Domain Decomposition

Jacky Körner





FAKULTÄT FÜR INFORMATIK

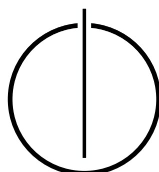
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Enabling Massive Parallelism for the AutoPas Demonstrator MD-Flexible using Adaptive Domain Decomposition

Realisierung von Massivem Parallelismus für den AutoPas Demonstrator MD-Flexibe mit Hilfe von Adaptiver Domänen Unterteilung

Author:	Jacky Körner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Alexander Gratl, M.Sc.
Date:	15.10.2021



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

Munich, 15.10.2021 Jacky Körner

Acknowledgments

I want to thank the Scientific Computing Chair at TUM for enabling me to do the Master's Thesis in this topic and Fabio Gratl for being a relaxed and helpful supervisor.

Additionally, I want to say sorry to all my friends and family who had to relinquish my presence at multiple events and who complained about me being a workaholic, only because I tried to put as much time as expected from a student into this project. Thank you for your patience!

Abstract

Real world scenarios in molecular dynamics simulations are highly complex and cannot be efficiently approximated by single core applications. To achieve convincing results in a reasonable amount of time, scientists use supercomputers with a large number of compute units.

So far, the demonstrator MD-Flexible for the AutoPas particle simulation library was limited to a single process. MD-Flexible now has been turned into a massively parallel application using MPI for inter-processor communication and adaptive domain decomposition for workload balancing. It subdivides the simulation domain into a regular grid which is either balanced by ALL's Tensor method or by the here presented Inverted Pressure method. After taking a brief introduction into Molecular Dynamics and Adaptive Domain Decomposition, we take a look at the technologies involved in the parallelization of MD-Flexible and compare it to other well known massively parallel Molecular Dynamics applications. Before going into detail about the implementation of massive parallelism, we explain on a high level how MD-Flexible worked before and what has changed during the implementation. Next, we describe the details about the adaptive domain decomposition, the resulting inter-process communication and other requirements that arise from the parallelization. The result has been evaluated by running three scenarios with up to 128 processes. The best speedup we achieved using 128 processes is 40. The performance of the parallelization is investigated during the evaluation and several suggestions are given on how it can be improved.

Contents

Acknowledgments vii

Abstract ix

I. Introduction and Background 1

1. Introduction 2
1.1. General Applications for Massive Parallelism 2
1.2. Massive Parallelism for AutoPas' MD-Flexible 2

2. Theoretical Background 4
2.1. Molecular Dynamics 4
2.1.1. The N-Body Problem 4
2.1.2. Reducing the Complexity of MD Simulations 4
2.2. Adaptive Domain Decomposition 6
2.2.1. Different Domain Decompositions 6
2.2.2. Handling Communication and Inter-Domain Forces 8

3. Technical Background 10
3.1. AutoPas 10
3.2. MD-Flexible 11
3.3. ALL 14
3.4. Technologies 14
3.4.1. MPI 14
3.4.2. Visualization Toolkit 15

4. Related Work 16
4.1. GROMACS 16
4.2. LAMMPS 16
4.3. ls1-mardyn 17

II. Refactoring MD-Flexible 18

5. High Level Application Architecture 19
5.1. Old Application Architecture 19
5.1.1. Overview 19
5.1.2. Initialization of the Simulation Component 20

5.1.3. The old Simulation Loop 21
5.2. New Application Architecture 22
5.2.1. Overview 22
5.2.2. Initialization of the new Simulation Component 23
5.2.3. The new Simulation Loop 24
5.2.4. Minor Application Features 24

6. Implementation **26**
6.1. Adaptive Domain Decomposition 26
6.1.1. Setting up the Regular Grid Decomposition 26
6.1.2. Diffuse Load Balancing of Regular Grid using the ALL load balancing library 31
6.1.3. Diffuse Load Balancing using the Inverted Pressure Method 32
6.2. Point-to-Point Communication for Sending and Receiving Particles 35
6.2.1. Serialization and Deserialization of Particles 36
6.2.2. Sending and Receiving Point-to-Point Messages 37
6.2.3. Step-wise communication with neighbor domains 38
6.3. Serial Simulation 42
6.4. Meaningful Metadata 43
6.5. Visualizing the Parallel Simulation 44
6.5.1. Parallel Vtk Writer 44
6.5.2. Creation of particle records 45
6.5.3. Visualization of domains 45
6.5.4. Additional information about domains and particles 48

III. Evaluation **51**

7. Evaluation **52**
7.1. Speedup And Efficiency 52
7.1.1. Setup 52
7.1.2. Results 53
7.2. Detailed Discussion of the Scenarios Performances 54
7.2.1. Performance of Spinodal Decomposition Scenario 55
7.2.2. Performance of Falling Drop Scenario 55
7.2.3. Performance of Exploding Liquid Scenario 56
7.3. Comparing ALL's Tensor Method the Inverted Pressure Method 58

IV. Future Work **60**

8. General Performance Improvements **61**
8.1. Configurable Load Balancing Interval 61
8.2. Soften the Minimum Cell Size Restriction 61

9. Improving the Adaptive Load Balancing 62
9.1. Improving Inverted Pressure . . . 62
9.1.1. Global Balancing along a coordinate axis . . . 62
9.1.2. Improving Stability . . . 63
9.2. Increasing parallelism . . . 63

10. Other Improvements and possible Features 65
10.1. Thermostat and Homogeneity . . . 65
10.2. Extended Subdivision Constraint . . . 65
10.3. Proper Testing . . . 65

V. Summary 66

VI. Appendix 69

Bibliography 73

Part I.

Introduction and Background

1. Introduction

Molecular Dynamics Simulations are very complex by nature, not only due to the rules defined by a specific scenario but also due to their increasing scale. For several decades now, MD simulations are computed on massively parallel systems such as the supercomputers at the Leibniz Rechenzentrum[1]. But without properly designing and implementing scalable MD applications, researchers will not gain any advantage from a supercomputer's computational power. To harness this power applications requires additional development time and technical knowledge. In return for this investment, previously unfeasible tasks can be realized in a reasonable amount of time. In other words, massive parallelism can heavily reduce the time-to-solution.

1.1. General Applications for Massive Parallelism

The usefulness of massively parallel applications extends far beyond Molecular Dynamics simulations. Predicting weather and climate[8] or natural disasters like earthquakes[12] are computationally very challenging tasks and greatly benefit from supercomputers to swiftly create accurate results.

In general, every computational task requiring large amounts of data benefits from a parallel implementation. This is also the case in medicine where it is key to quickly visualize data, for example, created by computer tomography. If doctors have faster access to the visualized data, they are able to develop a diagnosis and treatment much earlier[13]. Pharmaceutic companies require the visualization of data created by electron microscopes to analyze errors in concoctions or to reproduce products of other companies.

Massive parallelism is also heavily used to speed up the training of neural networks. Researchers at the TU Kaiserslautern managed to achieve a speedup of six when training a residual neural network on 256 cores by using a layer-parallel approach[5].

1.2. Massive Parallelism for AutoPas' MD-Flexible

With AutoPas being one of the few auto-tuning libraries, demonstrating its effectiveness when using many nodes is essential. Many MD scenarios are very non-homogeneous and can greatly benefit from node level tuning.

Although AutoPas is a node-level auto-tuning library, its demonstrator application MD-Flexible does not need to be limited to a single node. In fact, because of the complexity of MD simulations, users are much more interested in the performance of AutoPas on large scale machines when used in a (MPI) parallelized software.

Having a demonstrator with multiple well known scenarios running on massively parallel systems benefits the adaption rate for the AutoPas library. Possible users now are able to

[1] `https://www.lrz.de/english/`

quickly compare performance and implementation to other MD libraries, use MD-Flexible as a basis for their own simulations, and decide whether AutoPas is suited to their purpose.

Until now, MD-Flexible has not been able to demonstrate the inter-node tuning feature implemented in AutoPas. Including massive parallelism in MD-Flexible turns it into a feature-complete demonstrator.

2. Theoretical Background

Before we talk about how massive parallelism has been integrated into MD-Flexible, it is important to understand the purpose of Molecule Dynamics simulations and why they are extremely complex. Based on this knowledge, we can highlight techniques from Adaptive Domain Decomposition and other methods to reduce the time-to-solution.

2.1. Molecular Dynamics

The purpose of Molecular Dynamics is "understanding the properties of assemblies of molecules in terms of their structure and the microscopic interactions between them"[2]. These interactions occur on a microscopic range and time scale and, therefore, cannot be observed properly using traditional methods. In addition, not every environment is suited for practical experiments. For instance, how would researchers be able to observe the nuclear fusion occurring in the inside of the sun? Molecular Dynamics simulations allow researchers to visualize arbitrary scenarios which are only limited by compute resources. From those visualizations they are able to gain previously unobtainable insights.

2.1.1. The N-Body Problem

Molecular Dynamics Simulations are categorized as N-Body Simulation. In simple words, N-Body Simulations consist of particles interacting with each other. Depending on the range of the interactions, the simulations can turn into a problem of $O(N^2)$ complexity, because each of the N particles interacts with every other particle, with N corresponding to the number of particles in the simulation. Reducing this complexity is one of the main challenges in MD and is referred to as the N-Body Problem. Interactions can be categorized into long-range and short-range interactions. Both have dedicated solutions to this problem.

2.1.2. Reducing the Complexity of MD Simulations

Short-range interactions are most prevalent in Molecular Dynamics. Their complexity can be reduced by minimizing the number of particles which need to be checked whether they interact with a particular particle or not. The distance check used to determine if an interaction occurs between particles uses a *cutoff radius* r_c: If a particle lies within this radius around another particle, these two particles will interact with each other. Along with the naive **Direct Sum** approach, there are two other common techniques to reduce the number of distance checks called **Linked Cells** and **Verlet Lists**. Understanding these approaches is important when introducing massive parallelism as they influence which particles need to be communicated between processors.

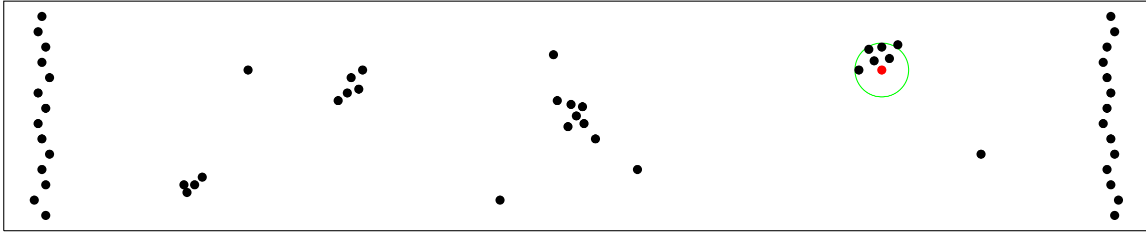


Figure 2.1.: Depicts a constructed exploding liquid scenario. Looking at the red particle, with the green circle indicating the cutoff distance, the distance check needs to be applied to every other particle within the domain, when using the **Direct Sum** approach. 55 distance checks need to be performed for the red particle.

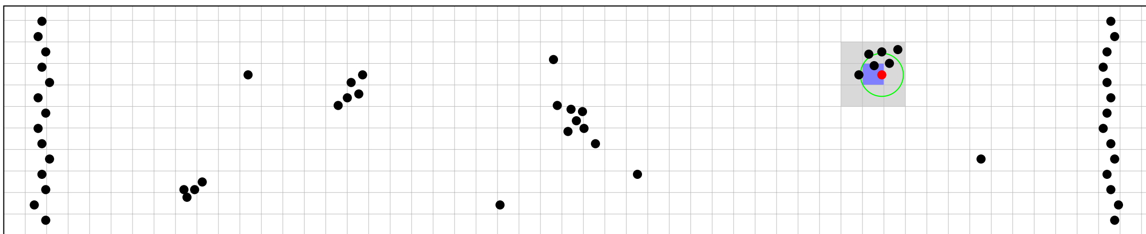


Figure 2.2.: Using the Linked Cells approach, the number of particles which need to be checked for the red particle has been reduced to 6 because only the particles within the blue cell and the particles within the neighboring cells need to be checked. Two of the 6 particles are actually within the cutoff radius.

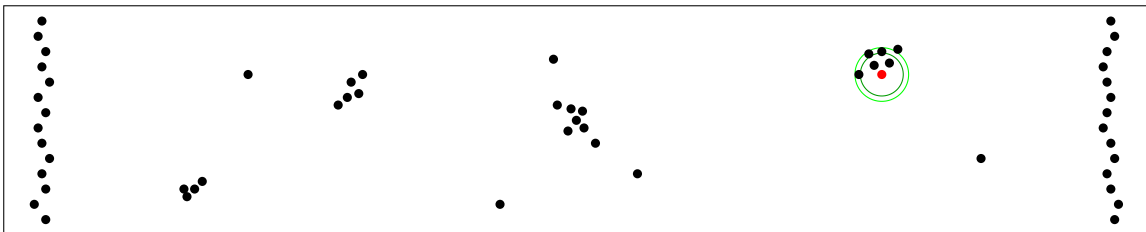


Figure 2.3.: The Verlet Lists approach reduces the number of distance checks for the red particle to 5. Additionally, the three particles which are close to being within the cutoff radius are likely to migrate into the radius in upcoming iterations. The light green circle represents $r_c + r_{skin}$ while the dark green circle represents r_c.

The simple **Direct Sum** approach (see Figure 2.1) is only using the cutoff radius as a criterion to determine required interactions. Here, the distance check needs to be performed for every particle which results in a complexity of $O(N^2)$.

The **Linked Cells** works by cutting the simulation domain into cells with a side length which is larger or equal to the *cutoff radius*. Now only the particles in the neighboring cells need to be tested if they lie within the interaction radius (see Figure 2.2). This reduces the complexity to $O(N)$ if the number of cells is proportional to the number of particles.

Nonetheless, some particles will be tested which are, in the worst case, two times the diagonal of a cell apart, which is more than two times the cutoff radius.

To overcome this issue, the **Verlet Lists** approach creates a list of neighbor particles for every particle, where each neighbor is likely to interact with this respective particle (see Figure 2.3). This can be achieved by increasing the *cutoff radius* by a *skin width* r_{skin}. Using this skin includes particles into the lists which are likely to become interacting neighbors in the near future. This also allows the simulation to reuse the same Verlet Lists over multiple iterations. Reconstructing these lists every time step would degrade the approach to the Direct Sum. When constructing the Verlet Lists with the help of the Linked Cells approach, the complexity of the distance checks can be reduced to $O(N)$. In addition, the worst case distance of a possibly interacting particle is reduced to $r_c + r_{skin}$ instead of two times the diagonal of a cell as in the Linked Cells approach. To determine which particles will be part of the Verlet List for a specific particle, only the particles in the same cell or in adjacent cells need to be checked.

2.2. Adaptive Domain Decomposition

Not only is it essential to minimize the number of distance checks to reduce the runtime of a simulation, but also to parallelize the simulation with help of Adaptive Domain Decomposition.

This concept refers to the process of subdividing the simulation domain into subdomains while maintaining a balance between those domains over the course of the simulation. The resulting subdomains are distributed among a set of processes which are responsible for simulating their assigned region. Balance is maintained by ensuring that each process performs a similar amount of work. This can be achieved by resizing the subdomains according to the topology of the domain or the work performed by the processes. A developer implementing Adaptive Domain Decomposition faces three major challenges: Actually creating a proper decomposition, handling the communication between processes, and balancing the work performed by the processes.

2.2.1. Different Domain Decompositions

A domain can be subdivided in many ways, from a simple grid decomposition, to a heterogeneous mesh decomposition constructed out of different polygons. To decide which type of decomposition is most suitable depends not only on the scenario or the time available to implement a solution, but also on the hardware topology of the system on which the scenario is simulated. A system's layout influences communication time and a proper domain subdivision can reduce congestion in it's communication network. The hardware aspect will not be considered during the integration of massive parallelism into MD-Flexible, because the main goal is to focus on Adaptive Domain Decomposition.

Two well known tools for domain decomposition are *Grids* and *Octrees*. While MD-Flexible only provides a regular grid decomposition, the advanced octree decomposition is well worth mentioning in this context.

Regular Grid Decomposition subdivides the domain into a grid of subdomains, where the number of subdomains is equal to the number of processes. Not only is it easy to

implement, but the *domain neighbor lists*, which are required later for communication, only have to be computed a single time. Additionally, the grid does not need to be reconstructed during load balancing. The developer just needs to shift the respective division planes between the subdomains. Unfortunately, this comes with a major drawback: The subdomains cannot be resized arbitrarily because they are restricted to the same neighbors / position in the subdivision grid. When increasing the size of a single subdomain, every other subdomain in the respective column needs to be enlarged, too, even if they are well balanced.

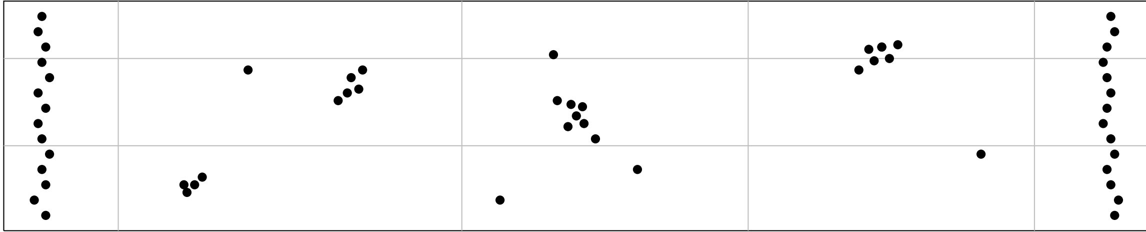


Figure 2.4.: A grid decomposition of the Exploding Liquid scenario.

In contrast to the Grid Decomposition a tree-based subdivision like the **Octree Decomposition** generates a locally refined mesh which is generated by subdividing each domain recursively into eight equal sized subdomains. The recursion stops if a subdomain contains less than a predefined number of particles or has reached a minimum domain size. Because the number of subdomains may be larger than the number of available processes, they need to be explicitly assigned to a process. This can be very challenging, because not only is it required to balance the workload of the processes but also to minimize the communication between processes. Additionally, parts of the decomposition need to be recomputed during load balancing. Consequently, the domain neighbor lists need to be recomputed. Therefore, the computational costs of maintaining a proper Octree Decomposition are potentially much higher compared to the Grid Decomposition.

Alternatively to stopping the recursion based on the particle count, it can be stopped as soon as the subdomain count is equal to the number of processes available. Using this approach, each subdomain can be assigned to a single processor. Unfortunately, this does not result in a well balanced decomposition, because some subdomains of the octree might have very few particles or might even be empty (see quadtree example in Figure 2.5). On top of that, the user is restricted to use a number of processes which is a multiple of eight.

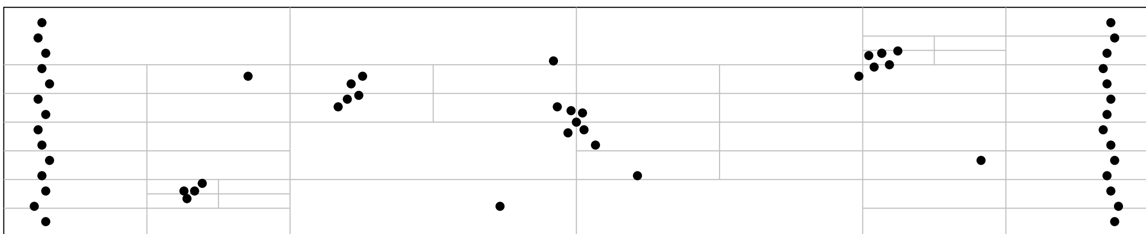


Figure 2.5.: A quadtree decomposition with four levels on the exploding liquid scenario. A quadtree is the equivalent to an octree but in 2D instead of in 3D. It is clearly visible that some of the subdomains do not contain any particle which leaves processes inactive if every process is assigned a single subdomain.

After creating the domain subdivision, each process only knows about its own subdomain(s). Therefore, to retain a continuous and realistic simulation processes are required to communicate with each other.

2.2.2. Handling Communication and Inter-Domain Forces

The majority of this communication comes from particles which may migrate from one subdomain to another, or from particles of an adjacent domain required to compute all forces on particles within the domain. In the second case, the communicated particles are considered *halo particles*. These particles are not actually owned by the receiving process, but are nonetheless stored on the process's local memory. Which particles are considered halo particles is determined by a *halo region* surrounding every subdomain. The designer of the simulation needs to define the width of the halo region which can be oriented around the cutoff radius and the skin width used in Verlet Lists method. To be able to communicate the particles, a process needs to know which other processes own the domains adjacent to his own. Maintaining those *domain neighbor lists* can be, depending on the subdivision, very expensive because it requires synchronization and additional communication between the processes.

Using halo particles comes with several drawbacks. First of all, every halo particle is an additional particle which needs to be simulated within a subdomain increasing the total workload on the processors. Second, this concept does not take advantage of Newton's third law $F_{ij} = -F_{ji}$ where F_{ij} is the force exerted from particle j on particle i. For each halo particle, we do a force calculation which will also be calculated on the process owning the original particle. On top of that, storing the halo particles consumes additional local memory and keeping them up to date introduces additional communication every time step.

An alternative to halo particles is the *eighth shell method*[7]. Using this method, a process receives all particles within cutoff radius around it's domain from seven neighbor processes in total, assuming that the simulation domain has been divided into a regular grid. The receiving process then calculates the force updates for owned particles and all particles it received. The results for each particle are then send to their original processes.

An equivalent method in 2D could be considered a fourth shell method. According to the the authors of the paper about Algorithms for Highly Efficient Load-Balanced, and Scalable Molecular Simulation[7], the method can be described in the following way: Assuming a two dimensional domain is decomposed into a regular grid, the neighbors of a subdomain can be categorized into quadrants. The neighbor domain above the reference domain is tagged as N for "north of domain" while the domain at the top left of the reference domain is tagged as NE for "north-east of domain". Following this approach, each neighbor domain receives a unique tag from the following tag list: N, NE, E, SE, S, SW, W, and NW. An example for this setup is illustrated in Figure 2.6. Each subdomain then receives the relevant particles from the quadrants locally tagged as N, NE and E, calculates the force updates for all particles, and sends the results back to the quadrants. Simultaneously the reference domain sends the particles required from the quadrants tagged as S, SW and W. The processes responsible for those quadrants calculate the force updates and send the results back to the reference domain. Now the domain can correctly update it's owned particles. A visualization in 2D and in 3D can be seen in Figure 2.6.

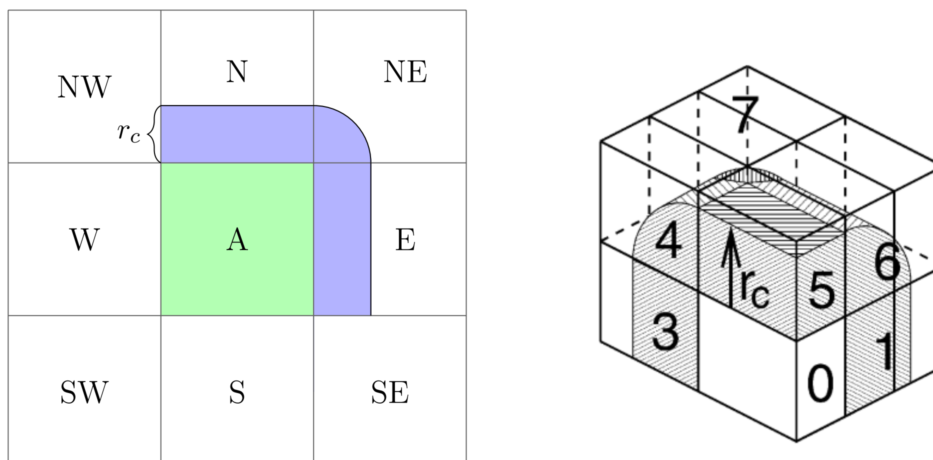


Figure 2.6.: The left image displays the fourth shell scheme. The particles in the blue region owned by cells *E*, *NE* and *N* are send to A who calculates all force interactions. The right image displays the eighth shell method where the particle information in the regions marked from 1 to 7 are communicated to the corner cell marked with a 0 [7]. r_c refers to the cutoff radius.

The eighth shell method reduces the amount of local memory required to store particles of neighbor domains and allows the simulation to calculate each force update only a single time. It therefore takes advantage of Newton's third law, thereby reducing the computational work required for the force calculations. With this method the processes need to communicate twice every time step which may strain the network.

The decision which approach to use should depend on the simulation and the hardware. A simulation which is compute bound and running on a supercomputer with a fast network would definitely profit from the eighth shell method. On the other hand, a communication bound simulation should consider using the halo particle approach.

Although the eighth shell method is in theory more efficient, MD-Flexible implements halo particles. AutoPas containers do not calculate forces between particles which they do not own. Until this is changed, AutoPas does not support the eighth shell method.

3. Technical Background

But what actually is the purpose of AutoPas? Because a large amount of MD-Flexible's features are directly linked to AutoPas we will have a look at the library and MD-Flexible itself. Additionally, we will learn details about ALL and the most important technologies used for the parallelization of MD-Flexible.

3.1. AutoPas

AutoPas is a C++ library allowing users to automatically tune the node-level performance after integrating it into their particle simulations. It has been developed by the Chair for Scientific Computing in Computer Science at the Technical University of Munich.

AutoPas acts as a data container which handles particle updates, node-level optimizations and shared memory parallelization. For the users, the AutoPas container is a black box which only allows them to initialize particle properties, access particles, and define short-range force interactions between particles. In addition to the black-box container, AutoPas provides a functor and a particle class for the Lennard-Jones potential commonly used in molecular dynamics.

To minimize the number of distance checks when determining required pairwise interactions between particles, AutoPas chooses between Linked Cells, Verlet Lists, the combination of both, or other algorithms based on the two methods. Which optimization approach is chosen is determined during tuning phases where the performance of selected methods will be tracked for several iterations. The method with the best performance will then be used for the simulation until the next tuning phase is triggered. Users can decide to share the tuning results with other AutoPas containers running on different nodes if they use MPI parallelization within their simulation. The containers then commit their tuning results into a common knowledge base which can be used by other containers to decide which optimization strategy they want to employ.

AutoPas is not required to run on multiple nodes and therefore the inter-node tuning feature needs to be enabled manually. The library can be used in a massively parallel simulation nonetheless which is demonstrated by the example application MD-Flexible. The demonstrator uses the interface functions `addParticle()`, `addOrUpdateHaloParticle()`, `updateContainer()`, `resizeBox()`, `getNumberOfParticles()`, `iteratePairwise()`, and `getCutoff()` which are all provided by the AutoPas container. Two iterators, also provided by the AutoPas container, allow MD-Flexible to access specific particles within the container. The `addParticle()` and `addOrUpdateHaloParticle()` functions can be used to include additional particles into the container, the `updateContainer()` function updates it's the internal data, returns particles which left the container's domain boundaries and indicates if it actually has been updated. The domain boundaries can be changed using the `resizeBox()` function. The `getNumberOfParticles()` function is used to retrieve how many particles are currently simulated by the container. `iteratePairwise()` allows users to pass a functor

which will then be used to calculate pairwise interactions between particles. Using the `getCutoff()`, the cutoff distance configured for the container can be retrieved.

In addition to the already mentioned interface members, MD-Flexible uses several setters and other functions to configure the AutoPas container during initialization. All interface functions are documented either in the source code or in the official AutoPas documentation on github[1].

3.2. MD-Flexible

The MD-Flexible application is intended to be a showcase for AutoPas. It serves not only as a means to evaluate AutoPas' potential but also helps developers to integrate it into their own applications, themselves.

After downloading and building the application, the users can run several basic scenarios. These scenarios are generated using particle clusters of a predefined shape like a sphere or several types of blocks. The blocks differentiate themselves by the way the particles are positioned inside the block. The particles within the blocks can either be very close packed, positioned according to a Gauss or Uniform distribution or positioned on a grid. Alternatively to using these objects to initialize the particles, the users are also able to load a previously recorded simulation state. The particle objects and other configuration values, including all AutoPas parameters, can be defined for each scenario individually either by using command line arguments or a configuration file.

The **Falling Drop** scenario uses three different objects to simulate a drop falling into a body of the same liquid as can be seen in Figure 3.1. The sphere object is used to create the drop and the body of liquid is created using the grid where the particles are packed closest. Because the scenario employs a global force to simulate gravity, particles need to be prevented from falling "below ground". For this reason, the setup incorporates a grid block of particles with an infinite mass. The high mass prevents the particles from moving which leaves the grid block acting like a floor. In general, the scenario uses periodic boundaries so particles can wander from one side of the simulation domain to the other.

The **Spinodal Decomposition Equilibration** uses the grid block object to fill the domain with particles. The simulation then runs for one million iterations leaving the particles in an equilibrium state. The record of this state is required by the **Spinodal Decomposition** scenario. Starting from this record, the scenario then drops the temperature forcing the particles to build clusters. Figure 3.2 shows the scenario after 30000 iterations.

The last scenario for which MD-Flexible provides a configuration file is the **Exploding Liquid** scenario visible in Figure 3.4. It also uses the block with the closest packed particles as initial state to simulate a rapidly expanding liquid.

[1] `https://github.com/AutoPas/AutoPas`

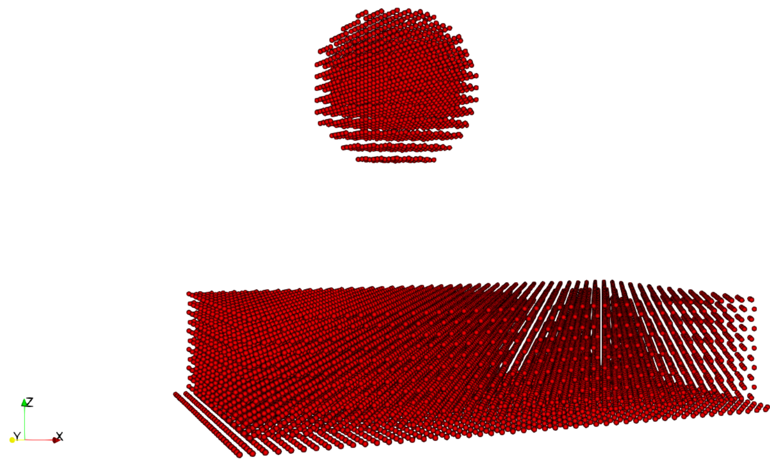


Figure 3.1.: The initial state of the Falling Drop scenario. Here, it is easy to discern which particles have been initialized using the sphere object.

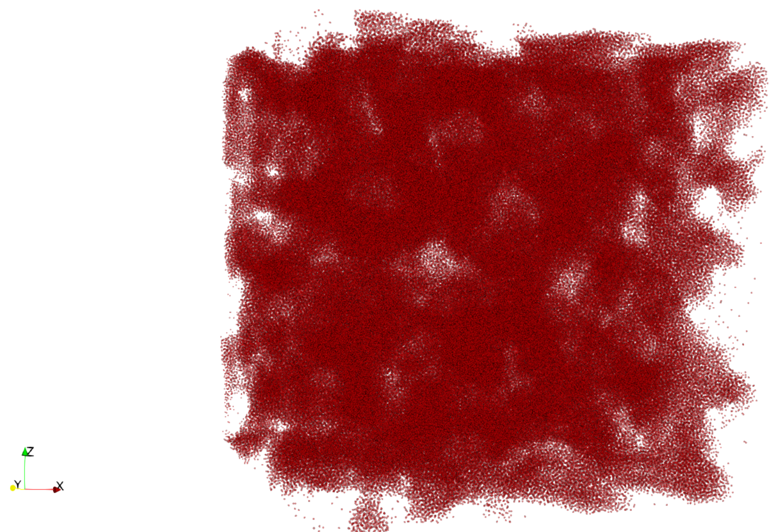


Figure 3.2.: The final state of the Spinodal Decomposition scenario with 30000 iterations.

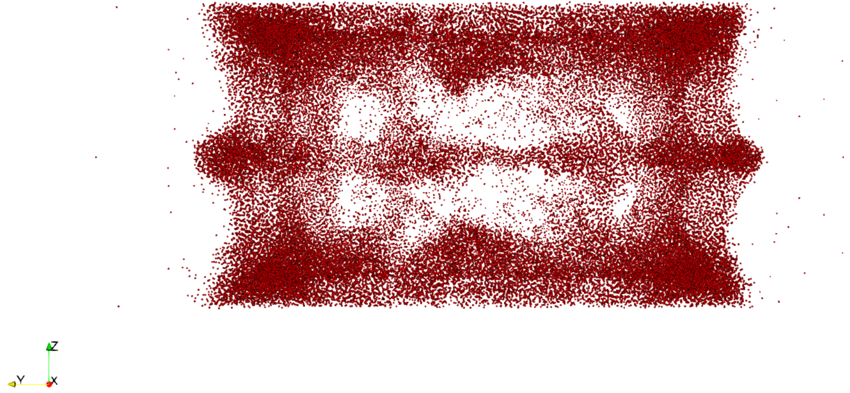


Figure 3.3.: The Exploding Liquid scenario at 250 iterations in the simulation. This image has been created using the test configuration described in Subsection 7.1.1.

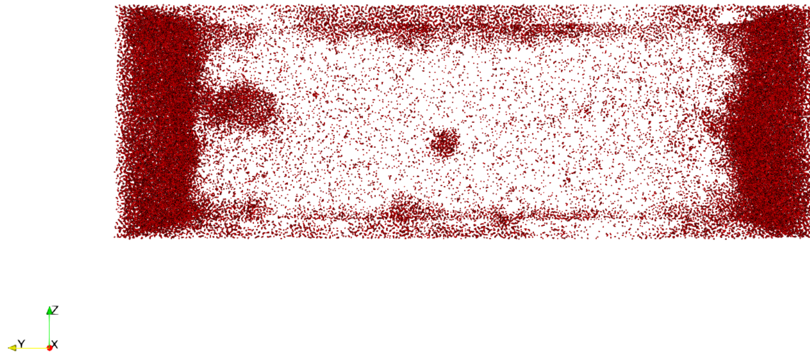


Figure 3.4.: The Exploding Liquid scenario after 50000 iterations in the simulation. This image has been created using the test configuration described in Subsection 7.1.1.

All of these scenarios can be run using a single or multiple processes. Based on number of processes, MD-Flexible will automatically generate a grid decomposition. The users can also configure along which coordinate axes the application is allowed to subdivide the simulation domain. For instance, because the Falling Drop scenario employs a global downwards force,

it might be reasonable to limit the subdivision to the x and y axis because gravity will then implicitly balance the load along the z axis. How this restriction is realized will be described in Subsection 6.1.1.

3.3. ALL

The new MD-Flexible application implements two approaches to the adaptive load balancing of the grid decomposition. A custom algorithm based on pressure (see Subsection 6.1.3) and the Tensor Method provided by ALL[2] which is developed by the Simulation Laboratory Molecular Systems of the Juelich Supercomputing Centre at the Research Centre Juelich (Forschungszentrum Jülich GmbH) in Germany. The library is free of charge and provides three load balancing schemes commonly used with simulations employing domain decomposition: The Tensor Method, the Staggered Grid Method and the Histogram Method. While the Tensor method assumes that individual domains can be balanced by equalizing the load in each cartesian direction, the Staggered Grid Method takes a hierarchical approach to balance the load on the processes. First, all domains will be balanced which share the same cartesian coordinate index in the highest dimension (z in 3D). Afterwards, the planes in the decomposition sharing the same coordinate index considering the next lower dimension (y in 3D) will be balanced. Last, the individual subdomains load is compared with the direct neighbors and shifted along the next lower dimension (x in 3D). It results in better load balancing compared to the Tensor Method, but requires the domain neighbor lists to be updated. The Histogram Method works globally and requires the creation of partial histograms to compute a global histogram and, finally, a global work distribution. The partial histograms can be created by calculating the number of particles along a direction in each dimension. This results in the most optimal load balancing but causes the most communication. [6]

The Tensor Method has been implemented in MD-Flexible mainly because its simplicity. It also suits best to compare and evaluate the performance of the Inverted Pressure load balancing method, as both use a non-staggered grid for domain decomposition.

3.4. Technologies

AutoPas and MD-Flexible are implemented in C++ and use OpenMP for thread-level parallelism, the Message Passing Interface (MPI) for node-level parallelism and the Visualization Toolkit (VTK) to display the generated data.

3.4.1. MPI

As mentioned in the theoretical background, processes need to communicate information between each other. This is achieved with help of MPI which has grown to be the industry standard for message passing between processes since it's initial release in 1994. There are

[2] https://slms.pages.jsc.fz-juelich.de/websites/all-website/

many implementations of the interface like OpenMPI[3], the Intel® MPI Library[4] or MPICH[5]. MPI supports a wide range of features commonly used in HPC.

In short, MPI is an interface to pass messages between processors. It provides methods for point-to-point, all-to-one, one-to-all, and all-to-all communication, as well as tools for explicit synchronization. In addition, users are able to group processes into communicators allowing them to restrict communication to a specific set of processes.

An application which integrates MPI is classified as Single-Program-Multiple-Data (SPMD). This means that the application runs the same code on multiple processes, which are called *ranks*, where each rank is responsible for a different set of data. In this class of application, there is always a tradeoff between introducing redundant computations and additional communication. The developers need to make a decision which option they prefer. This will come up several times in Part II.

3.4.2. Visualization Toolkit

The Visualization Toolkit[6] (VTK) provides software for displaying and manipulating scientific data. It is developed by the company Kitware[7] which also develops other well known tools like CMake and Paraview[8].

The toolkit offers different file types to visualize various kinds of scientific data[9]. Each file type offers a serial definition used by applications which only run on a single process and a parallel definition for multi process applications. For instance, for visualizing image data VTK defines two XML file types: *.vti* for serial applications and *.pvti* for parallel applications. The toolkit also provides equivalent file types for rectilinear grids (*.vtr* and *.pvtr*), structured grids (*.vts* and *.pvts*), unstructured grids (*.vtu* and *.pvtu*), and unstructured data (*.vtp* and *.pvtp*).

VTK also includes C++ libraries to generate files of each file type, which then can be displayed using Paraview. Alternatively, these records can also be generated manually which is the case in MD-Flexible.

[3] `https://www.open-mpi.org/`
[4] `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/mpi-library.html`
[5] `https://www.mpich.org/`
[6] `https://vtk.org/`
[7] `https://www.kitware.com/`
[8] `https://www.paraview.org/`
[9] `https://kitware.github.io/vtk-examples/site/VTKFileFormats/`

4. Related Work

Because Massive Parallelism has been around for several decades, there are some well established molecular dynamics projects implementing their own approaches to domain decomposition, or workload sharing, in general.

4.1. GROMACS

GROMACS[1] was first introduced in 1995 by H.J.C Berendsen, D. van der Spoel and R. van Drunen[3]. Although initially, users had to pay to get access to GROMACS, it is now licensed under GNU Lesser General Public License. It is primarily designed for the simulation of biomolecules, supports GPU acceleration using Nvidia's CUDA GPU programming language and is updated multiple times a year.

In contrast to MD-Flexible, GROMACS does not use halo regions to store particles of neighbor domains. Instead it uses the eight shell method to avoid the redundant force calculations. Also it uses a staggered grid approach allowing subdomain boundaries to shift almost independently along a single dimension. This improves the overall load balancing as it enhances the local refinement capabilities of the grid decomposition. But because the domain neighbor lists may change during rebalancing, maintaining a staggered grid introduces additional communication as processes are required to update their domain neighbor lists every time the subdomains have been rebalanced. [1]

4.2. LAMMPS

Another well known project is LAMMPS[2]. It is based on LAMMPS 2001 which has been developed under a cooperative research & development agreement since 1990. The current release is licensed under the GNU general public license version 2. It is still being developed and on average releases one update each year. LAMMPS simulations can run on hybrid compute clusters containing both GPUs and CPUs[4].

In addition to a full shell spatial-decomposition scheme using halo particles, LAMMPS implements two alternative decomposition approaches.

The Atom-Decomposition disperses all particles equally to all processors. The particles which are assigned to a single processor need not to be spatially related in any way. Each processor then is responsible for simulating the assigned particles during the entire simulation which means that particles do not move between processes. This approach not only requires all-to-all communication in each iteration, but also ignores Newton's third law. According to the paper by Steve Plimpton, Atom-Decomposition runs best on a small number of processes or setups where communication cost are expected to be negligible[10].

[1] `http://www.gromacs.org/`
[2] `https://www.lammps.org/`

The second decomposition scheme takes the force matrix F, consisting of all force interactions between particles, and distributes the required force interactions equally to all processes. If only short-range interactions are being considered, this force matrix F is sparse and can be efficiently distributed using well known block-decomposition schemes common in linear algebra. The Force-Decomposition scheme also requires all-to-all communication to update the force matrix, but takes advantage of Newton's third law[10]. If the relation N/P is small, where N refers to the number of particles and P to the number of processes, this approach outperforms LAMMPS' spatial decomposition approach which performs better with growing N/P. [10]

4.3. ls1-mardyn

The last project we want to mention is the ls1-mardyn[3] Molecular Dynamics code base originally developed by the High Performance Computing Center Stuttgart in collaboration with several German universities. Now, it is mainly extended and maintained by the Chair of Scientific Computing at the Technical University of Munich. It focuses on the simulation of thermodynamics and nanofluidics. In 2019 a simulation using ls1-mardyn achieved the world record for the largest particle simulation with over 20 trillion particles[11].

When it comes to load balancing, ls1-mardyn uses an approach similar to k-d trees. It recursively subdivides the simulation domain along division planes which are perpendicular to a coordinate axis. To guarantee the best load balancing, the optimal division plane with the least load imbalance is selected from all possible division planes. The recursion will stop a soon as every subdomain has been assigned to a process. ls1-mardyn also supports diffuse load balancing based on the ALL load balancer. [9]

[3] `https://www.ls1-mardyn.de/home.html`

Part II.

Refactoring MD-Flexible

5. High Level Application Architecture

Before massive parallelism was enabled in MD-Flexible, it implemented only thread-level parallelism, mainly through AutoPas. The design also did not incorporate a possible massively parallel implementation in the future. In other words, integrating massive parallelism turned out to be a challenging task, not only because of the complexity imposed by the domain decomposition and the load balancing, but also because several application components were not suited for usage in a parallel environment.

5.1. Old Application Architecture

5.1.1. Overview

The original application consists of four major components. The *Parser*, the *Configuration*, the *AutoPas Container*, and the *Simulation*. All components are represented in their own class, while the AutoPas container is provided by the AutoPas library. An overview of the program flow can be seen in Figure 5.1. The Parser is responsible for parsing and validating a provided configuration file while the Configuration is taking care of storing the parsed values and for the initialization of the simulation domain. The Simulation component needs to be initialized using an AutoPas container and a valid Configuration. It is responsible for the initialization of the particles and for the orchestration of the simulation. The particles are initialized based either on the objects defined in the configuration file (see Section 3.2) and / or on a checkpoint file and are then assigned to the AutoPas container. Subsequently, the simulation will be executed. If it is successful and the respective configuration parameters have been set, the application prints execution statistics to the console and creates a configuration file containing the configuration which has been used for the simulation.

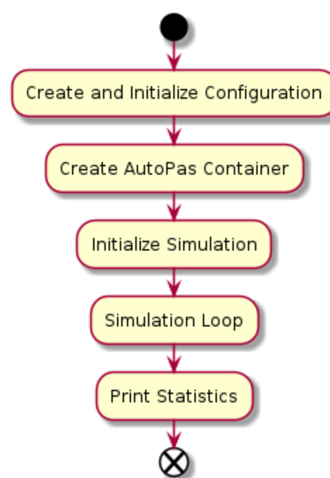


Figure 5.1.: An overview of the old MD-Flexible application.

The Configuration component provides functions to initialize the particles based on objects. Aside from this fact, the initialization of the component is very straightforward and therefore unimportant for the refactoring of MD-Flexible. The same holds for the creation of the AutoPas container as it's implementation was not touched during refactoring.

5.1.2. Initialization of the Simulation Component

It is necessary, though, to look at how the Simulation component has been initialized in the old application. The important steps are summarized in Figure 5.2. During the initialization of the AutoPas container MD-Flexible defines the extend of the container's simulation domain, along with several other configuration parameters. In the old application, the size of the domain corresponds to the global simulation domain. It is therefore defined by the global box's minimum and maximum coordinates represented by the front bottom left corner and the back top right corner of the global domain. After the container has been initialized, the particles will be created using the checkpoint file, if it has been configured, along with the objects defined in the configuration. Because the container encapsulates the whole simulation domain, it is certain that it will hold every particle in the simulation. This allows us to blindly assign the particles to the container, which is not the case in the new application, as we will show later in Subsection 5.2.2. In the last step of the Simulation initialization, the Thermostat is initialized which is responsible for maintaining the temperature during the simulation.

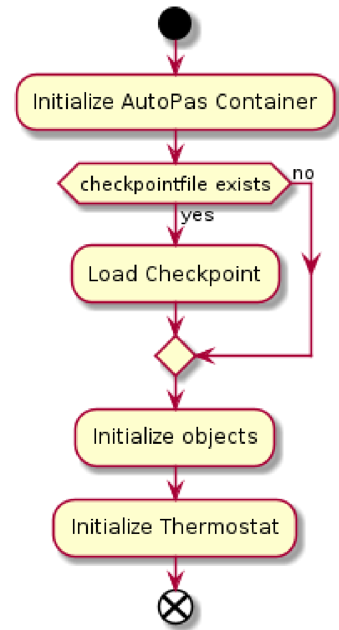


Figure 5.2.: An overview of the old initialization of the Simulation component.

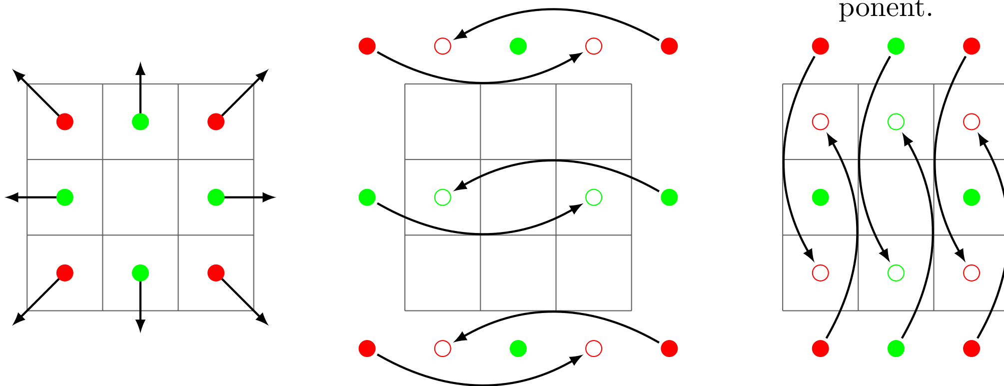


Figure 5.3.: This figure shows the shifting of migrating particles in a 2-dimensional domain with periodic boundaries. The red and green circles represent particles and the arrows migration and shift translations. The left picture shows the particles before leaving the domain and their movement directions, the middle picture shows the particles being shifted along the horizontal coordinate axis, and the right picture shows the particles being shifted along the vertical coordinate axis. Note, that the red particles need to be shifted twice so that they end up in their proper position.

5.1.3. The old Simulation Loop

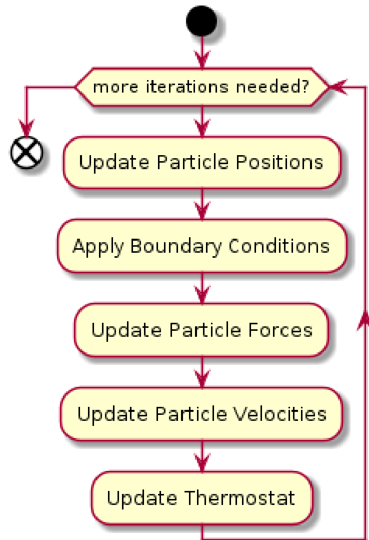


Figure 5.4.: An overview of MD-Flexible's old simulation loop.

The simulation loop (see Figure 5.4) is the heart of the application and has seen most of the changes during the integration of massive parallelism. Initially, it consisted of five major steps, based on the Verlet-Störmer Integration: The update of the particle positions, the application of the boundary conditions to particles which leave the simulation domain, the particle force update, the particle velocity update, and last, the temperature update.

As the first step, MD-Flexible calculates the new particle positions based on the current velocity and the current forces acting on each particle. This update may cause particles to leave the simulation domain, but because MD-Flexible uses periodic boundary conditions, these particles need to be shifted, accordingly. This is handled by the Boundary Conditions step. It is essential to understand what is happening here as the same problem needs to be addressed for a subdivided domain. Leaving particles may cross a single domain boundary for each dimension. In 2D, they may cross two boundaries at the same time (three boundaries in 3D) as can be seen in Figure 5.3. After the migration is complete, the Boundary Conditions step creates the halo particles required for a correct simulation of the particles near the periodic boundaries. This is illustrated in Figure 5.5.

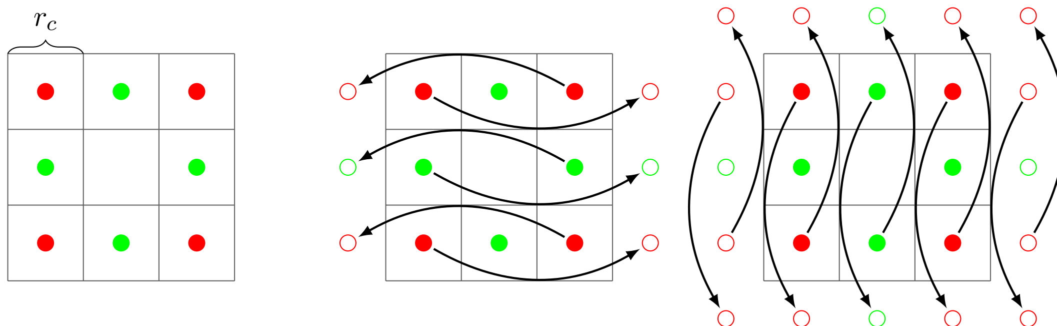


Figure 5.5.: The creation of the halo particles in two steps. The left picture displays the initial particles from which the halo particles will originate. The middle picture shows the first step where the original particles are duplicated and the duplicates shifted along the horizontal coordinate axis to their proper halo positions. The right picture shows the third step, this time shifting and duplicating all relevant particles (halo and originals) along the vertical coordinate axis. In both steps, only the particles within the cutoff radius of the domain boundary need to be addressed. The cutoff radius r_c is equal to the width of a small cell.

Understanding the particle migration and the halo particle creation is very important, not only for the sequential application, but also for the parallel version. Wrong implementations can break any approach for the load balancing, which will be discussed later in Section 6.1.

The next two steps in the simulation loop update the forces acting on the particles and the particles' velocities. Both steps are not touched during the parallelization of MD-Flexible and therefore will not be discussed in detail.

The last step in the loop is the update of the Thermostat which calculates the average temperature of the simulation using the kinetic energy of the particles. In the sequential application the resulting temperature automatically corresponds to the global temperature. This is not the case in the parallel version and therefore needs to be addressed in the implementation.

5.2. New Application Architecture

The massively parallel application has seen several changes compared to the sequential version. Before getting into detail about the implementation we need an overview of those changes and want to understand why they have been implemented. It is also important to keep in mind that an application which integrates MPI is classified as SPMD as mentioned in Subsection 3.4.1.

5.2.1. Overview

The new application flow includes some additional steps and changed how some of the previously existing steps work. An overview of the flow can be seen in Figure 5.6.

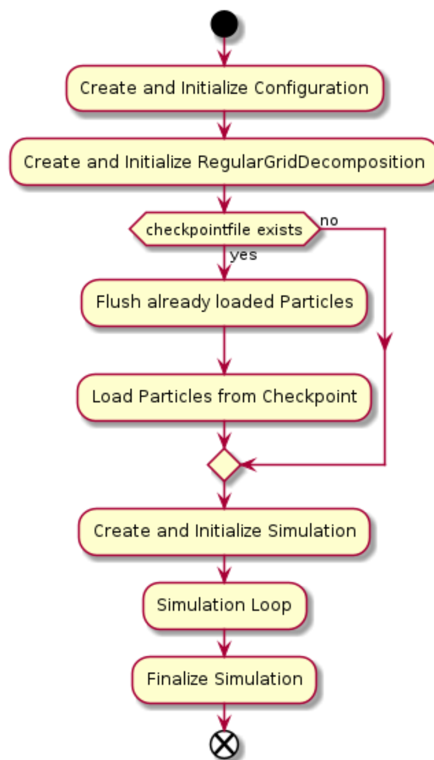


Figure 5.6.: An high level overview of MD-Flexible after massive parallelism has been enabled.

As in the old architecture, the first step is the initialization of the configuration component based on the configuration file and / or on command line parameters. In contrast to the old application, the particles will be initialized during this first step instead of at the start of the simulation. Each process loads all particles in the global domain as the processes do not yet know for which part of the domain they will be responsible. This is decided in the new *Regular Grid Decomposition* component which is initialized directly after the initialization configuration component. Based on the global domain boundaries and the number of available processes, the Regular Grid Decomposition subdivides the domain into subdomains and decides for which subdomain the current process is responsible. Alongside means to migrate particles and to update halo particles, it also provides functions

to update the domain decomposition using a diffuse *work* metric. All this is described in detail in Section 6.1. Each process also gets assigned a domain index corresponding to the MPI rank and a three dimensional domain ID corresponding to the position of their subdomain in the decomposition grid.

This information is important when loading a potentially provided checkpoint. Particles loaded from a checkpoint can be directly assigned to the correct process, if the currently running application uses the same number of processes as were used when the checkpoint has been created. If this is not the case, each process again needs to load all particles and then only add those which lie within their subdomain into their respective AutoPas container.

After the domain decomposition has been created and eventual checkpoints have been loaded, the Simulation component can be initialized and started.

5.2.2. Initialization of the new Simulation Component

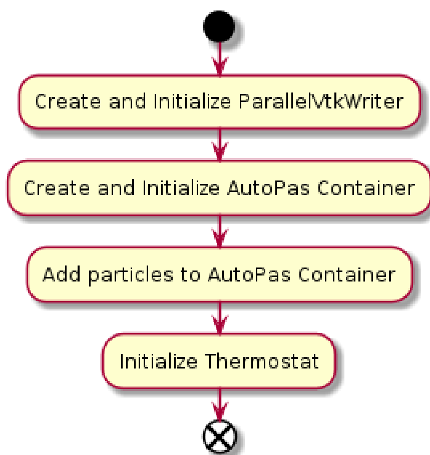


Figure 5.7.: The inititilaztion of the Simulation component in the massively parallel application.

The first step in the new simulation initialization visualized in Figure 5.7 prepares the environment for the creation of the parallel recording of simulation states. In the old application, the Simulation component handled the creation of the simulation records. This has been moved to a new *Parallel VTK Writer* component which provides functions to record the current state of the particles and the current domain decomposition. This component will be described in detail in Subsection 6.5.1.

Aside from the Parallel VTK Writer, the initialization of the new simulation component is very similar to the old version. The AutoPas container is not provided with the global domain boundaries but is configured with the current processes' subdomain, and the creation of the particles has been moved to the configuration component, as mentioned before.

5.2.3. The new Simulation Loop

As in the old application, the new simulation loop (see figure Figure 5.8) is executed as long as additional iterations are required, and again, the first step within the loop is to update the particle positions. Afterwards, the application updates the AutoPas container and, if an update of the AutoPas container actually occurred, rebalances the subdomains and migrates the particles.

The boundary update step has been removed because the boundary conditions are handled during particle migration and the halo particle update. The particles only need to be shifted if their new position lies outside the global domain boundaries. This depends on the receiving neighbor, who is determined before particles are communicated. Additionally, the old implementation of the boundary condition update is not compatible with the new communication scheme described in Section 6.2.

After the halo particles have been exchanged, only the updates for the particle forces and velocities, as well as the update of the Thermostat remain to be performed in the simulation loop. Because the global domain has been subdivided, the Thermostat step requires global communication to determine the current temperature. Here, every process calculates the temperature of it's respective subdomain before receiving the local temperature of the other processes so it can calculate the global average.

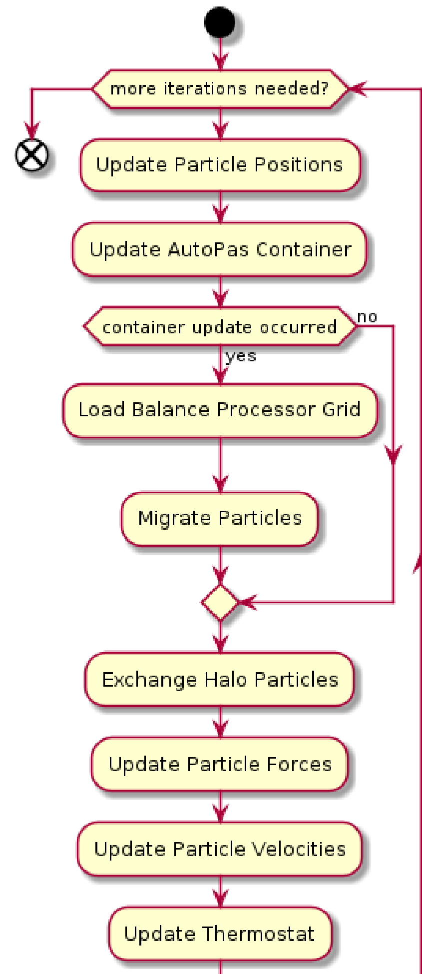


Figure 5.8.: The new Simulation Loop.

5.2.4. Minor Application Features

So far, only the major features required to perform the actual MD simulation have been discussed. Additionally, MD-Flexible includes a progress bar, a logger to get information during runtime, and a simulation summary providing data about the execution time of some steps in the simulation.

The progress bar and the logger can be toggled and will only be displayed on the root process's output stream. The *Timer* component tracks the total execution time of the application, the time it took to initialize the simulation, and the time required for the actual simulation. Additionally, it tracks the timings for the update of the particle positions, forces and velocities, the creation of the simulation records, the exchange of migrating particles and halo particles, time required for load balancing, and thermostat update. All timings are tracked locally for all the processes and will be summed up and printed to the configured

output stream at the end of the simulation. On top of that, the Timer also tracks the work performed by a process since the last decomposition update. This work is not printed to the output stream, but is used as a metric for the load balancing step in the simulation loop.

6. Implementation

Now that we have an idea of what has changed during the integration of massive parallelism into MD-Flexible, we can take a detailed look at the implementation. It followed four major goals: Retaining the possibility to run MD-Flexible serially, generating meaningful metadata to be able to evaluate the application and to identify potential problems, the visualization of particles and domain decomposition, and finally to enable massive parallelism using adaptive domain decomposition.

6.1. Adaptive Domain Decomposition

The adaptive domain decomposition is the biggest addition to MD-Flexible. It influences all other aspects of the refactoring and, therefore, is discussed, first.

6.1.1. Setting up the Regular Grid Decomposition

Before anything else, the global domain is decomposed into subdomains during the initialization of the new `RegularGridDecomposition` class. With the help of `MPI_Comm_size()`, MD-Flexible retrieves the number of available processes and decomposes the global domain into an equal amount of subdomains which is stored in the variable `subdomainCount`. Each MPI rank is then assigned a single subdomain. Note that the implementation does not use MPI functions directly. It uses wrappers, instead, which have been created to retain the capability to compile MD-Flexible without MPI. This is discussed in detail in section Section 6.3. For the sake of simplicity, we will use the function names defined by MPI instead of the function names defined by the MPI wrapper.

The new `DomainTools` class provides the function `generateDecomposition()` which is responsible for generating the decomposition grid. This function takes, along with the number of desired subdomains, an array containing three boolean values. These values indicate which dimensions can be subdivided. An element of the array represents dimension x, y, z, respectively. If a value is `true`, the corresponding dimension can be subdivided. This allows users to restrict the subdivision to specific dimensions to do implicit load balancing (refer to the end of section Section 3.2 for a detailed explanation).

The decomposition is generated by factorizing the variable `subdomainCount` with the help of prime factorization[1] using the Regular Grid Decomposition Generation algorithm presented in Figure 6.1. After the prime factorization, the two smallest factors are multiplied with each other to reduce the number of factors until there are only three or less left, one for each subdividable dimension. This minimizes the difference in the number of cuts along each dimension. As soon as the number of prime factors is less or equal than the number of subdividable dimensions, the prime factors are stored in the `decomposition` variable.

[1] `https://www.tutorialspoint.com/prime-factor-in-cplusplus-program`

The variable stores the number of subdivisions along each dimension and is used for the definition of the local subdomain and, later, for communication.

Algorithm 1: Regular Grid Decomposition Generation

Input: subdomainCount, subdivideDimension
Output: decomposition

```
1  Function generateDecomposition(subdomainCount, subdividableDimension):
2      int[3] decomposition
3      list primeFactors
       // Calculate prime factorization
4      while subdomainCount % 2 == 0 do
5          primeFactors.push_back(2)
6          subdomainCount = subdomainCount / 2
7      for i ← 3; i ≤ subdomainCount; i = i + 2 do
8          while subdomainCount % i == 0 do
9              primeFactors.push_back(i)
10             subdomainCount = subdomainCount / i
       // Determin number of subdividable domains
11     int numberOfSubdividableDimensions
12     for element in subdivideDimension do
13         numberOfSubdividableDimensions += element
       // Shirnk factors to number of subdividable domains
14     while primeFactors.size() > numberOfSubdividableDimensions do
15         sort(primeFactors)
16         set firstElement = primeFactors.front()
17         primeFactors.pop_front()
18         primeFactors.front *= firstElement
       // Assign factors to decomposition
19     for i ← 3; i ≤ subdomainCount; i = i + 2 do
20         if not primeFactors.empty() and subdivideDimension[i] then
21             decomposition[i] = primeFactors.front()
22             primeFactors.pop_front()
23         else
24             decomposition[i] = 1
25     return decomposition
```

Figure 6.1.: Algorithm to create a regular grid decomposition based on a target number of subdomains and subdividable dimensions. It factorizes the number of subdomains using prime factorization and generates a grid subdivision from the resulting factors.

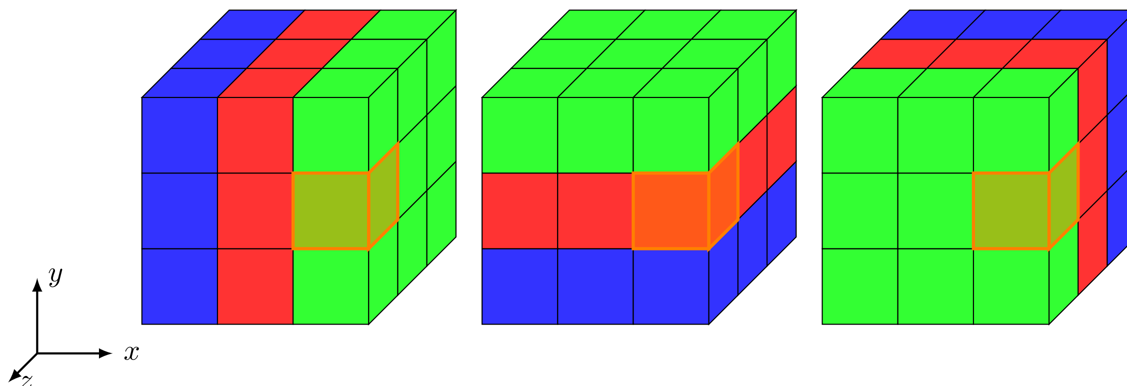


Figure 6.2.: A visualization of the planar communicators in a 3-by-3 grid decomposition of a cube domain. In the left cube, all subdomains which share an equal x-coordinate are grouped into a single communicator. The middle cube shows the grouping by y-coordinate and the right cube by z-coordinate. The highlighted domain has the domainId [2,1,0].

After the decomposition grid has been generated, the cartesian MPI communicator, which is used for inter process messages, is initialized using MPI's `MPI_Cart_create()` function. Each subdomain is assigned a `domainIndex` which corresponds to the process's rank within the new communicator.

This information, along with the `decomposition` is then used to generate the local domain's id in the process grid. The `domainIndex` is passed to the `MPI_Cart_get()` function to retrieve the position of the current process's subdomain within the grid. The position is then stored as the `domainId`, which afterwards is used to initialize the planar communicators required for the *Inverted Pressure* load balancing algorithm described in Subsection 6.1.3. It is important to note, that the `domainId` referes to a three-dimensional array storing the position of the domain in the decomposition grid, while the `domainIndex` refers to the index of all subdomains within an enumeration. Every process in the cartesian communicator which shares an index at the same position in the `domainId` is considered part of the same planar communicator, making each process part of three planar communicators. Figure 6.2 visualizes these communicators in a 3-by-3 grid decomposition. They are created using MPI's `MPI_Comm_split()` function. The method requires, along with the original communicator and the `domainId`, a key which defines the rank of the process within the new communicator. This key is generated by enumerating each process within the new communicators plane.

Before defining the process's subdomain extent, the `RegularGridDecomposition` stores the extent of the global simulation domain. The size of the process's subdomain is determined by the `domainId` and by the number of subdivisions along each dimension that are stored in the `decomposition` variable. The width of a subdomain is defined by the number of subdomains and by the global domain's width along a coordinate axis. The resulting subdomains all have the same side lengths. The minimum and the maximum value in a subdomain are stored as `localBoxMin` and `localBoxMax` and are calculated according to the process's `domainId`, as can be see in the algorithm presented in Figure 6.3.

Algorithm 2: Initialization of a processes subdomain minimum and maximum value

Input: domainId, decomposition, globalBoxMin, globalBoxMax
Output: localBoxMin, localBoxMax

```
1 Function initializeLocalBox():
2     for i ← 0 to 2 do
3         int localBoxWidth = (globalBoxMax[i] - globalBoxMin[i]) / decomposition[i]
4         localBoxMin[i] = domainId[i] * localBoxWidth + globalBoxMin[i]
5         localBoxMax[i] = (domainId[i] + 1) * localBoxWidth + globalBoxMin[i]
          // Directly assign globalBoxMax value to avoid any numeric
             errors
6         if domainId[i] == decomposition[i] - 1 then
7             localBoxMax[i] == globalBoxMax[i]
```

Figure 6.3.: Algorithm to determine front bottom left corner (minimum) and back top right (maximum) coordinates of a process's subdomain.

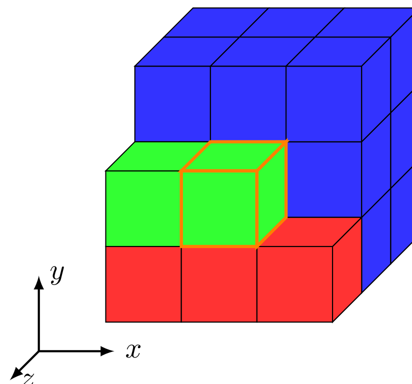


Figure 6.4.: The index of the highlighted subdomain is computed by summing up the number of blue domains first, then adding the number of red domains and finally the number of green domains.

The domain neighbor lists required for communication are created as the previous to last step during the initialization of the `RegularGridDecomposition` class. Generating the domain neighbor list does not require any communication between processes because `domainIndex` and `domainId` are directly correlated. Therefore, the domain neighbors `domainIndex` (which corresponds to the rank of the neighbor process) can be derived from the current process's `domainId` using the algorithm displayed in Figure 6.5. To map an id to the corresponding index, the `DomainTools` library provides the function `convertIdToIndex()`. How this conversion works is illustrated in Figure 6.4. The communication scheme which is described later in this Chapter only requires the direct neighbors excluding any diagonal neighbors. Consequently, it is enough to compute six neighbor indices which are stored

Algorithm 3: Initialization of the current process's domain neighbors

```
Input:  domainId, decomposition
Output: neighborDomainIndices

   // Algorithm for domain neighbor initialization
1  Function initializeNeigborIndices():
2      int[6] neighborDomainIndices
3      for i ← 0 to 2 do
           // Calculate index of left / preceeding neighbor
4          int neighborIndex = i * 2
5          int[3] preceedingNeighborId = domainId preceedingNeighborId[i] =
            (preceedingNeigborId[i] + decomposition[i]) % decomposition[i]
6          neighborDomainIndices[neighbourIndex] =
            convertIdToIndex(preceedingNeighbourId, decomposition)
           // Calculate index of right / succeeding neighbor
7          int[3] succeedingNeighborId = domainId
8          succeedingNeighborId[i] = (++succeedingNeigborId[i] + decomposition[i]) %
            decomposition[i]
9          neighborDomainIndices[neighbourIndex + 1] =
            convertIdToIndex(succeedingNeighbourId, decomposition)
10     return neighborDomainIndices

   // Algorithm to convert a domain id to its corresponding index
11 Function convertIdToIndex(domainId,decomposition):
12     int domainIndex = 0
13     for i ← 0 to 2 do
14         int accumulatedTail = 1
15         if index < decomposition.size() - 1 then
16             accumulatedTail = accumulate(decomposition.begin() + index + 1,
                decomposition.end(), 1);
17         domainIndex = domainIndex + accumulatedTail
18     return domainIndex
```

Figure 6.5.: Algorithm to determine the current process's domain neighbors. The resulting indices correspond to the respective process's rank. The function `accumulate` is a placeholder to the c++ standard library function std::accumulate. Here, it accumulates the given range by multiplying the values.

in pairs of left and right neighbors along each dimension. This means, that the first two values in the resulting array correspond to the left and right neighbor along dimension x, the second two values to the left and right neighbor along dimension y and the last two values along dimension z.

Setting up the ALL's Tensor load balancer is the final step in the initialization of the decomposition. To initialize the load balancer, the following parameters are required: The method which will be used for load balancing, the cartesian communicator which has been created earlier, the number of dimensions used in the simulation, and a minimum subdomain size. Refer to ALL's official documentation[2] for a detailed description on how to initialize the load balancer.

During the initialization of the `RegularGridDecomposition` class, a flag `mpiCommunicationNeeded` is enabled if multiple processes have been allocated for the application. It is disabled otherwise. This flag is used to ensure that no communication occurs if MD-Flexibe only runs on a single process and is discussed in more detail in Section 6.3.

6.1.2. Diffuse Load Balancing of Regular Grid using the ALL load balancing library

Most of the variables initialzied during domain decomposition are used either for the communication between processes or the *Inverted Pressure* load balancing method. ALL's *Tensor* load balancing method only requires the cartesian communicator at initialization and the local box coordinates of the subdomain owned by the current process. During the balancing step illustrated in Listing 6.1, MD-Flexible converts the local box boundaries to the type `ALL::Point` which is defined by the library. These points then are passed to the ALL load balancer along with the process's performed work. Afterwards, MD-Flexible calls the library function `balance()` which calculates the balanced local box coordinates. They can then be retrieved using the `getVertices()` function and assigned to the local box.

To calculate the coordinates of the balanced local box, ALL takes the difference of the work done by the process grid plane on the right and the left of the shiftable boundary. The difference is then normalized and divided by combined domain sizes of the two involved process grid planes along the shift direction. This means, ALL's Tensor method does not consider the global work along the shift axis, similar to the Inverted Pressure method, as we will see in Subsection 6.1.3.

The shift length is also scaled by a factor $\frac{1}{\gamma}$ to improve the stability of the method. The *gamma* value is calculated on the fly and cannot be influenced by the user.

The balanced coordinates are only calcualted by a single process in the process grid plane an need to be communicated to the other processes afterwards.

```
1 void balanceWithAllLoadBalancer(const double &work) {
2   std::vector<ALL::Point<double>> domain(2, ALL::Point<double>(3));
3
4   for (int i = 0; i < 3; ++i) {
5     domain[0][i] = _localBoxMin[i];
6     domain[1][i] = _localBoxMax[i];
7   }
```

[2] `https://slms.pages.jsc.fz-juelich.de/websites/all-website/`

```
8
9     _allLoadBalancer->setVertices(domain);
10    _allLoadBalancer->setWork(work);
11    _allLoadBalancer->balance();
12
13    std::vector<ALL::Point<double>> updatedVertices = _allLoadBalancer->
          getVertices();
14
15    for (int i = 0; i < 3; ++i) {
16      _localBoxMin[i] = updatedVertices[0][i];
17      _localBoxMax[i] = updatedVertices[1][i];
18    }
19  }
```

Listing 6.1: Implementation of the load balancing using ALL's TENSOR method.

6.1.3. Diffuse Load Balancing using the Inverted Pressure Method

The *Inverted Pressure* load balancing method is based on the scenario depicted in Figure 6.6: Two adjacent closed volumes of air each with a different air pressure press against a movable wall between the domains.

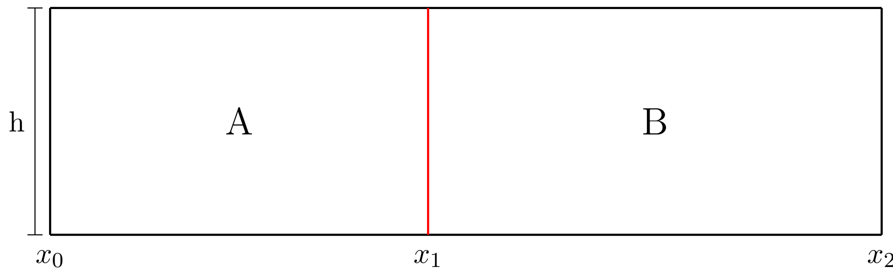


Figure 6.6.: Base scenario of the Inverted Pressure Load Balancing Method. A and B are the domains with different pressure and the red line represents the movable wall. h represents the height of the domains and x_0, x_1 and x_2 the coordinates required to calculate the width of the domains.

If we assume that the pressure in domain A is higher compared to domain B, the pressure of domain A would shift the movable wall to the right until the pressure in A and B are equalized. But how do we calculate the new ideal position x'_1 where the pressure is balanced? First, we define pressure as "work per area" in a given domain. So the pressure would be

$$P_A = w_A/a_A$$

where w_A is the work performed in domain A and a_A refers to the area of A. The pressure is defined for B, equivalently. As mentioned before, the scenario is in an equilibrium state if $P_A = P_B$. With the area of the domains being $a_A = (x_1 - x_0) * h$ and $a_B = (x_2 - x_1) * h$, the following equation describes the balanced domains:

$$\frac{w_A}{(x_1 - x_0)h} = \frac{w_B}{(x_2 - x_1)h}$$

Unfortunately, using this equation to calculate the balanced x-position of the shiftable wall would increase the area of A and therefore the work performed in A. This is exactly the opposite of our goal. To counteract this, we just use the inverse of the pressure resulting in the following equation:

$$\frac{(x_1 - x_0)h}{w_A} = \frac{(x_2 - x_1)h}{w_B}$$

Solving this equation for x_1 tells us where to position the movable wall for a perfectly balanced workload between domain A and B:

$$x_1' = \frac{w_B x_2 + w_A x_0}{w_A + w_B} \quad (1)$$

The Inverted Pressure load balancing algorithm uses this equation to balance adjacent planes in the process grid generated by the domain decomposition. For this, we first have to calculate the average work performed in all subdomains in a process grid plane and send the result to the respective neighbor process in the adjacent plane. Because the algorithm only performs a local update, it is not required to share the average work of any other process grid planes than the adjacent one. Having received the neighbor plane's average work, each participating process can calculate the shifted position, independently. This is done in parallel for every pair of adjacent process grid planes along a coordinate axis. The domains are first balanced along the x-axis, then along the y-axis and finally along the z-axis. The algorithm does not consider already shifted planes but only uses the plane coordinates as they were at the start of the load balancing.

Because the algorithm only takes the two adjacent planes into account, it is not globally optimal and may even shift the minimum boundary of a domain to the right of the maximum boundary. To counteract this, the load balancer only shifts the boundary by half of the distance between the "optimal" position and the old position. This does not reduce the efficiency of the balancing because equation (1) only assumes that a single domain boundary is shifted. In most cases though, both boundaries along a coordinate axis are shifted. So the domain boundaries are shifted from both sides by half of the "optimal" distance. Figure 6.7 and Figure 6.8 illustrate the algorithm in pseudo code. First, we need to make a backup of the old local box coordinates as the local box might get overwritten before all data has been sent. Also, it is important to use the old box values for the balancing of the domains. Otherwise the balancing might create gaps between the domains which are not simulated by any process.

The algorithm contains two separate loops which both iterate over the three coordinate axis. The average work in a process grid plane is calculated during the first loop, along with the non-blocking send calls to communicate the result to the respective neighbor processes. The `Allreduce()`, `Send()` and `Recv()` functions used in the algorithm are placeholders for MPI functions. The second loop handles the receive calls for the previously communicated data. When the work of the adjacent plane has been received, the algorithm calls the `balanceAdjacentDomains()` function which is part of the `DomainTools` library. Before applying the balanced position to the local box, it is shifted by half of the original distance in direction of the respective local box value. This prevents the minimum boundary from shifting beyond the maximum boundary, as mentioned in the beginning of the previous paragraph.

The `balanceAdjacentDomains()` function does nothing more but evaluating equation (1) which we derived in the beginning of the section.

This is the Inverted Pressure Method currently implemented in MD-Flexible. As we will see in Part IV, there is much potential for improvement when it comes to efficient communication and precision of the load balancing.

Algorithm 4: First half of Inverted Pressure Load Balancing

```
   Input:   work
1  Function balanceWithInvertedPressureLoadBalancer(work):
2      double[3] oldLocalBoxMin = localBoxMin
3      double[3] oldLocalBoxMax = localBoxMax
4      double[3] distributedWorkInPlane
       // Iterator over coordinate axes
5      for dimensionIndex ← 0 to 2 do
           // Calculate number of domains in process grid plane
6          int domainCountInPlane = decomposition[(i + 1) % dimensionCount] *
            decomposition[(i + 2) % dimensionCount]
           // Calculate average work in process grid plane
7          distributedWorkInPlane[i] = work
8          if domainCountInPlane ¿ 1 then
9              Allreduce (work, distributedWorkInPlane[i], planarCommunicators[i])
10             distributedWorkInPlane[i] = distributedWorkInPlane[i] /
                domainCountInPlane
           // Identify left and right neighbor
11         int leftNeighbor = neighborDomainIndices[i * 2]
12         int rightNeighbor = neighborDomainIndices[i * 2 + 1]
           // Send work and box coordinates to the left and right neighbor
13         if localBoxMin[i] != globalBoxMin[i] then
14             Send (distributedWorkInPlane[i], leftNeighbor)
15             Send (oldLocalBoxMax[i], leftNeighbor)
16         if localBoxMax[i] != globalBoxMax[i] then
17             Send (distributedWorkInPlane[i], rightNeighbor)
18             Send (oldLocalBoxMin[i], rightNeighbor)
```

Figure 6.7.: The first half of the load balancing using the Inverted Pressure method.

Algorithm 5: Second half of the Inverted Pressure Load Balancing

```
   Input:    work
1  Function balanceWithInvertedPressureLoadBalancer(work):
       // Iterate over coordinate axes
2      for dimensionIndex ← 0 to 2 do
           // Identify left and right neighbor
3          int leftNeighbor = neighborDomainIndices[i * 2]
4          int rightNeighbor = neighborDomainIndices[i * 2 + 1]
5          double neighborPlaneWork, neighborBoundary, balancedPosition
           // Receive work from neighbour and balance left boundary
6          if localBoxMin[i] != globalBoxMin[i] then
7              Recv (neighborPlaneWork, leftNeighbor)
8              Recv (neighborBoundary, leftNeighbor)
9              balancedPosition = balanceAdjacentDomains (neighborPlaneWork,
                 distributedWorkInPlane[i], neighborBoundary, oldLocalBoxMax[i], 2 *
                 (cutoffWidth + skinWidth))
10             localBoxMin[i] += (balancedPosition - localBoxMin[i]) / 2
           // Receive work from neighbour and balance right boundary
11         if localBoxMax[i] != globalBoxMax[i] then
12             Recv (neighborPlaneWork, rightNeighbor)
13             Recv (neighborBoundary, rightNeighbor)
14             balancedPosition = balanceAdjacentDomains
                 (distributedWorkInPlane[i], neighborPlaneWork, oldLocalBoxMin[i],
                 neighborBoundary, 2 * (cutoffWidth + skinWidth))
15             localBoxMax[i] += (balancedPosition - localBoxMax[i]) / 2
```

Figure 6.8.: The second half of the load balancing using the Inverted Pressure method.

6.2. Point-to-Point Communication for Sending and Receiving Particles

With the regular grid decomposition set up, the only thing left for the processes to do before the start of the actual simulation is to initialize their respective AutoPas container. As mentioned in Subsection 5.2.2, each process only adds those particles to the container which lie within their own subdomain. Consequently, processes need to communicate during the runtime of simulation. To be exact, for any migrating particle or halo particle, following particle attributes have to be included in the messages: Id, position, velocity, force, old force, and type id. For us, it is not important to understand the purpose of these attributes. It is only important to recognize, that the attributes have multiple data types.

6.2.1. Serialization and Deserialization of Particles

The challenge here is to send the heterogeneous data using MPI, while MPI, by default, only supports messages of a single data type. On top of that, particles are not communicated one particle at a time, but in groups. To be able to send a single or multiple particles using a single MPI message, each particle needs to be serialized, send and finally deserialized, once received by another process.

In C and C++, the simplest way to serialize heterogeneous data is to store the data in a struct, create a pointer of the desired message type pointing at the struct in memory, and then pass this pointer to MPI. This also can easily be done using a vector of structs. Unfortunately, this approach requires developers to modify code at multiple locations, if the communicated data changes.

MD-Flexible avoids this by making use of the C++ fold expressions which have been introduced in C++17[3]. The application implements a custom library called `ParticleSerializationTools` for the purpose of serializing and deserializing particles. It provides a function `serializeParticle()` which stores a particle's data at the end of a provided vector of type `char`. The library also contains the function `deserializeParticle()` to restore the data of a single particle, and the function `deserializeParticles()` to deserialize data of multiple particles. The class `MoleculeLJ`, which is used in MD-Flexible for the particles, defines an enumeration containing all attributes of the particle. Additionally, it provides a generic getter to retrieve any of those attributes. The fact that this getter is generic allows MD-Flexible to use the before mentioned fold expressions. Listing 6.2 shows an example of how a developer might use the getter to retrieve a particle's velocity.

```
1 MoleculeLJ particle;
2 std::array<double, 3> velocity{};
3 velocity[0] = particle.get<MoleculeLJ::AttributeNames::velocityX>();
4 velocity[1] = particle.get<MoleculeLJ::AttributeNames::velocityY>();
5 velocity[2] = particle.get<MoleculeLJ::AttributeNames::velocityZ>();
```

Listing 6.2: Example of MolecIueLJ's generic getter used here to retrieve the particle position.

The `serializeParticle()` and the `deserialzieParticle()` functions are both wrappers for a corresponding generic function which actually implements the de-/serialization. Listing 6.3 shows the implementation of the `serializeParticle()` function.

```
1  template <size_t... I>
2  void serializeParticleImpl(const ParticleType &particle, std::vector<char> &
       serializedParticle,
3                             std::index_sequence<I...>) {
4    // Serialize particle attributes
5    size_t startIndex = 0;
6    std::vector<char> attributesVector(AttributesSize);
7    (serializeAttribute<I>(particle, attributesVector, startIndex), ...);
8
9    // Add serialized attributes to serialized particle
10   serializedParticle.insert(serializedParticle.end(), attributesVector.begin()
       , attributesVector.end());
11 }
```

Listing 6.3: The implementation of serializeParticle using the expansion operator.

[3] `https://en.cppreference.com/w/cpp/language/fold`

The function takes a set of indices, called an *index sequence*, as template argument. In Listing 6.3 it is represented by the template parameter `I`. As can be seen in line 12 of Listing 6.3, the fold expression then takes care of calling an additional generic function `serializeAttribute()` for every index in the index sequence. The implementation of the `serializeAttribute()` function, presented in Listing 6.4, calls the generic getter provided by the `MoleculeLJ` class.

```
1 template <size_t I>
2 void serializeAttribute(const ParticleType &particle, std::vector<char> &
     attributeVector, size_t &startIndex) {
3   const auto attribute = particle.get<Attributes[I]>();
4   const auto sizeOfValue = sizeof(attribute);
5   std::memcpy(&attributeVector[startIndex], &attribute, sizeOfValue);
6   startIndex += sizeOfValue;
7 }
```

Listing 6.4: Implementation of the particle attribute serialization.

The `Attributes` array, used when passing a value to the template parameter of MolecyleLJ's `get()` function, is defined in the `ParticleSerializationTools` library and contains the enumeration values of all the particle attributes which we want to send to another process. This array is one of the two locations which need to be maintained by developers, if the communicated attributes would change. The second location is the `AttributesSize` variable, also defined in the `ParticleSerializationTools` library. This variable stores the accumulated size of all communicated particle attributes in bytes and is required to initialize the target buffer for the serialized attributes, as can be seen in line 6 of Listing 6.3.

So even when using the fold expression, developers need to update two locations in the source code when changing the particle attributes which are relevant for other processes.

The implementation for the `deserializeParticle` and the `deserializeParticles` functions follows the same principles and will not be discussed in detail.

6.2.2. Sending and Receiving Point-to-Point Messages

The `ParticleSerializationTools` are used by the `ParticleCommunicator` class which handles point-to-point communication of particle data in MD-Flexible.

The `ParticleCommunicator` is initialized with the MPI communicator created by the `RegularGridDecomposition` class and provides, among others, the `sendParticles` method which receives a vector of unserialized particles and the rank of the receiving process. The function then serializes the particles and sends the data to the receiver using the non-blocking `MPI_Isend` function. Non-blocking means, that the program continues execution, without making sure that the message has actually reached it's target. On the one hand, this allows MD-Flexible to overlap communication and therefore increases overall concurrency in the program. On the other hand, non-blocking sends allow the program to delete the message content from memory before it actually has been sent to the receiver. To avoid this issue, the `ParticleCommunicator` stores each send request and the connected data in vectors. The users then have to call the function `waitForSendRequests()` if they want to make sure, that every message successfully reached it's receiver. When calling `waitForSendRequests()`, every issued send is checked for completion, in which case the send request and the corresponding message buffer are deleted.

Along with the `sendParticles()` function for sending messages, processes receive messages by calling the `receivceParticles()` function. Here, particles are deserialized and appended to a buffer provided by the user. The method uses the blocking `MPI_Receive()` function to wait for expected sends. Using a blocking function prevents deadlocks with help of implicit synchronization between the sender and the receiver. On top of that, the receiver requires the expected data to continue it's execution.

6.2.3. Step-wise communication with neighbor domains

With the point-to-point communication in place, every tool required for processes to communicate with their domain neighbors is set up. Now it is time to talke about the employed communication pattern which should minimize the strain the application puts on the network. This reduces the risk of congestion, consequently reducing wait times, and can either be achieved by sending less data overall or by sending fewer messages at a time. While the solution in MD-Flexible focuses on the latter, Subsection 2.2.2 introduced the eighth shell method which not only reduces the number of messages but also the average size of the communicated data. Nonetheless, independent of how an application handles inter domain force calculation, the amount of messages sent at once can be reduced by using the *step-wise communication scheme*.

To understand the advantages of *step-wise communication* we will look at a naive approach first. Generally, in a 3D regular grid subdivision of a domain with periodic boundaries, every process has a total of 26 neighbors with which it has to exchange information in each iteration. This can be simply implemented by sending 26 messages at once, consequently putting a massive strain on the network, because every process sends these messages at a single point in time resulting in a total of $26 \times p$ requests. p refers to the number of processes which have been allocated for the application.

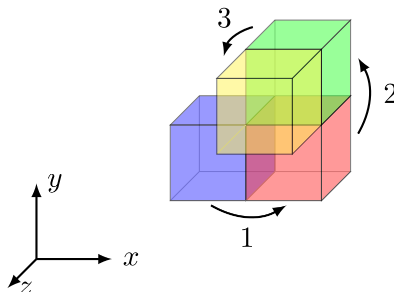


Figure 6.9.: The step-wise commmunication pattern for a 3D regular grid decomposition of a cube domain. Here, only the communication to the right neighbor along the respective coordinate axis is visualized. Assuming a particle migrates from the blue to the yellow domain, it gets send first from blue to red, then from red to green and last from green to yellow.

The load on the network can be spread out over time by splitting the communication into three steps. Each step then handles only those messages passed to a neighbor along a specific coordinate axis. Practically this means, that in the first step messages are send to the left and right neighbor along the x-axis, the second step sends messages along the y-axis

and the last step along the z-axis. Figure 6.9 displays this approach while only visualizing the communication to the right. Every process sends 3×2 messages, making a total of 6 messages. Compared to the naive approach, step-wise communication not only spreads the communication out, but also reduces the number of total of messages sent by a process.

Algorithm 6: Exchange of Migrating Particles

```
   Input:  autoPasContainer, emigrants
1  Function exchangeMigratingParticles(autoPasContainer, emigrants):
2      for dimensionIndex ← 0 to 2 do
3          vector<ParticleType> immigrants, remainingEmigrants,
             particlesForLeftNeighbor, particlesForRightNeighbor
           // Retrieve neighbors from neighbor list
4          int leftNeighbor = neighborDomainIndices[(dimensionIndex * 2) %
             neighborCount]
5          int rightNeighbor = neighborDomainIndices[(dimensionIndex * 2 + 1) %
             neighborCount]
           // Determine which particles need to be sent to the left and
              right neighbor.
6          categorizeParticlesIntoLeftAndRightNeighbor (emigrants,
             dimensionIndex, particlesForLeftNeighbor, particlesForRightNeighbor,
             remainingEmigrants)
7          emigrants = remainingEmigrants
           // Communnicate particles to the respective
8          sendAndReceiveParticlesLeftAndRight (particlesForLeftNeighbor,
             particlesForRightNeighbor, leftNeighbor, rightNeighbor, immigrants)
           // Add particles to AutoPas container
9          for particle in immigrants do
10             if isInsideLocalDomain (particle.getPosition())) then
11                 autoPasContainer.addParticle(particle);
12             else
13                 emigrants.push_back(particle);
       // Add remaining emigrants to current container
14     for particle in emigrants do
15         autoPasContainer.addParticle(particle);
```

Figure 6.10.: Exchange of migrating particles using step-wise communication where each rank only needs to communicate with 6 neighbors instead of 26 (as in the naive approach). The function `sendAndReceiveParticlesLeftAndRight()` contains sends and is called 3 times over the loop iterations resulting in a total of 6 communications.

Algorithm 7: Identifying particles for the left and right neighbor along a desired coordinate axis

Input: particles, direction, leftNeighborParticles, rightNeighborParticles, uncategorizedParticles

```
1  Function categorizeParticlesIntoLeftAndRightNeighbor(particles, direction,
   leftNeighborParticles, rightNeighborParticles, uncategorizedParticles):
2      double[3] globalBoxLength = sub (globalBoxMax, globalBoxMin)
       // Reserve memory to reduce amount of memory allocations.
3      leftNeighborParticles.reserve(particles.size() / 3)
4      rightNeighborParticles.reserve(particles.size() / 3)
5      uncategorizedParticles.reserve(particles.size() / 3)
6      for particle in particles do
7          double[3] position = particle.getPosition()
8          if position[direction] < localBoxMin[direction] then
9              leftNeighborParticles.push_back(particle);
               // Apply boundary condition
10             if localBoxMin[direction] == globalBoxMin[direction] then
11                 position[direction] = min (nextafter (globalBoxMax[direction],
                    globalBoxMin[direction]), position[direction] +
                    globalBoxLength[direction])
12                 leftNeighborParticles.back().setPosition(position)
13         else if position[direction] ≥ localBoxMax[direction] then
14             rightNeighborParticles.push_back(particle)
               // Apply boundary condition
15             if localBoxMax[direction] == globalBoxMax[direction] then
16                 position[direction] = max (globalBoxMin[direction], position[direction]
                    - globalBoxLength[direction])
17                 rightNeighborParticles.back().setPosition(position);
18         else
19             uncategorizedParticles.push_back(particle);
```

Figure 6.11.: Sorts particles into containers which are sent to the left or right neighbor, respectively. This function also applies the boundary conditions to any particle which is sorted into one of the before mentioned containers.

The step-wise communication scheme is also rather easily implemented which is illustrated in Figure 6.10. The algorithm loops over every dimension. In every loop iteration it takes the emigrating particles and classifies them into ”for left neighbor” or ”for right neighbor” depending on their position. The correct neighbors are identified using the neighbor list generated during the domain decomposition (see Subsection 6.1.1). Along with the categorization, the function `categorizeParticlesIntoLeftAndRightNeihbor()` also

applies the boundary conditions discussed in Subsection 5.1.2. The implementation of this function is illustrated in Figure 6.11. As can be seen in the Figure 5.3, the green particles migrating to the top and the bottom domain are not shifted during the first step. Equivalently, they are not communicated during the first step which is why they cannot yet be assigned to the left or right neighbor during the first loop iteration and therefore have to be stored as emigrants for the next loop iteration. Once the particles have been sorted out, they are send to their respective neighbors using the function `sendAndReceiveParticlesLeftAndRight()` (see Figure 6.12). This function also stores any particles received as potential immigrants. They are then added to the AutoPas container, if they actually lie within the local domain, or are included into the emigrants to be forwarded to other neighbors in later iterations of the loop. After exiting the loop, all remaining emigrants are added to the AutoPas container. Here, we do not need to check if they lie within the local domain, because every emigrant that is left meets this condition.

The function `categorizeParticlesIntoLeftAndRightNeihbor()` described by Figure 6.11 works simply by checking the particles position along the desired coordinate axis. If it lies left of the local boxes minimum, the particle is included into the `leftNeighborParticles` and to the `rightNeighborParticles`, if it lies on or right of the local boxes maximum. If none is the case, the particle is included into the `uncategorizedParticles`. Additionally, particles are shifted to the other side of the global box domain, if the respective local box boundary is equal to the global box boundary. This boundary condition update is applied in lines 11-13 and 17-19 of Figure 6.11, depending on the particle's categorization.

Algorithm 8: Sends / receives particles to / from the left and right neighbor

Input: particlesToLeft, particlesToRight, leftNeighbor, rightNeighbor, receivedParticles

```
1  Function sendAndReceiveParticlesLeftAndRight(particlesToLeft,
     particlesToRight, leftNeighbor, rightNeighbor, receivedParticles):
2      if mpiCommunicationNeeded and leftNeighbor != domainIndex then
3          ParticleCommunicator particleCommunicator(communicator)
4          particleCommunicator.sendParticles(particlesToLeft, leftNeighbor)
5          particleCommunicator.sendParticles(particlesToRight, rightNeighbor)
6          particleCommunicator.receiveParticles(receivedParticles, leftNeighbor)
7          particleCommunicator.receiveParticles(receivedParticles, rightNeighbor)
8          particleCommunicator.waitForSendRequests()
9      else
10         receivedParticles.insert(receivedParticles.end(), particlesToLeft.begin(),
             particlesToLeft.end())
11         receivedParticles.insert(receivedParticles.end(), particlesToRight.begin(),
             particlesToRight.end())
```

Figure 6.12.: Sends particles to the left and right neighbor while also receiving potential immigrants.

The before mentioned function `sendAndReceiveParticlesLeftAndRight()` illustrated in Figure 6.12 encapsulates the communication of the categorized particles. Not only does it send them to their respective targets but also receives potential immigrating particles. Here, we make use of the `ParticleCommunciator` class introduced in Subsection 6.2.2.

6.3. Serial Simulation

Although this paper focuses on parallelization, the main purpose of MD-Flexible is to be a suitable demonstrator for AutoPas. Therefore, users might not be interested in the parallel performance of multiple containers but rather in the effectiveness of AutoPas' auto-tuning feature. For this reason it is important to retain the possibility to run MD-Flexible only using a single process.

Practically, this means that MPI should only be used if multiple processes have been allocated for the program execution. This is achieved with a wrapper library called `WrapMPI` which actually has been implemented by the AutoPas library. This wrapper allows the users to exclude MPI during compile time, making it impossible to execute MD-Flexible with multiple processes while using the generated binary. Whether MPI is included during compile time is decided by a cmake option called MD_FLEXIBLE_USE_MPI which can be enabled by appending *MD_FLEXIBLE_USE_MPI=ON* to the *cmake* command or toggling it in the ccmake front-end. This option is disabled by default.

```
1  inline int AutoPas_MPI_Comm_size(AutoPas_MPI_Comm comm, int *size);
2
3  #if defined(AUTOPAS_INCLUDE_MPI)
4  inline int AutoPas_MPI_Comm_size(AutoPas_MPI_Comm comm, int *size) {
5    return MPI_Comm_size(comm, size);
6  }
7  #else
8  inline int AutoPas_MPI_Comm_size(AutoPas_MPI_Comm comm, int *size) {
9    if (nullptr == size) {
10     return AUTOPAS_MPI_ERR_ARG;
11   }
12   *size = 1;
13   return AUTOPAS_MPI_SUCCESS;
14 }
15 #endif
```

Listing 6.5: Example for wrapped MPI function.

If MPI has been included during compile time, MD-Flexible can be executed with any number of processes. When only a single process is allocated for the application, MD-Flexible behaves the same way as if MPI had not been enabled. This is ensured using the flag `mpiCommunicationNeeded` which has been set during the initialization of the regular grid decomposition (see end of Subsection 6.1.1).

Using preprocessor directives, the WrapMPI library either implements dummy functions which essentially do nothing if MPI has not been enabled. Otherwise, it implements functions which actually call the respective MPI function. In both cases, the functions have the same name and signature allowing the developer who uses the WrapMPI library to avoid considering the fact that MPI might not be available. The Listing 6.5 contains an example using the `MPI_Comm_size()` function. Note that the signature of the `AutoPas_MPI_Comm_size()`

function takes the type `AutoPas_MPI_Comm` as input; WrapMPI does not only wrap MPI functions but also MPI data types and objects. Also note that the preprocessor directive at line 3 of the listing uses the macro `AUTOPAS_INCLUDE_MPI`. This variable is used in several locations within MD-Flexible's source code and is defined by cmake if the option `MD_FLEXIBLE_USE_MPI` has been enabled.

6.4. Meaningful Metadata

MD-Flexible generates statistics for every simulation, since before the integration of massive parallelism. The statistics include execution times of specific steps, like the time required for force calculation and position updates. If a simulation uses multiple processes, these timings do not truthfully represent the actually required CPU time because they only would represent the data of a single process. To counteract this, the local timings are summed up at the end of the simulation using `MPI_Allreduce()`. The actual wall-clock time is only reported by the rank 0 and does not require any communication.

As discussed in Subsection 5.2.3, MD-Flexible's new version replaced the boundary update timer by the load balancing, particle migration, and halo particle exchange step timers. For each of them, the CPU time is tracked separately so that they can be included in the simulation statistics.

Listing 6.6 shows an example of the statistics generated at the end of a simulation. The timings below line 6 are sorted in different groups which are indicated by the indentation level of a line. The percentage at the end of a line represents the share of time a measurement took within the respective group. For example: The *Simulate* time consists of the steps *PositionUpdate*, *Boundaries*, *ForceUpdateTotal*, *VelocityUpdate* and *LoadBalancing*, where the load balancing makes 9% of the time of the Simulate group.

```
1  Total number of particles at the end of Simulation: 75741
2  Owned: 16675
3  Halo: 59066
4  Standard Deviation of Homogeneity: 1.47451
5
6  Measurements:
7  Total accumulated            : 17959766231497 ns (17959.766s)
8    Initialization             :       71810561 ns (    0.072s) =  0.000%
9    Simulate                   : 17959654071423 ns (17959.654s) = 99.999%
10     PositionUpdate           :    86561143040 ns (   86.561s) =  0.482%
11     Boundaries:              : 10112193291433 ns (10112.193s) = 56.305%
12       HaloParticleExchange   :  9915536248451 ns ( 9915.536s) = 98.055%
13       MigratingParticleExchange :  196657042982 ns (  196.657s) =  1.945%
14     ForceUpdateTotal         :  6020681077604 ns ( 6020.681s) = 33.523%
15       ForceUpdatePairwise    :  5955739910281 ns ( 5955.740s) = 98.921%
16       ForceUdpateGlobalForces :    64805311767 ns (   64.805s) =  1.076%
17       ForceUpdateTuning      :  3197595651990 ns ( 3197.596s) = 53.110%
18       ForceUpdateNonTuninng  :  2758144258291 ns ( 2758.144s) = 45.811%
19     VelocityUpdate           :    75474193135 ns (   75.474s) =  0.420%
20     LoadBalancing            :  1651394223867 ns ( 1651.394s) =  9.195%
21 One iteration                :      199551711 ns (    0.200s) =  0.001%
22 Total wall-clock time        : 665177619839 ns (665.178s) =  3.704%
23
```

```
24 Tuning iterations                    : 18792 / 90000 = 20.88%
25 MFUPs/sec                            : 0.0175047
26 GFLOPs                               : 322.128
27 GFLOPs/sec                           : 0.0179362
28 Hit rate                             : 0.764462
```

Listing 6.6: Example statistics created at the end of a simulation

Along with the timings, several other statistics are reported at the end of any simulation. When it comes to the evaluation of AutoPas, the most interesting are, aside from the timings, the number of tuning iterations within the simulation and the resulting GFLOPs and GFLOPs/sec.

6.5. Visualizing the Parallel Simulation

While the performance statistics discussed in the previous section are important for the optimization of the simulation, researchers require an actual visualization to gain any meaningful insights. The serial version of MD-Flexible already enabled the user to create recordings of the simulation which then could be visualized using Paraview (see Subsection 3.4.2). When executed with multiple processes, each process would create it's own set of files which then had to be loaded into Paraview one by one, manually. This could become very time consuming if the simulation used a large amount of processes. Unfortunately, the legacy version of the VTK files used by MD-Flexible does not support recordings created by multiple processes. For users to be able to easily visualize their "parallel" data, MD-Flexible now uses modern XML based file formats[4] defined by the Visualization Toolkit. Each of VTK's file formats exists in two variants: One for recording actual data (referred to from here as *data file*), and the parallel file format (referred to from here as *parallel file*) for grouping several data files together so they can be loaded as a single record. Users can then create data files for each rank separately, group those files together using the corresponding parallel file, and load all data files at once by opening this parallel file.

MD-Flexible uses the format for structured grids for the visualization of domains, and, to visualize the particles, the format for unstructured grids. Files of both formats use a specific type: *.vts* for the structured grid and *.vtu* for the unstructured grid. The corresponding parallel files use 'p' as prefix in their file type, i. e. *.pvts* and *.pvtu*. Examples for both file formats can be seen in Figure 6.13 Figure 6.14, Figure 6.16, and Figure 6.17.

6.5.1. Parallel Vtk Writer

The new *Parallel VTK Writer* component is responsible for recording the simulation states. It allows users to configure a prefix (also called *session name*) for the generated records and an output folder for the newly created files. It also provides interfaces to create parallel records for structured and unstructured grid files. These functions are called by the Simulation component during the simulation loop at a specific iteration interval defined by the user.

At the start of the simulation, the host process (rank 0) creates an output folder (if it does not exist yet), generates the parallel files and shares the output location and session

[4] https://kitware.github.io/vtk-examples/site/VTKFileFormats/

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <VTKFile byte_order="LittleEndian" type="PUnstructuredGrid" version="0.1">
3    <PUnstructuredGrid GhostLevel="0">
4      <PPointData>
5        <PDataArray Name="velocities" NumberOfComponents="3" format="ascii"
6          type="Float32"/>
7        <PDataArray Name="forces" NumberOfComponents="3" format="ascii"
8          type="Int32"/>
9        <PDataArray Name="typeIds" NumberOfComponents="1" format="ascii"
10         type="Float32"/>
11       <PDataArray Name="ids" NumberOfComponents="1" format="ascii"
12         type="Float32"/>
13     </PPointData>
14     <PCellData/>
15     <PPoints>
16       <PDataArray Name="points" NumberOfComponents="3" format="ascii"
17         type="Float32"/>
18     </PPoints>
19     <PCells>
20       <PDataArray Name="types" NumberOfComponents="0" format="ascii"
21         type="Float32"/>
22     </PCells>
23     <Piece Source="example_0.vtu"/>
24     <Piece Source="example_1.vtu"/>
25   </PUnstructuredGrid>
26 </VTKFile>
```

Figure 6.13.: Example of the parallel file for the unstructured grid format with two referenced data files.

name with all other processes. During the simulation, each rank creates it's own data files in the target folder.

6.5.2. Creation of particle records

As mentioned before, MD-Flexible uses the VTK file format for unstructured grids to create the recordings of the particle states. The Parallel VTK Writer records velocities, forces, type ids, ids, and positions of the particles. Each of these data requires their own `DataArray` containers in the unstructured grid data file and an equivalent `PDataArray` definition in the parallel file. A single particle's data is then stored as a row within the respective data array, as can be seen in Figure 6.14. VTK considers the records created by different ranks as `Piece`. The parallel file contains a list of pieces which are considered part of a single recorded iteration. This can be seen in line 23 and 24 of Figure 6.13.

6.5.3. Visualization of domains

When it comes to the visualization of the domains, the structured grid file format requires the start and end indices of a domain along each dimension in the subdivision grid. These

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<VTKFile byte_order="LittleEndian" type="UnstructuredGrid" version="0.1">
  <UnstructuredGrid>
    <Piece NumberOfCells="0" NumberOfPoints="4">
      <PointData>
        <DataArray Name="velocities" NumberOfComponents="3" format="ascii"
          type="Float32">
        0 0 0
        0 0 1
        0 1 0
        1 0 0
        </DataArray>
        <DataArray Name="forces" NumberOfComponents="3" format="ascii"
          type="Float32">
        0 0 0
        0 0 0
        0 0 0
        0 0 0
        </DataArray>
        <DataArray Name="typeIds" NumberOfComponents="1" format="ascii"
          type="Float32">
        0
        0
        0
        0
        </DataArray>
        <DataArray Name="ids" NumberOfComponents="1" format="ascii"
          type="Float32">
        0
        1
        2
        3
        </DataArray>
      </PointData>
      <CellData/>
      <Points>
        <DataArray Name="positions" NumberOfComponents="3" format="ascii"
          type="Float32">
        0 0 0
        0 0 1
        0 1 0
        1 0 0
        </DataArray>
      </Points>
      <Cells>
        <DataArray Name="types" NumberOfComponents="0" format="ascii"
          type="Float32"/>
      </Cells>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

Figure 6.14.: Example of a data file for the unstructured grid format containing the data of four particles.

indices are called the extent of a domain. A parallel structured grid file contains the extent of the global domain and the extents of all subdomains. The global extent or `WholeExtent` is defined in line 3 of the Figure 6.16 and the extents of the two subdomains in lines 28 and 29. The data files also contain the extent of single subdomain as can be seen in line three and four in Figure 6.17. The extent of a domain can be computed using the function `getExtendOfSubdomain()` (see Listing 6.7) provided by the newly implemented *Domain Tools* library. The function calculates the extent based on the respective process's rank which can directly be mapped to the subdomain's position in the decomposition grid using the `convertIndexToId()` function which takes a domainIndex or rank and converts it to the corresponding coordinates in the subdivision grid.

Along with the extent, the structured grid files need to contain the eight vertices of each referenced subdomain while the parallel file contains the corner vertices of the global domain. These vertices can easily be derived from the minimum and maximum values of the respective domain.

Additionally, the parallel file needs to provide the vertices of the global domain's corners, while the data files provide the vertices of their respective subdomains. An example of a domain visualization is illustrated in Figure 6.15.

```
1  std::array<int, 6> getExtentOfSubdomain(const int subdomainIndex, const std::
       array<int, 3> decomposition) {
2    std::array<int, 6> extentOfSubdomain{};
3
4    const std::array<int, 3> subdomainId = convertIndexToId(subdomainIndex,
         decomposition);
5
6    for (size_t i = 0; i < 3; ++i) {
7      extentOfSubdomain[2 * i] = subdomainId[i];
8      extentOfSubdomain[2 * i + 1] = subdomainId[i] + 1;
9    }
10
11   return extentOfSubdomain;
12 }
```

Listing 6.7: Implementation of the getExtendOfSubdomain function.

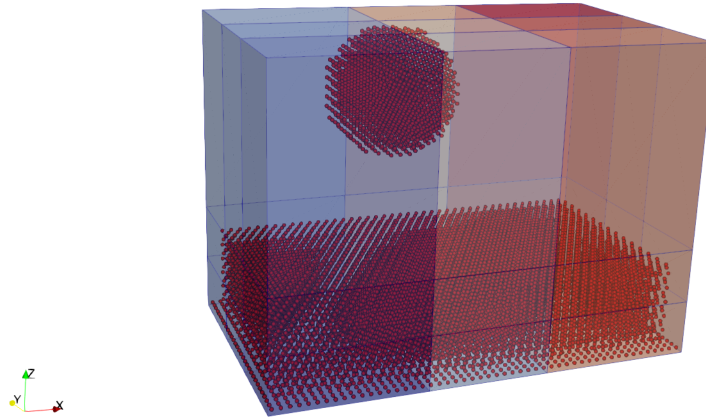


Figure 6.15.: A visualization of the subdivision generated for the falling drop scenario shortly after the start of the simulation using 27 ranks.

6.5.4. Additional information about domains and particles

Both the structured and unstructured grid files can contain additional information about the particles or domains. The information again needs to be declared in the respective parallel file along with the actual data stored in the data file.

The structured grid records the domain id, rank, and data about the state of the AutoPas container responsible for the simulation of the subdomain. Most of the data is stored as integer values corresponding to an enumeration defined by AutoPas. For instance, the cell property *DataLayout* recorded in line 15 to 17 of Figure 6.17 stores as value 0 which refers to the "Array of Structs" layout. Unfortunately, users have to look into the source AutoPas' source code to retrieve the meaning behind the enumeration values. Using string values to store the data either stopped Paraview from loading the recordings or the Paraview's animation feature would cause the program to freeze.

The particle positions, the velocities, forces types, and ids of the particles are recorded by the unstructured grid format. As in the structured grid format, the data is stored in `DataArray`'s as can be seen in lines 6-33 of Figure 6.14.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <VTKFile byte_order="LittleEndian" type="PStructuredGrid" version="0.1">
3    <PStructuredGrid WholeExtent="0 1 0 1 0 2" GhostLevel="0">
4      <PPointData/>
5      <PCellData>
6        <PDataArray type="Int32" Name="DomainId" />
7        <PDataArray type="Float32" Name="CellSizeFactor" />
8        <PDataArray type="Int32" Name="Container" />
9        <PDataArray type="Int32" Name="DataLayout" />
10       <PDataArray type="Int32" Name="FullConfiguration" />
11       <PDataArray type="Int32" Name="LoadEstimator" />
12       <PDataArray type="Int32" Name="Traversal" />
13       <PDataArray type="Int32" Name="Newton3" />
14       <PDataArray type="Int32" Name="Rank" />
15     </PCellData>
16     <PPoints>
17       <DataArray NumberOfComponents="3" format="ascii" type="Float32">
18         -0.5 -0.5 -0.5
19         50.5 -0.5 -0.5
20         -0.5 30.5 -0.5
21         50.5 30.5 -0.5
22         -0.5 -0.5 37.296
23         50.5 -0.5 37.296
24         -0.5 30.5 37.296
25         50.5 30.5 37.296
26       </DataArray>
27     </PPoints>
28     <Piece Extent="0 1 0 1 0 1" Source="example_0.vts"/>
29     <Piece Extent="0 1 0 1 1 2" Source="example_1.vts"/>
30   </PStructuredGrid>
31 </VTKFile>
```

Figure 6.16.: Example of the parallel file for the structured grid format with two referenced data files.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2  <VTKFile byte_order="LittleEndian" type="StructuredGrid" version="0.1">
3    <StructuredGrid WholeExtent="0 1 0 1 1 2">
4      <Piece Extent="0 1 0 1 1 2">
5        <CellData>
6          <DataArray type="Int32" Name="DomainId" format="ascii">
7            1
8          </DataArray>
9          <DataArray type="Float32" Name="CellSizeFactor" format="ascii">
10           1
11         </DataArray>
12         <DataArray type="Int32" Name="Container" format="ascii">
13           6
14         </DataArray>
15         <DataArray type="Int32" Name="DataLayout" format="ascii">
16           0
17         </DataArray>
18         <DataArray type="Int32" Name="LoadEstimator" format="ascii">
19           0
20         </DataArray>
21         <DataArray type="Int32" Name="Traversal" format="ascii">
22           20
23         </DataArray>
24         <DataArray type="Int32" Name="Newton3" format="ascii">
25           0
26         </DataArray>
27         <DataArray type="Int32" Name="Rank" format="ascii">
28           1
29         </DataArray>
30       </CellData>
31       <Points>
32         <DataArray type="Float32" NumberOfComponents="3" format="ascii">
33           -0.5 -0.5 18.398
34           50.5 -0.5 18.398
35           -0.5 30.5 18.398
36           50.5 30.5 18.398
37           -0.5 -0.5 37.296
38           50.5 -0.5 37.296
39           -0.5 30.5 37.296
40           50.5 30.5 37.296
41         </DataArray>
42       </Points>
43     </Piece>
44   </StructuredGrid>
45 </VTKFile>
```


Figure 6.17.: Example of a data file for the structured grid format.

Part III.

Evaluation

7. Evaluation

Now that massive parallelism is implemented in MD-Flexible, it is time to evaluate it's effectiveness. For this, we will have a look at speedup and parallel efficiency for the three scenarios Falling Drop, Explosive Liquid, and Spinodal Decomposition described in Section 3.2. Afterwards we will compare ALL's Tensor load balancer to our custom Inverted Pressure load balancer.

All of the tests have been run on the CoolMUC-2 cluster segment of the LRZ Linux cluster. Table 7.1 shows the hardware specification of the cluster.

CPU Vendor	CPU	CPUs per node	Threads per CPU	Frequency (turbo)	Memory (DDR4) per node
Intel	Xeon E5-2697 v3	2	14×2	2.6 (3.6) GHz	64 GB (Bandwidth 120 GB/s - STREAM)

Table 7.1.: CoolMUC2 Hardware specification[a]

[a] https://doku.lrz.de/display/PUBLIC/CoolMUC-2

7.1. Speedup And Efficiency

7.1.1. Setup

To test the speedup and the efficiency, the scenarios have been simulated using both implemented load balancers, each with eight test runs. For each test run, the number of processes has been doubled, starting with only a single process, with the eighth test using 128 processes.

For the tests of the Falling Drop scenario, MD-Flexible's default configuration has been used where the size of grid block used to simulate the floor is $51 \times 31 \times 1$, the grid block representing the body of liquid is $48 \times 28 \times 10$ large and the sphere representing the drop has radius 6. The AutoPas container rebuilds the verlet skin every 10 iterations with a skin radius of 0.3 and a cutoff radius of 3. The simulation contains 16675 particles and runs for 90000 iterations.

For the Exploding Liquid scenario a simulation domain size of $60 \times 180 \times 60$ and a closest packed block of size $40 \times 30 \times 40$ has been chosen with the simulation running for 50000 iteration containing 68600 particles. The verlet skin radius is 0.2 with a cutoff radius of 2 and a verlet rebuild frequency of 2.

To create the record used for the Spinodal Decomposition scenario, a grid block of size $80 \times 80 \times 80$ has been simulated for 100000 iterations to create an equilibrium state which

then has been loaded into a simulation domain of 120 × 120 × 120 to simulate the Spinodal decomposition. The cutoff radius is 2.5, the verlet skin radius is set to MD-Flexibles default value of 0.2, and the verlet skin rebuild frequency is 10. Being the largest of the three scenarios, Spinodal Decomposition simulates 511806 particles.

7.1.2. Results

Before getting into the reasons for the performance, it is important to note that the scenario size used for the Falling Drop scenario was selected too small to run with 128 ranks because both load balancing methods employ a minimum box size. The small size has been selected to reduce the runtime of the scenarios on few processes and is the reason why there are data points missing in the diagrams at 128 processes for this scenario. In addition, the Inverted Pressure load balancer breaks the simulation of the Falling Drop scenario when running with 64 processes. The Exploding Liquid scenario breaks starting at 128 ranks. This is why for this load balancer there are again data points missing at 64 ranks. The reason will be investigated during the detailed discussions of the scenarios in Section 7.2. The Spinodal Decomposition scenario finished successfully using Inverted Pressure on 64 and 128 ranks.

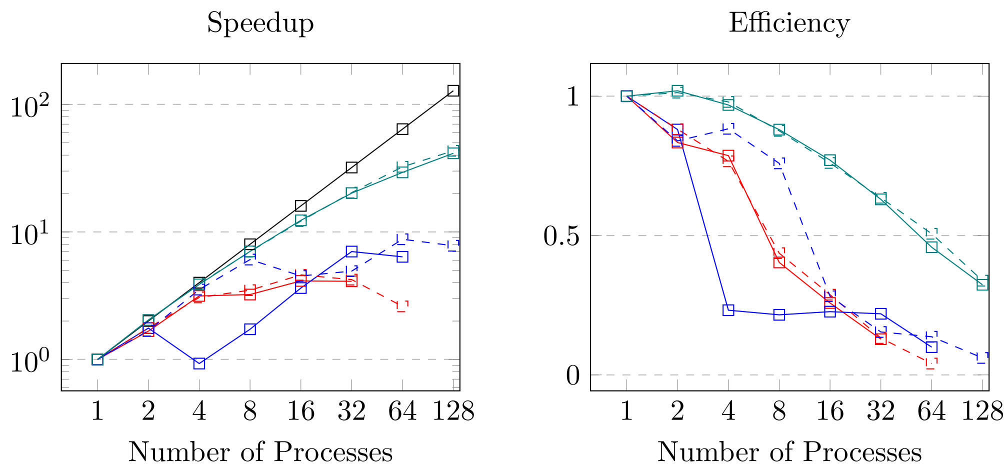


Figure 7.1.: Speedup and Efficiency of the parallelization for MD-Flexible's three scenarios. Legend: Falling Drop, Exploding Liquid, Spinodal Decomposition, Ideal Speedup. A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used.

The speedup and the efficiency of the parallelization are illustrated in Figure 7.1. As is clearly visible, the current parallelization is suboptimal. With a speedup of approximately 40 with 128 processors, looking at the ALL load balanced run of the Spinodal Decomposition scenario represented by dotted teal line. We do not take nearly full advantage of the additional processing power. The scenarios Falling Drop and Exploding Liquid performed even worse with a speedup of under 10 on 64 processes using ALL's Tensor load balancer. Only the Spinodal Decomposition does not display a large drop in efficiency starting at 4 processes but drops increasingly fast the more ranks are employed.

The low performance is caused by several problems. Some of them are specific to a certain

scenario and will be explained in Section 7.2. The main factor which affects all scenarios is the rebalancing of the decomposition in every iteration. Normally, the subdomains should be rebalanced only at a specific interval, for example every 100 iterations. The resulting effect is clearly visible in Figure 7.2. For most of the tests, the percentage taken by the load balancing compared to the total time stays constant up to 16 processes. From 32 ranks on, the percentage increases drastically for each scenario. The increasing number of subdivisions along each coordinate axis slows down the load balancing because more and more process grid planes are involved. Additionally, the number of particles which need to be communicated increases drastically. For instance, the Exploding liquid scenario simulates 68600 particles but requires 77450 halo particles on 64 ranks.

Another aspect influencing overall performance are the `MPI_Allreduce()` calls at several locations during an iteration which explicitly synchronizes all processes and, therefore, increases wait times. This is another reason for the steep increase visible in Figure 7.2. With an increasing number of processes, their synchronization takes longer, resulting in even longer wait times. An alternative for `MPI_Allreduce()` would be a fan-in and fan-out communication discussed in Part IV.

A last factor influencing the perfromance is the fact, that for the test runs with more than 16 processes, 16 tasks have been allocated for every compute node, leaving almost no taks for thread-level parallelism.

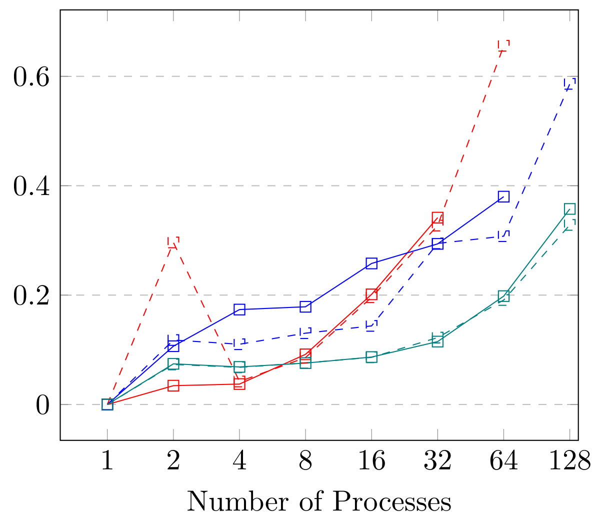


Figure 7.2.: Percentage of load balancing time compared to total execution time. Legend: Falling Drop, Exploding Liquid, Spinodal Decomposition. A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used.

7.2. Detailed Discussion of the Scenarios Performances

With the Spinodal Decomposition displaying the best performance, we will first have a look at this scenario, before investigating the bad performances of the Falling Drop and

Exploding Liquid scenarios.

7.2.1. Performance of Spinodal Decomposition Scenario

The Spinodal Decomposition scenario has by far the biggest domain size and simulates the most particles. So why does it have the best performance out of all the three scenarios?

Compared to the other scenarios, the Spinodal Decomposition has the lowest particle density gradient throughout the simulation domain. This means, that the particles are spread out more homogeneously compared to the Falling Drop and the Exploding Liquid scenario which minimizes the effect of the bad local refinement employed by a regular grid decomposition as mentioned in Subsection 2.2.1. The Spinodal Decomposition implicitly ensures that every process has a similar amount of work. Looking at Figure 7.3, it is clearly visible that all of the subdomains at the top of the simulation domain do not contain any particles leaving the respective process without work. This is caused by a minimum cell size required by both the ALL and the Inverted Pressure Load balancer. The fact that the scenarios sizes of Falling Drop and Explosive Liquid have been configured too small also plays into their bad performance.

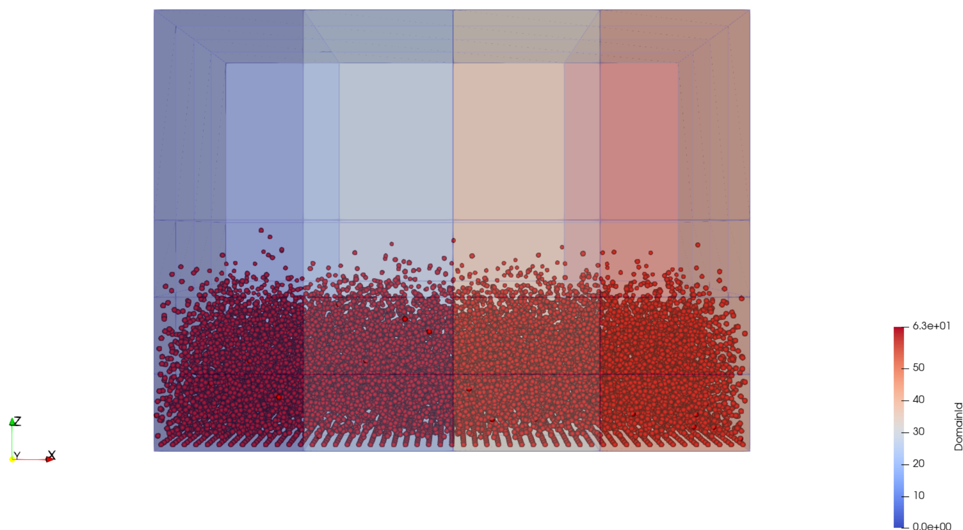


Figure 7.3.: Bad Local Refinement in the Falling Drop Scenario. Many of the subdomains do not contain any particles.

7.2.2. Performance of Falling Drop Scenario

While the Falling Drop scenario displays a steep decrease in efficiency with increasing number of ranks, the efficiency curve only starts to drop drastically starting with 8 ranks and afterwards shows a slightly more even slope. This is, again, because of a rapidly changing density gradient along the z-dimension (up) and the bad local refinement of the regular grid decomposition. With 8 processes, the z-coordinate axis is subdivided the first time resulting in some subdomains having very few to no particles.

As mentioned before, the small scenario size plays into the bad performance as well. On 64 ranks, the simulation requires 77221 halo particles compared to the 16675 ”real” particles. These halo particles have to be updated in every iteration inducing a massive amount of communication. On top of that when using more than 16 processes, the minimum cell size employed by the load balancing causes processes to be idle, as mentioned in Subsection 7.2.1.

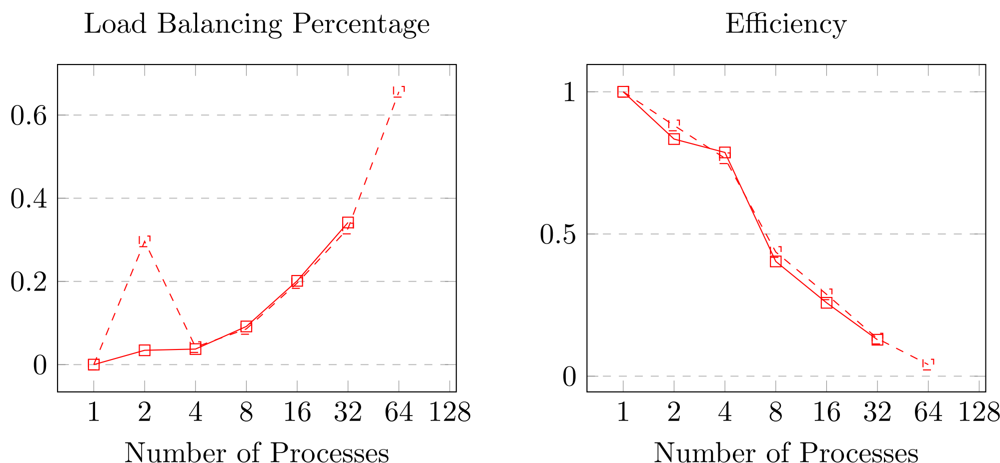


Figure 7.4.: Load Balancing percentage and Efficiency in the Falling Drop Scenario. A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL’s Tensor method has been used.

7.2.3. Performance of Exploding Liquid Scenario

Looking at the graphs in Figure 7.5, the Exploding Liquid scenario behaves similarly to the Falling Drop scenario. While the latter reached 60% load balancing in overall execution time at 64 ranks, the Exploding Liquid scenario reaches the same percentage at 128 ranks. And although the efficiency stays constant between 4 and 32 to ranks, it displays a huge drop at 4 ranks with the Inverted Pressure method. Here, the load balancer displays an oscillating behavior even though the domains are balanced in the initially generated decomposition. The two states of this oscillation are illustrated in Figure 7.6. These oscillations also have been observed in the Falling Drop scenario and can occur if the particles are very clumped up. Looking at the first image in Figure 7.6 the two domains on the right side do all the work. Therefore the algorithm shifts the left boundary of those domains a large distance to the right, basically jumping over the particle cloud instead of splitting it. If the particles are this close this generates the decomposition which mirrors the previous one, as can be seen in the second image of the Figure. This oscillation continues until the particles migrate beyond one of the ”oscillation boundaries”. As soon as more processors are involved, such large oscillations become impossible because the subdomain size, along with the difference in the performed work within the subdomains, decreases. Therefore, the efficiency remains constant up to 64 processes. This oscillation behavior can be fixed, as we will learn in Subsection 9.1.2.

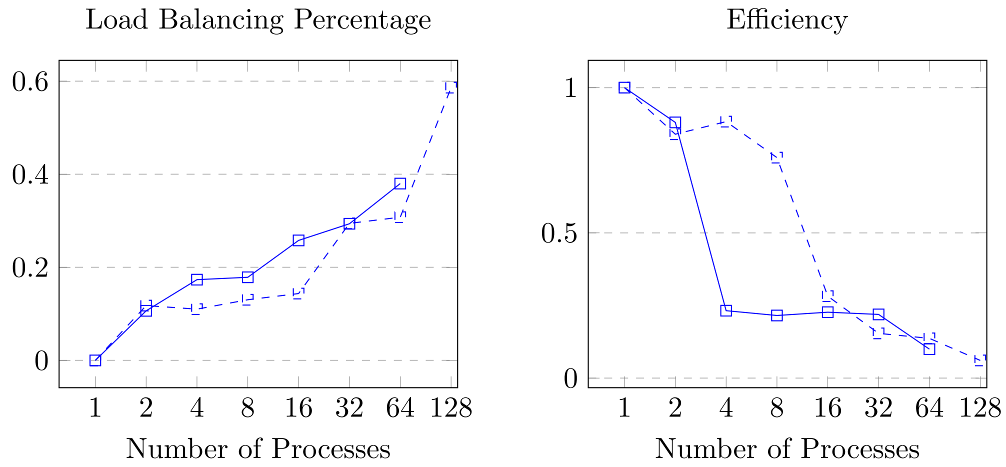


Figure 7.5.: Load Balancing Percentage and Efficiency in the Exploding Liquid Scenario. A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used.

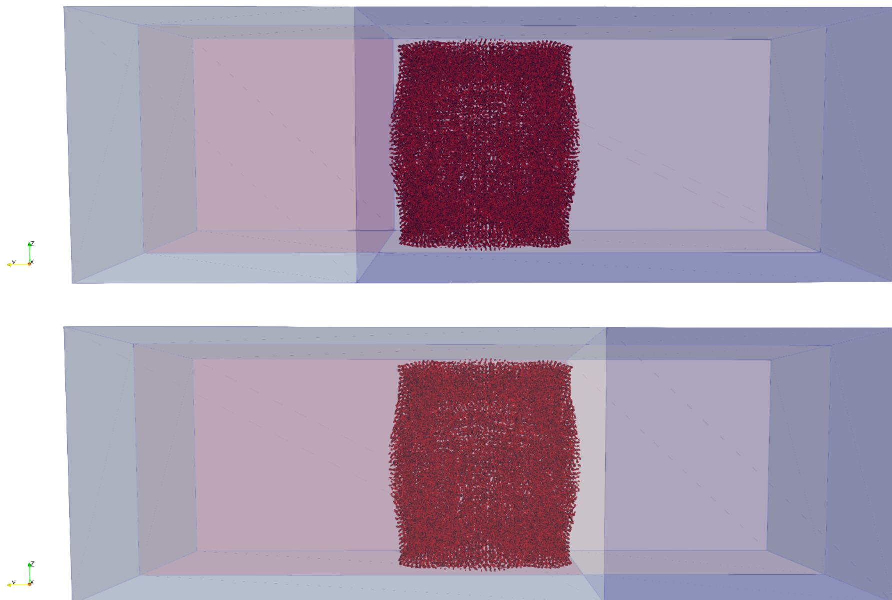


Figure 7.6.: Oscillation pattern in the early Exploding Liquid scenario. The decomposition oscillates between the two configurations until particles have traveled far enough to the left/right for the decomposition to become more stable.

7.3. Comparing ALL's Tensor Method the Inverted Pressure Method

The previous graphs suggest that ALL's Tensor method performs slightly better compared to the Inverted Pressure method. But this advantage does not result in a better overall execution time as can be seen in Figure 7.7. In Subsection 7.2.3 we observed that the Inverted Pressure method generates larger oscillations between boundary positions. Consequently, more particles need to be migrated after each load balancing, compared to the Tensor method. Looking at Figure 7.8, this seems to be true when a small amount of processes is involved. When using a larger amount of processes, the Inverted Pressure method gains stability and requires less time for the migration of particles than the Tensor method. Looking at Figure 7.9, the Tensor method performs slightly better again. In general, a good decomposition requires less halo particles to be exchanged. While at many of the data points the Tensor and the Inverted Pressure method perform equally, the former displays an overall better performance. Only in the Exploding Liquid scenario can we find large differences in the communication times. This again is caused by the oscillating behavior described in Subsection 7.2.3. Although only two domains need to exchange particles, they have to exchange a large amount consequently investing more time in communication.

In the end, both load balancing approaches show similar performance. Nonetheless ALL's Tensor method is more reliable and displays less oscillations. As we will learn in Section 9.1, the performance of the Inverted Pressure can be improved. But until the suggested features are implemented, the Tensor Method is the better choice when conducting experiments.

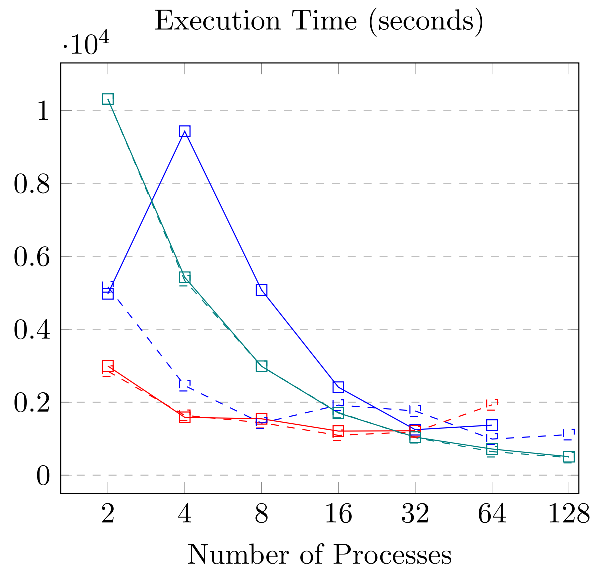


Figure 7.7.: The total execution times of the scenarios using ALL's Tensor method and Inverted Pressure. Legend: Falling Drop, Exploding Liquid, Spinodal Decomposition. A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used. The execution time on a single rank has been omitted in the graph to enhance the resoution at the other data points.

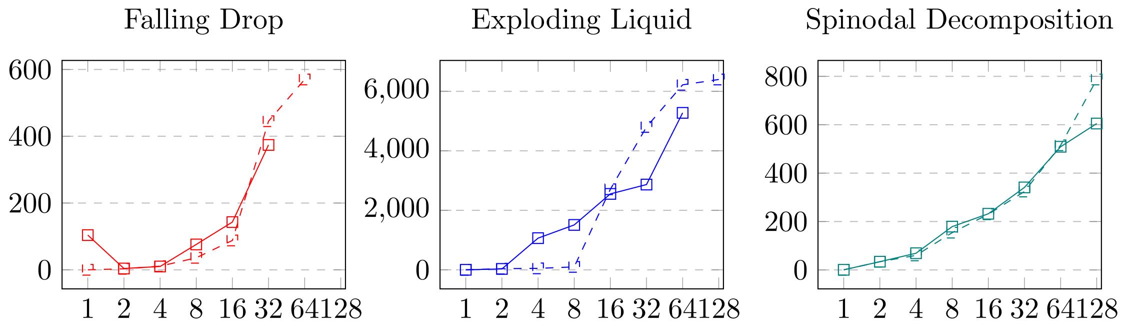


Figure 7.8.: The times used for particle migration using ALL's Tensor method and Inverted Pressure. Legend: A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used.

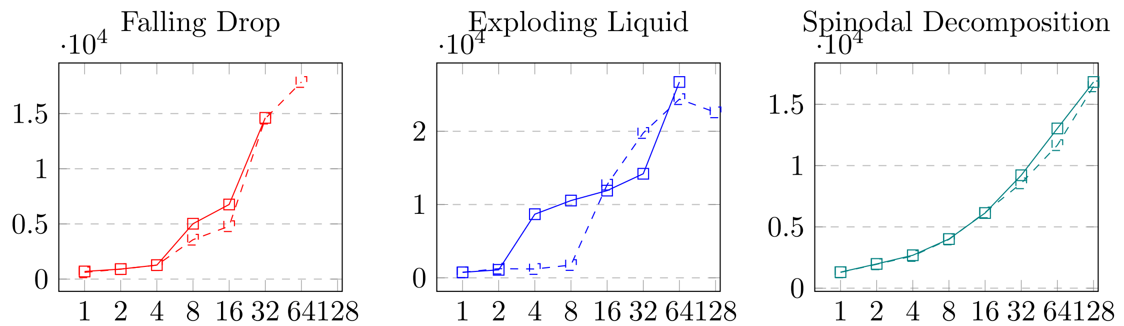


Figure 7.9.: The times used for the exchange of halo particles using ALL's Tensor method and Inverted Pressure. Legend: A continuous line indicates that the Inverted Pressure load balancing method has been used and a dashed line that ALL's Tensor method has been used.

Part IV.

Future Work

8. General Performance Improvements

8.1. Configurable Load Balancing Interval

As mentioned during Chapter 7, the load balancing during every iteration is one of the principle causes of the bad performance. This can be fixed rather easily by including an additional *load-balancing-interval* configuration parameter. Users can then define an iteration interval, when the load balancing should take place. Considering that in some cases, load balancing might not be required at all this variable should also allow users to turn it off completely, for example, by setting the interval to zero. This is useful in the Falling Drop scenario if the domain decomposition is restricted to the horizontal coordinate axis. As mentioned at the end of Section 3.2, the gravity employed in the Falling Drop scenario ensures an implicit load balancing if only the horizontal coordinate axis has been subdivided.

It might even be useful to adaptively adjust the load balancing interval over the runtime of a simulation. The Exploding Liquid scenario displays a rapid change of the particle layout during the early phase which might require a higher load balancing frequency compared to later phases of the simulation.

8.2. Soften the Minimum Cell Size Restriction

Another aspect which limits the efficiency of the load balancing is the minimum cell size defined for both ALL's Tensor and the Inverted Pressure method which is currently restricted by the cutoff and skin radius. Therefore, the load balancers are not able to properly distribute the work of areas in the simulation domain with a high density of particles. Allowing a subdomain to have a side length smaller than the cutoff radius does require processes to communicate beyond their direct neighbors. This can easily be implemented with an additional loop during the step-wise communication which iterates over all neighbors along a direction. The additional communication might outweigh the advantage but it is definitely worth investigating on properly configured test scenarios.

9. Improving the Adaptive Load Balancing

9.1. Improving Inverted Pressure

The Inverted Pressure method slightly underperforms compared to ALL's Tensor method, omitting the fact, that it seems to be very unstable above 32 ranks.

9.1.1. Global Balancing along a coordinate axis

Currently both the Tensor and the Inverted Pressure method do not globally balance the subdomains along a coordinate axis. The Inverted Pressure method can be extended to include the global balancing.

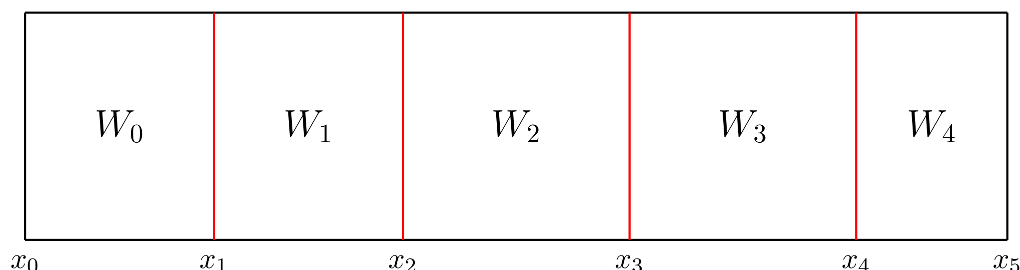


Figure 9.1.: Global load balancing scenario for Inverted Pressure along a single coordinate axis.

As can be seen in Figure 9.1, we now want to use all the weights along a coordinate axis to calculate a global load balancing using the inverted pressure method. The function (1) derived in Subsection 6.1.3 only takes the weights of those domains into account which lie adjacent to the boundary we want to shift. Let's first rewrite the function for an arbitrary boundary x_n.

$$x'_n = (W_n x_{n+1} + W_{n-1} x_{n-1}) \ \frac{1}{W_{n-1} + W_n} \tag{2}$$

Now let the maximum number of boundaries, including the global domain boundaries, be B. This leaves us with the corresponding work W_n of the nth subdomain, and shiftable boundaries at positions x_1 to x_{B-2} (see Figure 9.1). To consider the global work we take the weights of all subdomains to the left of the current shiftable boundary and the weights of all subdomains to the right, instead of just the weight of the adjacent domains. Taking this into account, equation (2) will look like this:

$$x'_n = ((x_n - x_0) \sum_{i=0}^{n-1} W_i + (x_{B-1} - x_n) \sum_{i=n}^{B-2} W_i) \ \frac{1}{\sum_{i=0}^{B-2} W_i}$$

From now on we will omit the fraction at the end of the previous term because it does not depend on n. If we factorize the function with respect to the x-values we get following result:

$$x'_n = x_0 \sum_{i=0}^{n-1} W_i + x_n(\sum_{i=0}^{n-1} W_i - \sum_{i=n}^{B-2} W_i) + x_B \sum_{i=n}^{B-2} W_i$$

Setting $a_n = \sum_{i=0}^{n-1} W_i$ and $b_n = \sum_{i=n}^{B-2} W_i$ yields the following function:

$$x'_n = a_n x_0 + (a_n - b_n)x_n + b_n x_B$$

Using Figure 9.1 as setup, B is equal to 6. For this specific amount of subdivisions, when writing the previous function as matrix vector multiplication, and including the constant term we omitted earlier, we get following system of equations:

$$\frac{1}{\sum_{i=0}^{B-2} W_i} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ a_1 & (a_1 - b_1) & 0 & 0 & 0 & b_1 \\ a_2 & 0 & (a_2 - b_2) & 0 & 0 & b_2 \\ a_3 & 0 & 0 & (a_3 - b_3) & 0 & b_3 \\ a_4 & 0 & 0 & 0 & (a_4 - b_4) & b_4 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ x'_5 \end{pmatrix}$$

This can easily be parallelized using techniques from sparse linear algebra. Along with the communication improvements suggested in Section 9.2 this does not only improve the precision but also the speed of the Inverted Pressure load balancer.

9.1.2. Improving Stability

In contrast to the Inverted Pressure method, ALL's Tensor method employs a damping factor $\frac{1}{\gamma}$ to improve the stability. A similar approach can be implemented in the Inverted Pressure algorithm.

Instead of using a damping factor, though, I would suggest remembering the last 4 positions of a domain boundary and, using this histogram, to calculate a weighted average of the new and the old positions. The weights should decrease with the "age" of a value in the histogram. This will strongly increase the stability of the method, if the weights have been chosen wisely. As a drawback, the algorithm is not able to quickly adapt to a new topology, anymore. This might be negligible, though, because rapid topology changes are very rare in most MD scenarios.

It should be enough to store the last 4 positions, as the weights should decrease exponentially with the age of a weight. For this reason, further values may not be necessary as they would have a diminishing influence on the result.

9.2. Increasing parallelism

As mentioned in Subsection 7.1.2, another way to improve performance is to reduce the number of explicit synchronization points mostly imposed by the use of `MPI_Allreduce()`.

During the load balancing with Inverted Pressure, all ranks within a planar communicator compute the same function to determine a new boundary. Instead of using `MPI_Allreduce()`, each rank should send the required data to the root rank within the respective communicator which is considered a fan-in operation. The root rank then calculates the new boundary and sends the result back to the other ranks using the fan-out communication pattern. This can be achieved only using point-to-point communication methods, allowing processes to do different work. For example, the balancing needs to be done six times (2 times per dimension) for each planar communicator which can be performed in parallel by processes with rank 0 to 5, if there are this much processes available within the communicator.

Another explicit synchronization point induced by `MPI_Waitall()` lies at the end of the particle migration and the halo particle exchange. It has been introduced to make sure that the send buffers are cleared after their message reached the receiver. At this point, the process waits for all pending send requests to finish. Instead of waiting, send buffers should be cleared as soon as the requests were successful which can be determined using the `MPI_Status`. In this case, the requests need to be checked perpetually which can be achieved using an independent worker thread.

10. Other Improvements and possible Features

10.1. Thermostat and Homogeneity

Two aspects of MD-Flexible have not been parallelized properly, so far. Both the thermostat and the homogeneity require a calculation of a global average over all subdomains. Currently, this is implemented naively using `MPI_Allreduce()` before dividing the reduced sum by the number of subdomains. This approach does introduce an error into the calculation of the global temperature and homogeneity as it does not consider the specific sizes of the subdomains. The local values need to be weighted according to their respective subdomain size. A small subdomain should have less impact on the global average as a large subdomain.

10.2. Extended Subdivision Constraint

As mentioned in Subsection 6.1.1, MD-Flexible allows users to restrict the subdivision of the simulation domain to specific coordinate axis. This can be extended by restricting the maximum number of cuts along an axis, allowing even more fine tuning. The simulation domain of the Exploding Liquid scenario has a size of $60\times180\times60$. So the y-axis is three times as large than the x- and z-axis. To reduce the surface area of each subdomain, it is necessary to limit the amount of subdivisions of the x- and z-axis so that most of the cuts are applied on the y-axis. Minimizing the surface area of the subdomains is key to reducing the number of halo particles in the simulation. This will reduce the amount of force interactions calculated between halo particles and owned particles while also decreasing the communication effort.

10.3. Proper Testing

In view of the bad configuration used for the Falling Drop and Exploding Liquid tests another thorough test session is required. When it comes to strong scaling, the size of the tests needs to be increased to support parallelization with 128 and more processes. A suitable subdivision configuration for the Exploding Liquid scenario has to be employed to minimize maximum surface area of the subdomains. On top of that, a series of weak scaling tests have to be designed to get a thorough impression of the weak scaling behavior of both load balancing approaches.

Part V.

Summary

Before the integration of massive parallelism into MD-Flexible is discussed in detail, we summarized the most important aspects of Molecular Dynamics and adaptive domain decomposition. Here we talk about the N-Body problem and the Linked Cells and the Verlet Lists methods used to reduce the computational complexity of MD simulations. Further we discuss two approaches to domain decomposition: Grid Decomposition and Octree Decomposition. To handle inter domain forces, which are imposed by the domain subdivision, we investigate the halo particle approach and the eighth shell method. Afterwards we have a short look at similar projects, namely GROMAKS, LAMMPS, and ls1-mardyn, and provide a short overview of AutoPas and introduce its demonstrator MD-Flexible.

The domain decomposition employs a regular grid decomposition which is generated using prime factorization. Users are able to restrict the subdivision along specific coordinate axis. The inter process communication has been realized using MPI and a step wise communication scheme to reduce the total number of messages. Particle data is serialized into `char` arrays before they are send to another process which then, once it received the package, deserializes them. Using the C++ fold expressions, the serialization and deserialization is mostly generic and requires low maintenance effort when particle attributes are introduced or removed.

We did not only implement the external load balancer ALL but also derived the custom load balancing algorithm called the Inverted Pressure method. We describe the idea on which the custom method is based and derive the equation which is then used for the load balancing. The method displays similar efficiency compared to ALL's Tensor method, is rather easy to understand and can be improved in multiple directions. In addition, we did lay the foundation for the integration of other decomposition approaches like the staggered grid or the octree decomposition.

With the help of VTK and Paraview, the massive parallel simulations can be visualized properly and the XML file formats used for the visualization can be extended easily.

The implementation has been tested on three simulation scenarios: Falling Drop, Exploding Liquid and Spinodal Decomposition. They have been executed on the CoolMUC2 supercomputer standing at the Leibniz Rechenzentrum. During the evaluation of the generated data, several points of potential improvement of the currently implemented load balancing have been discovered along with some stability issues of the Inverted Pressure load balancer. Afterwards we provide solutions to some of those shortcomings and derive an equation which allows the Inverted Pressure method to balance the subdomains globally along a coordinate axis.

Although we achieved the goal of integration massive parallelism with the help of adaptive domain decomposition into MD-Flexible, the result does does lack in efficiency. This is not only a result of the drawbacks of the regular grid which has been chosen as decomposition scheme, but also because of other issues damping the performance of the parallelization. In particular, load balancing the domains on each iteration needs to be removed so that MD-Flexible is a proper demonstrator for AutoPas and it's capabilities with massive parallelism. Nonetheless, this issue is easily fixed and will not dampen the performance of MD-Flexible for long.

Part VI.

Appendix

List of Figures

2.1. Direct Sum on constructed Exploding Liquid Scenario 5
2.2. Linked Cells on constructed Exploding Liquid Scenario 5
2.3. Verlet Lists on constructed Exploding Liquid scenario 5
2.4. A grid decomposition of the Exploding Liquid scenario 7
2.5. Quadtree decomposition on constructed Exploding Liquid scenario 7
2.6. Fourth and eighth shell method 9

3.1. Initial state fo the Falling Drop scenario 12
3.2. Final state of the Spinodal Decomposition scenario 12
3.3. Intermediate state of the Exploding Liquid scenario 13
3.4. Final state of the Exploding Liquid scenario 13

5.1. Old initialization of the Simulation component 19
5.2. Old initialization of the Simulation component 20
5.3. Shifting of migrating particles 20
5.4. MD-Flexible's old simulation loop 21
5.5. Halo Particle Creation 21
5.6. Overview of the new MD-Flexible application 22
5.7. New Initialization of the Simulation component 23
5.8. New Simulation Loop 24

6.1. Algorithm to create regular grid decomposition 27
6.2. Planar Communicators in a 3-by-3 grid decomposition 28
6.3. Algorithm to determin box coordinates 29
6.4. Conversion from domain id to domain index 29
6.5. Algorithm to determine the current process's domain neighbors 30
6.6. Base scenario of Inverted Pressure Method 32
6.7. Inverted Pressure - First half 34
6.8. Inverted Pressure - Second half 35
6.9. Step-wise communication pattern in 3D 38
6.10. Exchange of migrating particles using step-wise communication 39
6.11. Sorting of particle for left and right neighbor 40
6.12. Send and receive particles to the left and right neighbour 41
6.13. Parallel Unstructured Grid File 45
6.14. Data file of the unstructured grid format 46
6.15. Falling Drop scenario on 27 ranks 48
6.16. Parallel file of the structured grid format 49
6.17. Data File of the structured grid format 50

7.1. Speedup and Efficiency of MD-Flexible's scenarios 53
7.2. Percentage of load balancing time compared to total time 54
7.3. Bad Local Refinement in the Falling Drop Scenario 55
7.4. Load Balancing percentage and Efficiency in the Falling Drop Scenario . . . 56
7.5. Load Balancing Percentage and Efficiency in the Exploding Liquid Scenario 57
7.6. Oscillation pattern in the Exploding Liquid scenario 57
7.7. The total execution times of the scenarios using ALL's Tensor method and Inverted Pressure 58
7.8. The times used for particle migration using ALL's Tensor method and Inverted Pressure 59
7.9. The times used for the exchange of halo particles using ALL's Tensor method and Inverted Pressure 59

9.1. Global Inverted Pressure scenario 62

List of Tables

7.1. CoolMUC2 Hardware specification 52

Bibliography

[1] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *Elsevier*, 2015.

[2] M. P. Allen. Introduction to molecular dynamics simulation. Technical report, Centre for Scientific Computing and Department of Physics, University of Warwick, 2004.

[3] H. Berendsen, D. van der Spoel, and R. van Drunen. Gromacs: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 1995.

[4] W. M. Brown, P. Wang, and S. J. P. ans A. N. Tharrington. Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications*, 2011.

[5] S. Günther, L. Ruthotto, J. Schroder, E. C. Cyr, and N. Gauger. Layer-parallel training of deep residual neural networks. Technical report, TU Kaiserslautern, 2018.

[6] R. Halver, S. Schulz, and G. Sutmann. All - a loadbalancing library, c++ / fortran library. `https://gitlab.version.fz-juelich.de/SLMS/loadbalancing/-/releases`. accessed on 2021-13-10.

[7] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient load-balanced, and scalable molecular simulation. *Journal of Chemical Theory and Computation*, 2008.

[8] E. H. Müller and R. Scheichl. Massively parallel solvers for elliptic partial differential equations in numerical weather and climate prediction. Technical report, Put institution here, 2014.

[9] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and M. Horsch. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation*, 2014.

[10] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 1995.

[11] N. Tchipev, S. Seckler, M. Heinen, J. Vrabec, F. Gratl, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, N. Hammer, B. Krischok, M. Resch, D. Kranzlmüller, H. Hasse, H.-J. Bungartz, and P. Neumann. Twetris: Twenty trillion-atom simulation. *The International Journal of High Performance Computing Applications*, 2019.

[12] S. A. Wirp, A.-A. Gabriel, M. Schmeller, E. H. Madden, I. van Zelst, L. Krenz, Y. van Dinther, and L. Rannabauer. 3d linked subduction, dynamic rupture, tsunami, and inundation modeling: Dynamic effects of supershear and tsunami earthquakes, hypocenter location, and shallow fault slip. *Frontiers in Earth Science*, Jun 2021.

[13] W. Xiao, A. Sabne, P. Skadhnagool, S. J. Kisner, and S. P. Bouman, Charles A.and Midkiff. Massively parallel 3d image reconstruction. Technical report, Purdue University, High Performance Imaging LLC, Microsoft Corporation, 2017.