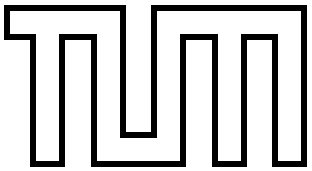


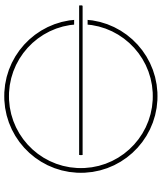
FAKULTÄT FÜR INFORMATIK

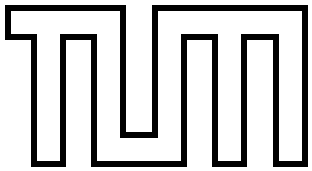
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing MPI Load Balancing in ls1 mardyn

Jeremy Harisch





FAKULTÄT FÜR INFORMATIK

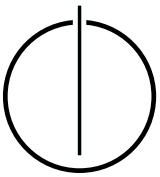
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing MPI Load Balancing in ls1 mardyn

Optimierung der MPI Lastbalanzierung in ls1 mardyn

Author:	Jeremy Harisch
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Steffen Seckler, M.Sc.; Fabio Alexander Gratl, M.Sc.
Date:	15.09.2020



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

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, 15.09.2020 Jeremy Harisch

Abstract

Since molecular dynamic simulations are very calculation heavy tasks, load balancing is one of the key points which should be optimized to gain an optimal runtime. In this thesis, three load balancing strategies of the simulation framework ls1-MarDyn and their optimizations will be discussed. Since ls1-MarDyn uses MPI as its communication protocol, some time is spent inside of the MPI-routines and therefore falsifies the measured time which is used as the input parameter for the load balancing. A way will be shown how to reduce this parameter by the time spent inside of the MPI-routines. Furthermore, this implementation is evaluated using three different simulation setups. The newly reduced time leads to an overall better performance - but this only applies if it is used in combination with a diffusive domain decomposition.

Contents

Part I.

Introduction and Background

1. Introduction

Molecular dynamic simulations are nowadays a popular way to research the behavior of up to trillions of molecules and atoms in the form of gases, liquids, or solids. They get used in the field of drug discovery [DM11], thermodynamics [FAC00], fighting nowadays diseases [CCWT08] and many more - most of the time when the experiments become too expensive or even impossible.

In these simulations, the atoms and molecules interact with each other using the forces given and calculated by the program - this is called an *N-Body-Problem*. This N-Body-Problem has a complexity of $\mathcal{O}(N^2)$ since each particle has to interact with the whole environment of N-particles. But this complexity can be reduced using some algorithms like the linked-cell algorithm, which subdivides the simulation environment using a cartesian mesh. Afterward, only the forces between particles in a given range, plus a given cutoff, of the neighboring cells will be calculated - which leads to a complexity of $\mathcal{O}(n)$. Using an algorithm, such as the mentioned linked-cell algorithm, makes it also easier to parallelize the calculation of the simulation since it is already subdivided into smaller sections. Each of those sections can then be allocated to the available computing nodes, using a decomposition-algorithm.

The reason why such a program should be parallelized is to ensure an optimal runtime and performance due to its calculation heavy routines. Decomposition algorithms try to balance the load to minimalize the time the processes have to wait for each other to finish their task, thus less time will be wasted by waiting. Three algorithms will be described in this thesis: A cartesian decomposition[Subsection 2.5.2], a k-d decomposition[Subsection 2.5.3], and a diffusive decomposition[Subsection 2.5.4].

In this thesis, the main focus will be on the General-Domain-Decomposition, which is based on comparing the runtimes from the tasks of all processes - These runtimes are called *last-traversal-time*. Unfortunately, in *ls1-MarDyn* the last-traversal-times are negatively affected by including the time spent inside of MPI-calls [Section 2.4]. Consequently, I demonstrate how it is possible to adapt the last-traversal-time to pass a more accurate time to the load balancer. In addition, a comparison of the old and the new implementation will be done in Chapter 5.

2. Background

2.1. ls1-MarDyn

ls1-MarDyn[1](large systems 1: molecular dynamics) is a still-in-progress open-source project by six institutes (Thermodynamics and Thermal Separation Processes, Technische Universität Berlin; High Performance Computing Center Stuttgart, University of Stuttgart; Laboratory for Engineering Thermodynamics, University of Kaiserslautern; Scientific Computing in Computer Science, Technische Universität München; Chair for High Performance Computing, Helmut-Schmidt-Universität; Scientific Computing Department, STFC Daresbury Laboratory)[2]. It is a molecular dynamics simulation framework that is optimized to run on supercomputing architectures. Furthermore, ls1-MarDyn is developed in a modular way and is still under development. Due to the modularity of the program, it is well optimized for easy extensibility.

The main part of the application is written in *C++* using many different libraries, including MPI which will be introduced in Section 2.4.

Since this project is for highly intensive molecular dynamic simulations, the program uses a linked-cell data structure which helps to split up the simulations into smaller subsections, which then will be assigned to available computing nodes. But this only works if a functioning MPI-library is installed on the machine, from now on we assume that it is. Also, *AutoPas*[3] will be used on each node for adaptive auto-tuning, more in Section 2.2.

The unit and the size of the timesteps of the simulation are going to be defined by the user, more in Subsection 2.1.1. After a given amount of iterations inside the simulation, the subsections are going to be rebalanced to gain a better load balancing which leads to a better runtime. For the rebalancing the user can choose from several strategies (DomainDecomposition [Subsection 2.5.2], k-d Decomposition [Subsection 2.5.3], General Domain Decomposition [Subsection 2.5.4]). The given input for those strategies are some parameters, stated in an XML-file by the user, and the LTT, given by the program itself. The last-traversal-time describes the time the process needed for calculating the new forces and positions of the particles within their subsections. Further load balancing implementation details will be described in Chapter 3.

2.1.1. XML-Input (in ls1-MarDyn)

To run a molecular dynamics simulation in ls1-MarDyn successfully, a proper XML-input-file is needed. It is used to define the parameters for the simulation, such as the general domain size, the definitions of all particles or molecules, and many more. Besides, plugins can be activated and their parameters can be adjusted. In ls1-MarDyn are already some plugins

[1] `https://github.com/ls1mardyn/ls1-mardyn`
[2] `https://www.ls1-mardyn.de/about.html`
[3] `https://github.com/AutoPas/AutoPas`

built-in and ready for use - Some Examples: A timer, a profiler, an individual potential per timestep, or several different output-writers. The structure of this XML-file depends on which plugins, load balancing strategies, etc. are going to be used.

In Listing 2.1 is an extract given of an example XML-input-file, which creates a domain of $[500, 500, 500]$, with a timestep-size of 1 and 100 timesteps in total. In addition, it is defined to use the General-Domain-Decomposition as the rebalancing strategy and to include AutoPas as a datastructure for each node.

```
1  [...]
2    <simulation type="MD">
3      [...]
4      <run>
5      <integrator type="Leapfrog">
6        <timestep unit="reduced">1</timestep>
7      </integrator>
8        <currenttime>0</currenttime>
9        <production>
10         <steps>100</steps>
11       </production>
12     </run>
13     <ensemble type="NVT">
14       <temperature unit="reduced">220</temperature>
15       <domain type="box">
16         <lx>500</lx>
17         <ly>500</ly>
18         <lz>500</lz>
19       </domain>
20       [...]
21     </ensemble>
22     <algorithm>
23       <parallelisation type="GeneralDomainDecomposition"/>
24       <datastructure type="AutoPas">
25       <allowedTraversals>c08</allowedTraversals>
26       <allowedContainers>linkedCells</allowedContainers>
27       [...]
28       </datastructure>
29       [...]
30     </algorithm>
31     <output>
32       <outputplugin name="CheckpointWriter">
33         [...]
34       </outputplugin>
35       [...]
36     </output>
37   </simulation>
38 [...]
```

Listing 2.1: Extract of an XML-input-file

2.2. AutoPas

AutoPas is an auto-tuned particle simulation framework on node-level and is part of the *TalPas*[4](Task-based Load Balancing and Auto-tuning in Particle Simulations) project. Node-level means that it can only run on a single node at a time.

Especially for ls1-MarDyn, this framework is very useful due to its foundation of self-optimizing algorithms for N-body simulations. Because AutoPas operates only on node-level, ls1-MarDyn creates one instance of AutoPas on each computing node to have an adaptive auto-tuning on all regions of the simulation. AutoPas is used as an extension to the ls1-MarDyn-project, by activating it during compilation time. In addition, some parameters of AutoPas need to be set using the XML-input-file[Subsection 2.1.1] - from now on we will always assume that it is activated, if not stated otherwise.

2.3. Molecular Dynamics - Theoretical Background

The following explanations all refer to the ls1-MarDyn-project, if not stated otherwise. A molecular dynamics simulation program, in general, is an application which calculates the forces and positions of given molecules or atoms in a given environment. For this, it uses a given physical background, stated in the next subsections.

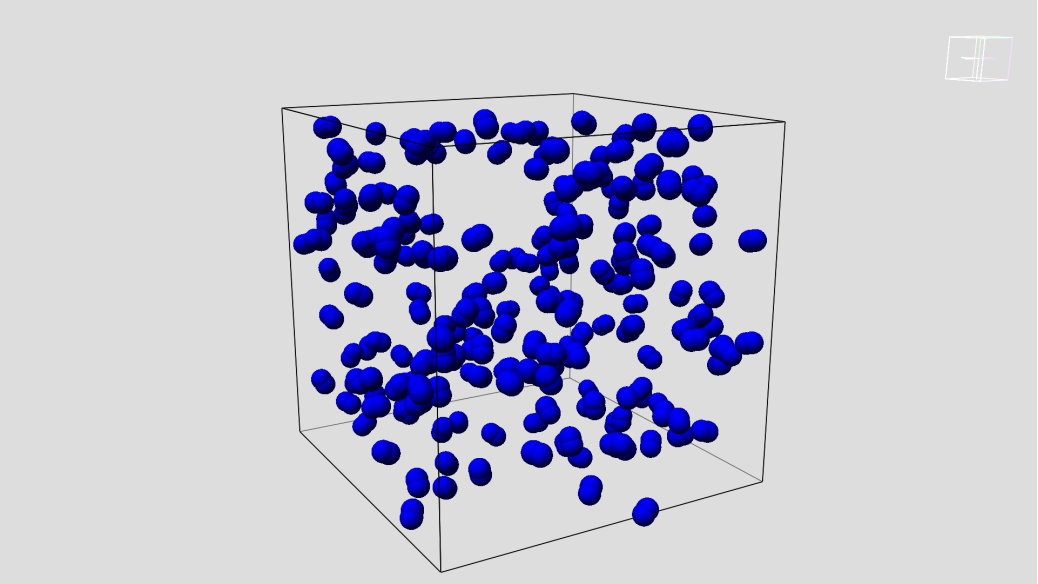


Figure 2.1.: N2-Gas Molecular Dynamic Simulation computed by ls1-MarDyn and visualized by ParaView Source: [n217]

2.3.1. Intermolecular Potentials

Between all particles acts a pairwise potential, this is called the *intermolecular potential*. In ls1-MarDyn the *Lennard-Jones-Potential*(Equation 2.1) is used for this as default, but other

[4] https://www.vi-hps.org/projects/talpas/

ones can also be used. It is defined through the relative range between two molecules r_{ij}, a potential well ϵ, and a zero-crossing ω.

$$U(r) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{2.1}$$

The potential between all molecules defines the resulting forces on each of them. The force and the velocity of each particle then leads to its position after a specific timestep. Furthermore, a *cutoff* radius can be introduced that conducts to a smaller amount of potential-calculations which have to be solved. It means that if a distance between the two molecules is bigger than the cutoff, the potential will not be calculated because it is significantly small, as you can see in Figure 2.2.

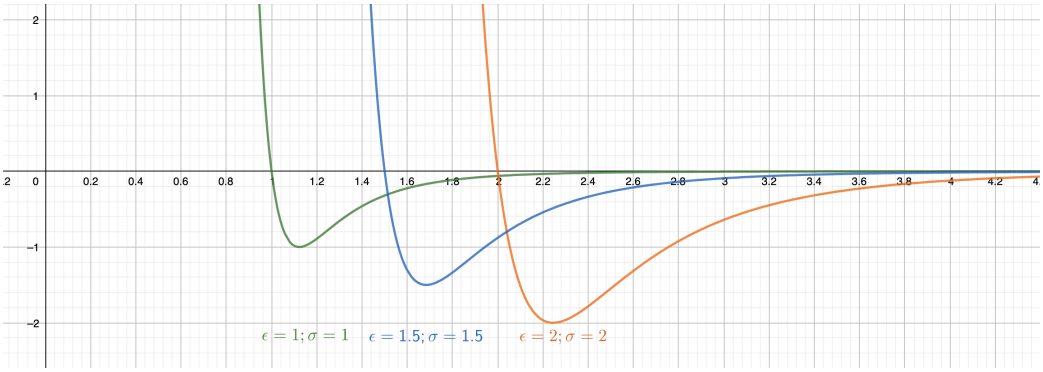


Figure 2.2.: Lennard-Jones-Potential for three different ϵ and σ

2.3.2. Newton's Laws of Motion

Newton's laws of motion are three laws that build up the foundation of physical movement between several bodies (here: molecules).

First law: If an object has a velocity of zero and has no force influencing it, it stays at a rest. If an object has a velocity unequal to zero and has no other net force influencing it, it moves in a straight line with perpetual speed [NMC68].

Second law: The momentum of an object is parallel to the magnitude and direction of the imposing force. Furthermore, it is conversely proportional to its body mass m. The resulting force is equal to $F = m * a$[NMC68].

Third law: The force F_1 of an object is the directional opposite of the force F_2 from the object which exerts F_2 on it, F_1 = - F_2. This means that F_1 and F_2 are equal in strength and the opposite in direction[NMC68].

2.4. Message Passing Interface (MPI)

The *Message Passing Interface* (short: MPI) is a communication protocol used for developing applications in the surrounding of parallel computing with the goal of high performance, scalability, and portability.

Until now, there are different well-known and tested implementations of MPI, such as *Open MPI*[5] or *Intel® MPI*[6] - For this thesis the Intel-implementation will be used.

MPI is used to define communication routes between several computing nodes in parallel applications. This means to parallelize an application still needs to be done by other parts of the program. MPI, therefore, provides an API for different network communication architectures like *Omnipath*[7], *Tofu*[8] or *Infiniband*[9][SL19]. This API has a variety of pre-implemented functions for communication, data reduction, and more. These implemented routines can be directly called from *C*, *C++*, and *Fortran* - If a working copy of an MPI implementation is installed and the library is included/imported in the program code.

Before each run of the application, it has to be defined how many *processes* the MPI-application has to start and run on. A process is one instance of a parallel thread from MPI. To define the number of processes, the user has to use the *np*-flag when executing the program. In Listing 2.2 you can see how the program *Example-Program* is called with 8 processes.

```
1 mpirun -np 8 ./Example-Program
```

Listing 2.2: Run MPI-Application, named *Example-Program*, with 8 processes

2.5. Load Balancing

Load balancing describes the process of distributing tasks over a given set of resources and trying to keep them in a good balance - with the goal of optimizing the overall performance. But every program has some different kind of task, thus there is not one solution which fits for every program to obtain the optimal performance.

Nowadays, there are different strategic approaches to converge as close as possible to the optimal solution for different problems. Three rebalancing strategies from ls1-MarDyn will be described in Subsection 2.5.2, Subsection 2.5.3 and Subsection 2.5.4.

2.5.1. ALL-library

The ALL[10]-library was developed by the E-CAM[11]-community and is part of the E-CAM meso- and multi-scale modules. The term 'module' is here used for any piece of software that creates a use to this community[oER19]. Among other things, ALL provides different dynamic load balancing techniques[oER19]. One technique, an implementation of the GDD, is used in ls1-MarDyn to balance the load, more in Subsection 2.5.4.

[5] https://www.open-mpi.org
[6] https://software.intel.com/content/www/us/en/develop/tools/mpi-library.html
[7] https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html
[8] https://www.fujitsu.com/global/documents/about/resources/publications/fstj/archives/vol48-3/paper05.pdf
[9] https://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf
[10] https://e-cam.readthedocs.io/en/latest/Meso-Multi-Scale-Modelling-Modules/index.html
[11] https://www.e-cam2020.eu

2.5.2. Domain Decomposition

MarDyn provides the Domain-Decomposition (short: DD) as a foundation for its implemented load balancing strategies, which are going to be introduced in the next subsections. The DD is not a real load balancing strategy since it is static and does not rebalance the load after creating sub-domains at the beginning of the simulation.

But it still splits the given domain into several sub-domains into a given form-factor. This is done coordinate-wise or in the data structure, in ls1-MarDyn it is done coordinate-wise in the form of cuboids. Then each of the sub-domains will be assigned to a computing node or process. Also, the neighbor sub-domains have to communicate with each other about the particles at the borders to update them correctly, see also Figure 2.5.

A dynamic application here will be a problem for the domain-decomposition because the load of the sub-domains can become an imbalance - thus the domains should also be adapted dynamically[BBF+03]. This is the point where the *General-Domain-Decomposition* [Subsection 2.5.4] helps out.

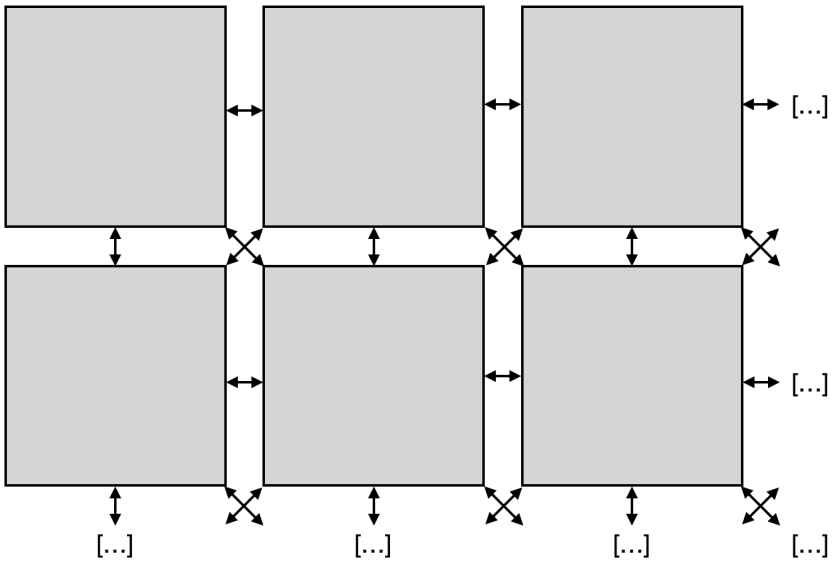


Figure 2.3.: Communication-structure of a domain decomposition algorithm using non-periodic boundaries. Arrows represent communication between domains(indicated by gray squares).

2.5.3. k-d Decomposition

The k-d Decomposition(short: KDD) relies on the principle of a k-d tree, which is a data-structure for partitioning given data into k-dimensions of the same size. But because of the straight borders between the sections, it does not work all the time.

In the surrounding of molecular dynamics, a k-d tree tries to determine where to set the borders between the particles in a way that each created section has the same amount of load. Then the load balancer assigns each of the sections to one of the available processes.

In Figure 2.4 is an example shown where a given space gets separated into four sections

with roughly the same amount of molecules. Each step, the redistributions, would be one rebalancing in the ls1-MarDyn project after a given amount of iterations.

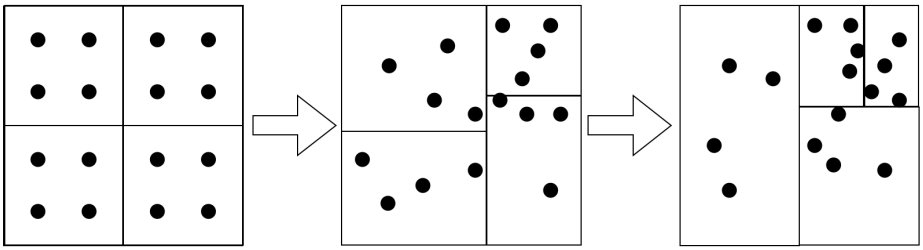


Figure 2.4.: First two redistributions of four subdomains using the k-d Decomposition. Tree gets built up new at each redistribution, trying to balance the amount of molecules in each subdomain.
Source: [ZGH+18]

2.5.4. General Domain Decomposition

The General-Domain-Decomposition (short: GDD) is based on the domain-decomposition [Subsection 2.5.2] with the big difference that it is not static and gets rebalanced in a diffusive way. Diffusive means that the borders of the sub-domains can shift in a way that the load of the processes are getting more balanced.

In ls1-MarDyn the load balancer shifts the borders by comparing the last-traversal-times of the adjacent sub-domains. The border of a sub-domain with a smaller last-traversal-time gets shifted into the direction of an adjacent sub-domain with a bigger last-traversal-time. In consequence, the imbalance between those two is more stable afterward. This mechanism then gets transferred to all sub-domains. Furthermore, the GDD-class in ls1-MarDyn acts as a wrapper for the ALL, described in Subsection 2.5.1. This means, that the just-described technique is originally implemented in ALL and then gets called from the load balancer of ls1-MarDyn. ALL takes care of restructuring the subdomains using the last-traversal-times of each sub-domain and the GDD-class then takes care of the particles in each domain. A more detailed look is given in Subsection 3.2.3.

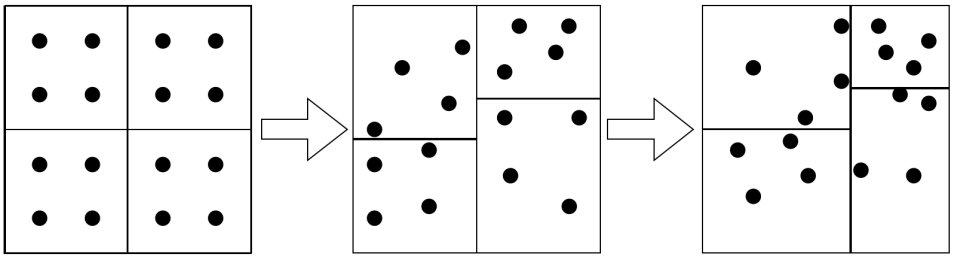


Figure 2.5.: First two redistributions of four subdomains using the General-Domain-Decomposition. Borders are getting shifted on each redistribution to balance the load of the subdomains.

Part II.

Load Balancing in ls1-MarDyn

3. Old Implementation

3.1. Measuring the Last-Traversal-Time

The load, in the ls1-MarDyn project, gets rebalanced after a given amount of iterations. This amount can be chosen by the user using the XML-input file. When the rebalancing function is called, it needs the last-traversal-time as an input. This time gets calculated by two checkpoints in the program, from now on let us call them *preTime* and *currTime*. *preTime* is the previous time of the rebalancing, thus at the first iteration *preTime* gets set to the current system time. In Algorithm 1 this time gets fetched by the function *currentSystemTime()*.

currTime can be set in different ways: One way would be by setting the current system time before the rebalancing is going to be started - this is the default timer. The other way would be by using the system time from the point where the force calculations of the domain have been finished - this timer is called *SIMULATION_FORCE_CALCULATION*-Timer(short: SFCT). Both ways have their pros and cons. One advantage of the first method is that plug-ins are considered for the rebalancing as well, a disadvantage would be that the time spent in MPI is included as well. The advantage and disadvantage of the second method is the exact opposite of the first one. Which means, an advantage is that time spent in MPI is automatically not included and a disadvantage is that plug-ins are not considered for the rebalancing. In Algorithm 1 this time is fetched by calling the function *currentTime()* The default timer will be used in this thesis, if not stated otherwise. The SFCT will be used in Chapter 5 to compare the different strategies.

Afterward, the last-traversal-time will be set equal to $currTime - preTime$. When the rebalancing has finished *preTime* will be set equals to *currTime*.

Algorithm 1: Calculating Last-Traversal-Time

```
   // Setting preTime to current system time in first iteration
1  preTime = currentSystemTime()
   // Updating preTime and currTime while simulation is running
2  while keepRunning do
3  |  currTime = currentTime()
4  |  lastTraversalTime = currTime - preTime
5  |  rebalance(lastTraversalTime)
6  |  preTime = currTime
```

Unfortunately, this time measurement takes anything into place what the program does - like print statements, writing into files, or the communication between the processes. How to adapt this time will be discussed in Chapter 4.

3.2. Using the Last-Traversal-Time

Just the KDD and the GDD will be explained in detail, because only those two are using the last-traversal-time as a factor for rebalancing. Furthermore, these will be used for the comparison of the newly implemented last-traversal-time calculation. The rebalance method of both algorithms, which will be called in each timestep, is named *balanceAndExchange(...)*. The implementation of this method for each algorithm, including major help functions, will be explained in Subsection 3.2.2 for the KDD and Subsection 3.2.3 for the GDD. In addition, both of the methods are using a frequency parameter to avoid updating the load balancing every timestep.

3.2.1. Pre-Defined Function: exchangeMoleculesMPI(...)

The KDD and the GDD are both using, among others, one pre-defined function, named *exchangeMoleculesMPI(...)*, to exchange molecules between the domains of the processes. In more detail: If a molecule is not in the region of the domain anymore, it gets transferred to its neighbor-domain. Furthermore, all the molecules from the halo region are getting transferred into it, if the parameter *doHaloPositionCheck* is set to true. This function is also optimized to reduce the maximum amount of domains to communicate with. For example, if a molecule needs to be transferred to the upper left domain, it first gets transferred to the upper one, afterward, it will be transferred from the upper to the upper left domain. As you can see in Figure 3.1, on the left side the domains have to communicate with three other domains, on the right side the domains only have to communicate with two domains each - this is also called *indirect neighbour communication*.

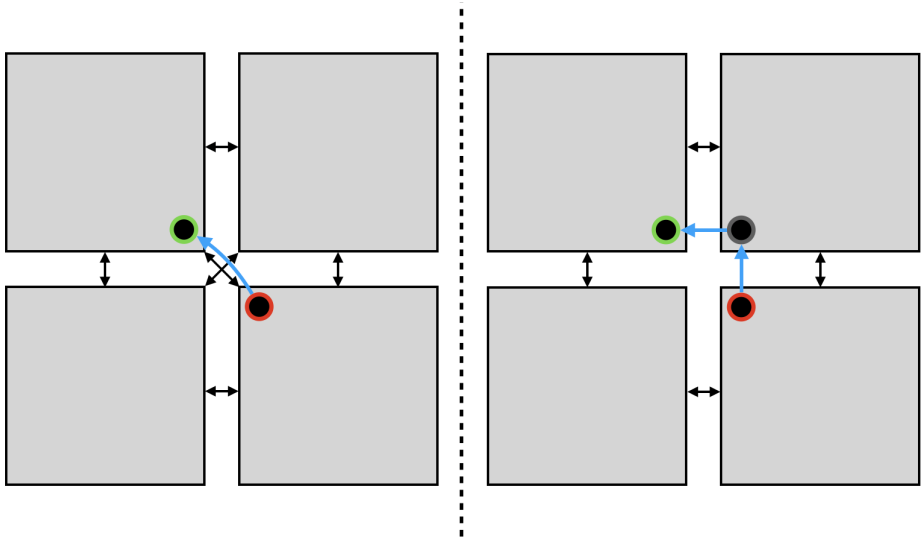


Figure 3.1.: Comparison between direct neighbor communication(left) and indirect neighbor communication(right) - Particle moves from bottom right(red marked position) subdomain to top left(green marked position) subdomain. Black arrows indicate communication between subdomains and blue arrows indicate the transfer of the particle.

The function itself needs five arguments in total:

- *moleculeContainer*: The container which holds all molecules inside of the simulation
- *domain*: The Domain of the current process
- *msgType*: Defines which particles should be exchanged, i.e. LEAVING_ONLY (exchange of all leaving molecules), HALO_COPIES (exchange of the halo copies), LEAVING_AND_HALO_COPIES (exchange of leaving molecules and halo copies)
- *doHaloPositionCheck*: Boolean if it should check for halo molecules; default is true
- *removeRecvDuplicates*: Boolean if it should remove received duplicates; default is false

In Listing 3.1 and Listing 3.2 will only the argument msgType be given, the other ones are not necessary for understanding the algorithms.

3.2.2. KD-Decomposition (Usage of LTT)

The KD-Decomposition uses the last-traversal-time as a secondary parameter, which here means that it is only used to determine if a rebalancing is necessary. In addition, the *balanceAndExchange(...)*-method uses two helper functions to determine if a rebalancing is needed, those are named *doRebalancing(...)* and *checkNeedRebalance(...)*.

checkNeedRebalance is the one which is called first. This function uses the last-traversal-time as an input and returns a boolean value. Using the MPI-function *MPIAllreduce(...)* with the *MPI_MAX* as a parameter, the last-traversal-times from all processes will be gathered, the maximum will be saved and processed in further instructions. If this maximum is greater than a set threshold (here: *_rebelanceLimit*), the return value will be set to True - if not it returns False.

doRebalancing(...) is called directly afterward. It returns a boolean and this will be used as the final decision if a rebalancing is needed. The task of this function is to check whether a rebalancing is forced or if a rebalancing is needed in the current timestep. To check if it is needed in this timestep, it uses the return value from the checkNeedRebalance-function, and in addition checks if the current timestep modulo the update-frequency is equal to 0 or if the current timestep is lower than 1 (which means that the simulation is in its first iteration).

The main rebalancing function *balanceAndExchange(...)* uses the two mentioned functions to determine what this function should do. If no rebalancing is needed it only exchanges the molecules between the processes using the exchangeMoleculesMPI-function [Subsection 3.2.1]. If a rebalancing is needed, it exchanges the leaving particles, builds up a new KD-Tree, rearranges the processes to it, and exchanges the halo particles at the end.

```
100 bool doRebalancing(bool forceRebalancing, bool needsRebalance, size_t steps,
        int frequency){
101   return forceRebalancing or ((steps % frequency == 0 or steps <= 1) and
          needsRebalance);
102 }
103
104 bool KDDecomposition::checkNeedRebalance(double lastTraversalTime) {
105   bool needsRebalance = false;
106   if (_rebalanceLimit > 0) {
107       // caluclates timerCoeff using given lastTraversalTime
...       [...]
```

```
113         MPI_CHECK(MPI_Allreduce(localTraversalTimes, globalTraversalTimes, 2,
                MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD));
...         [...]
118       // check if max. last-traversal-time is greater than a limit
119       if (timerCoeff > _rebalanceLimit) {
120         needsRebalance = true;
121       }
122     } else {
123       needsRebalance = true;
124     }
125     return needsRebalance;
126   }
127
128   void KDDecomposition::balanceAndExchange(double lastTraversalTime, bool
          forceRebalancing, ParticleContainer* moleculeContainer, Domain* domain) {
129     bool needsRebalance = checkNeedRebalance(lastTraversalTime);
130     bool rebalance = doRebalancing(forceRebalancing, needsRebalance, _steps,
            _frequency);
...     [...]
155     if (not rebalance) {
156         if ([...]) { // check for invalid particles
157         if (sendLeavingWithCopies()) {
158           DomainDecompMPIBase::exchangeMoleculesMPI(LEAVING_AND_HALO_COPIES);
159         } else {
160           DomainDecompMPIBase::exchangeMoleculesMPI(LEAVING_ONLY);
161   #ifndef MARDYN_AUTOPAS
162                   // delete outflow particles, if AutoPas is activated
163           moleculeContainer->deleteOuterParticles();
164   #endif
165           DomainDecompMPIBase::exchangeMoleculesMPI(HALO_COPIES);
166         }
167       } else {
168         DomainDecompMPIBase::exchangeMoleculesMPI(HALO_COPIES);
169       }
170     } else{
171         if (_steps != 1) {
172           DomainDecompMPIBase::exchangeMoleculesMPI(LEAVING_ONLY);
173         }
174         // Rebalance the load by creating a new KD-Tree
...         [...]
195         DomainDecompMPIBase::exchangeMoleculesMPI(HALO_COPIES);
196       }
197   }
```

Listing 3.1: Code-Structure of the KD-Decomposition

3.2.3. General-Domain-Decomposition (Usage of LTT)

The General-Domain-Decomposition uses the last-traversal-time as their main parameter for finding a new and probably better load balance than in the previous iterations. Instead of using it to check whether the load of the processes needs to be rebalanced, it just gets used to find a new distribution.

To check if the processes need a new distribution, a helper function named *queryRebalancing* gets called. This function checks if the current timestep modulo the update-frequency is

equal to 0. In this decomposition it differentiates between two frequencies - There is one frequency given for the simulation initialization phase and one for the actual simulation. And of course, it checks this operation with both of them. Therefore, setting up a new load balance does not depend on how good or bad the load balancing from previous iterations was, it just depends on the update-frequency and gets triggered at the last iteration of each frequency cycle. In the main rebalancing function *balanceAndExchange*, it gets checked if the simulation is in its first timestep, if this is true no rebalancing is needed, just the halo-copies of the molecules are getting exchanged between the sub-domains. If the simulation is not in its first timestep, it gets checked if a rebalancing is needed, using the output of the queryRebalancing-function. When it needs a rebalancing, it first exchanges all the leaving particles with the adjacent sub-domains and deletes all outer particles of the domain. Then the main rebalancing starts by calling the *rebalance*-function from the *loadBalancer* including the last-traversal-time as a parameter, implemented by the ALL. After this step has finished the halo-copies of the molecules are getting exchanged.

If the simulation needs no rebalancing in the current timestep, only the molecules of the subdomains are getting exchanged in such a way that there are no invalid particles available afterward.

```
100 bool GeneralDomainDecomposition::queryRebalancing(size_t step, size_t
        updateFrequency, size_t initPhase,size_t initUpdateFrequency, double /*
        lastTraversalTime*/) {
101   return step <= initPhase ? step % initUpdateFrequency == 0 : step %
          updateFrequency == 0;
102 }
103
104 void GeneralDomainDecomposition::balanceAndExchange(double lastTraversalTime,
        bool forceRebalancing,ParticleContainer* moleculeContainer, Domain* domain
        ) {
105   bool rebalance =
106     queryRebalancing(_steps, _rebuildFrequency, _initPhase, _initFrequency,
            lastTraversalTime) or forceRebalancing;
107   if (_steps == 0) {
108     // ensure that there are no outer particles
109     moleculeContainer->deleteOuterParticles();
110     // initialize communication partners
111     initCommPartners(moleculeContainer, domain);
112     DomainDecompMPIBase::exchangeMoleculesMPI(HALO_COPIES);
113   } else {
114     if (rebalance) {
115       // transfer leaving particles
116       DomainDecompMPIBase::exchangeMoleculesMPI(LEAVING_ONLY);
117       // ensure that there are no outer particles
118       moleculeContainer->deleteOuterParticles();
119       // rebalance
...             [...]
195       std::tie(newBoxMin, newBoxMax) = _loadBalancer->rebalance(
              lastTraversalTime);
...             [...]
195       DomainDecompMPIBase::exchangeMoleculesMPI(HALO_COPIES);
196     } else {
...       // exchange particles
...       [...]
195       }
```

```
196         }
197       }
198       ++_steps;
199     }
```

Listing 3.2: Code-Structure of the General-Domain-Decomposition

4. Reducing Last-Traversal-Time

As already mentioned in Chapter 3, the calculation of the last-traversal-time measures too much or too less to get an accurate parameter for the rebalancing strategies. One of them is the time which is spent inside of the MPI-function-calls, which means all the time the machine needs for the calculations when an MPI-function is called - in fact, this can make some serious differences. The time inside of an MPI-function can take up to several minutes, depending on the data which needs to be sent, received, or processed.

In the ls1-MarDyn project, MPI is used to let the processes communicate with each other - the main information they need to exchange is the information about the particles at the border of their sub-domain. And the more particles, the more data needs to be transferred, therefore the time spent inside of an MPI-call increases. In addition, ls1-MarDyn synchronizes the processes of MPI, which means the processes have to wait on each other to finish their task before starting a new one, as you can see in Figure 4.1. In consequence, waiting time will be measured which makes it useless as a parameter for balancing the load. Thus, to get a more efficient load balancing only the real molecule-calculation-times should be used as a parameter to find a better load balancing.

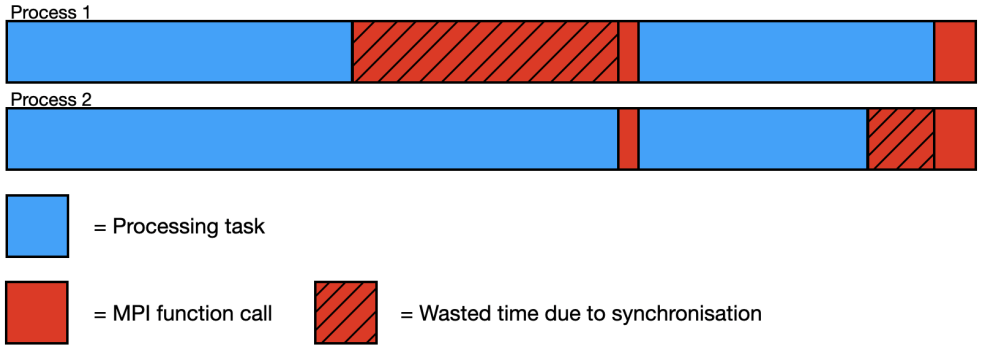


Figure 4.1.: Due to the synchronization of the MPI-processes, each process has to wait on each other after their main timestep calculations. This wasted time is also used for the load balancing strategies, which falsifies the input parameter and thus the load balancing in general.

4.1. Problems and Solution

To reduce the last-traversal-time by the time spent in MPI-calls, it needs to be measured. A research was done if an existing library is capable of doing this. Unfortunately, those which were analyzed only show results after the calculations of the given application have been

finished. However, to rebalance the load dynamically during the runtime a profiler is needed, on which the results can be checked while the simulation is running.

Fortunately, the MPI-library has a monitoring interface called PMPI, which will be used as the solution to track the time spent in MPI-calls. In further detail, the PMPI-functions will be used to create a name shift and override the existing MPI-functions.

In the MPI-library all the native MPI-functions are implemented as weak symbols, accordingly they can be overridden by the user - More in Subsection 4.2.4. This is used to reimplement all used MPI-functions with a built-in timer and using the PMPI-functions to trigger the original action the user wanted to call.

4.2. Implementation

The implementation consists of two main parts: The first one is a timer that is used for time measuring. The second one is the new implementation of the MPI-functions, which use the implemented timer.

4.2.1. Process-Timer

The timer consists of one class which is called *ProcessTimer* and it is mainly based on managing one *double* variable, named *_process_time*. This variable is used for storing the measured time of the current process. If std::thread[1] would be used for parallelizing the application, a map, mapping from thread-IDs to the measured times, should be used instead. As each MPI-process has its own stack, as well as heap, the *_process_time* variables, one in each process, do not interfere with each other.

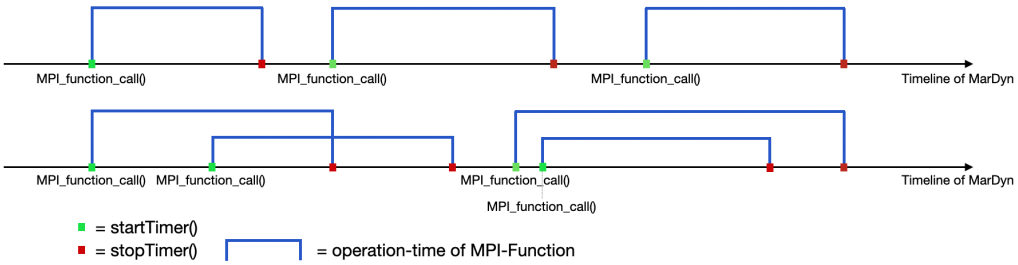


Figure 4.2.: Two example timelines for MPI-function-calls, which show that MPI functions can be called sequentially as well as parallel. Besides, the *Process-Timer* is shown how it handles the time measurement using the two functions *startTimer()* and *stopTimer()*. The green dots mark the time when the *startTimer()*-function is called, the red dots mark the time when the *stopTimer()*-function is called. Due to the implementation of the timer, *startTimer()* can be called several times in a row without the need of stopping the timer before starting it again. Thus, time can be measured for two MPI-function-calls simultaneously.

To start the timer for the current process the function *startTimer* needs to be called. This function checks for the process-ID, used later for a debugging-map, and gets the current

[1] `https://en.cppreference.com/w/cpp/thread/thread`

system wall-time using *MPI_Wtime()*. This time gets subtracted from the *_process_time* variable.

To stop the timer afterward, the function *stopTimer* needs to be called. This function also checks for the process-ID, used for the debugging-map as well, and the wall-time as the start-function does. Then the checked time gets added to the *_process_time*. Thus, the value which results from this calculation is the exact time between the start and the end-point. One benefit of using this method is that when it is used in combination with std::atomic[2] and more than one MPI-function needs to be measured for a single process at the same time, it would work - this can happen if threading is used on top of MPI-processes. std::atomic is used to ensure the racing-conditions of the threads. An example timeline can be found in the Figure 4.2.

Furthermore, a function, named *getTime*, is implemented which returns the measured time of a given process. This function has two optional parameters which are both set to *false* by default:

- *reset*(optional; default = false): If true the measured time of the process will be reset to 0 after checking the time.
- *debug*(optional; default = false): If true the measured time will be written into a csv-file, which then can be used in other ways during or after the runtime.

```
1  private:
2      double _process_time;
3      std::map<int, std::vector<double>> _processes_debug;
4      int _profiling_switch = 1;
5
6  public:
7      void startTimer() {
8        if (_profiling_switch == 1){
9          int process;
10         MPI_Comm_rank(Comm, &process);
11         double measurement_time = MPI_Wtime();
12         _process_time -= measurement_time;
13         _processes_debug[process].push_back(-measurement_time);
14       }
15     }
16
17     void stopTimer() {
18       if (_profiling_switch == 1){
19           double measurement_time = MPI_Wtime();
20         int process;
21         MPI_Comm_rank(MPI_COMM_WORLD, &process);
22         _process_time += measurement_time;
23         _processes_debug[process][_processes_debug[process].size()-1] -=
               measurement_time;
24       }
25     }
26
27     double getTime(bool reset = false, bool debug = false){
28       double time = _process_time;
29       if (debug){
```

[2] https://en.cppreference.com/w/cpp/atomic/atomic

```
30              int process;
31              MPI_Comm_rank(MPI_COMM_WORLD, &process);
32              writeProcessTimeLogSingle(process, _process_time, true);
33          }
34          if (reset)
35              resetTimer(process);
36          return time;
37      }
38
39      void resetTimer(int process) {
40          _process_time -= _process_time;
41  }
... [...]
```

Listing 4.1: Extraction from the implemented timer class - Main functions

Additionally, this class has a profiling switcher, called *_profiling_switch*, which allows us to switch the time measuring on and off. This can be used to exclude parts of the program which should not be taken into account for calculating the time spent inside of MPI-calls. It is implemented as an integer to stick to the structure of the profiling switcher given by the MPI-library.

4.2.2. MPI-Profiling-Interface

The profiling interface uses the timer described in Subsection 4.2.1 as its main and only timer. This interface consists of reimplementations of MPI-functions which are used in the ls1-MarDyn project - the list of reimplemented functions can be found at the end of Section 4.2.2.

The reimplemented functions have the same declarations as the ones from the MPI-library, therefore those are overwritten when the compiled code gets linked together, more in Subsection 4.2.4. The structure of each reimplemented function looks the same: First, the timer for the process gets triggered to start the measurement. Afterward, the corresponding PMPI-function gets called with the given parameters. At the end, the timer of the current process gets stopped.

To tell the program which implementation each class should use, the include-section of those classes have to be adapted. This means that the inclusion of the MPI-library is not needed anymore, instead, this profiling interface needs to be included.

```
1  int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest,
2               int tag, MPI_Comm comm) {
3    _processTimer.startTimer();
4    int result = PMPI_Send(buf, count, datatype, dest, tag, comm);
5    _processTimer.stopTimer();
6    return result;
7  }
8
9  int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
10            MPI_Comm comm, MPI_Status *status) {
11   _processTimer.startTimer();
12   int result = PMPI_Recv(buf, count, datatype, source, tag, comm, status);
13   _processTimer.stopTimer();
14   return result;
15 }
```

Listing 4.2: Exampe implementation of MPI_Send and MPI_Recv in the MPI-Profiling-Interface

Full list of reimplemented MPI-functions

- MPI_Allreduce
- MPI_Bcast
- MPI_Cart_coords
- MPI_Cart_create
- MPI_Cart_rank
- MPI_Comm_free
- MPI_Dims_create
- MPI_Exscan
- MPI_File_close
- MPI_File_get_position
- MPI_File_open
- MPI_File_write
- MPI_File_write_at
- MPI_Finalize
- MPI_Get_address
- MPI_Get_count
- MPI_Iprobe
- MPI_Irecv
- MPI_Isend
- MPI_Op_create
- MPI_Op_free
- MPI_Probe
- MPI_Recv
- MPI_Send
- MPI_Test
- MPI_Type_commit
- MPI_Type_create_struct
- MPI_Type_free
- MPI_Pcontrol

4.2.3. Adapting the used LTT

The adaption of the last-traversal-time happens when the particle containers are getting updated and the rebalancing-function is triggered - this is done by the function which is named *updateParticleContainerAndDecomposition(...)*. This function gets triggered in the main simulation-loop.

To adapt the LTT, used as the input parameter for *updateParticleContainerAndDecomposition(...)*, the *getTime(...)*-function of the *ProcessTimer*(Subsection 4.2.1) is called with the parameter *reset* set to True and *debug* set to False. This function then returns the time spent inside the MPI-routines of the current process since the last rebalancing. Then *updateParticleContainerAndDecomposition(...)* is called with the LTT minus the return-value of *getTime(...)* as you can see in Listing 4.3.

```
1   // Implementation of the simulation class
... [...]
101     // Main Simulation-Loop
102     while (keepRunning()) {
...         [...]
201         double lastTraversalTime = currentTime - previousTimeForLoad;
202         double timeSpentInMPI = _processTimer.getTime(true, false);
203         updateParticleContainerAndDecomposition(lastTraversalTime -
                timeSpentInMPI);
... [...]
```

Listing 4.3: Extraction of the main simulation-loop

4.2.4. Linking the Application

ls1-MarDyn is written in *C++*, therefore the code has to be compiled and linked to run the actual application. The compilation of the application is not changed for this thesis, only the linking stage needs to be adapted.

In the linking stage the object files, the output of the compilation stage, are getting checked for weak and strong symbols. Each weak symbol can be overwritten by a strong symbol, but a strong symbol cannot be overwritten by a weak symbol. This also means that only one strong symbol of each type can exist, otherwise, the linker would quit the linking of the application and the application cannot be built.

The PMPI interface is implemented in such a way that its methods are declared as weak symbols and can be overwritten by strong symbols - by default, all declared symbols are strong symbols. But this only works if the linker supports weak external symbols. If it does not support them, the linker has to be told that multiple definitions of strong symbols are allowed. Then it will use the definition which uses more memory space, which is in this case the reimplemented MPI-function. This can be done by adding the argument *--allow-multiple-definition* as a linker flag.

```
1 [...]
2 LDFLAGS +=  -Wl,--allow-multiple-definition
3 [...]
```

Listing 4.4: Extract of makefile - shows which linker flags have to be added in the makefile to link the application properly.

Part III.

Evaluation and Conclusion

5. Evaluation

To compare the old implementation with the new approach of reducing the last traversal time three examples will be introduced in the following subsections. Furthermore, in each example section, the old and new approaches will be compared for each.

5.1. Examples

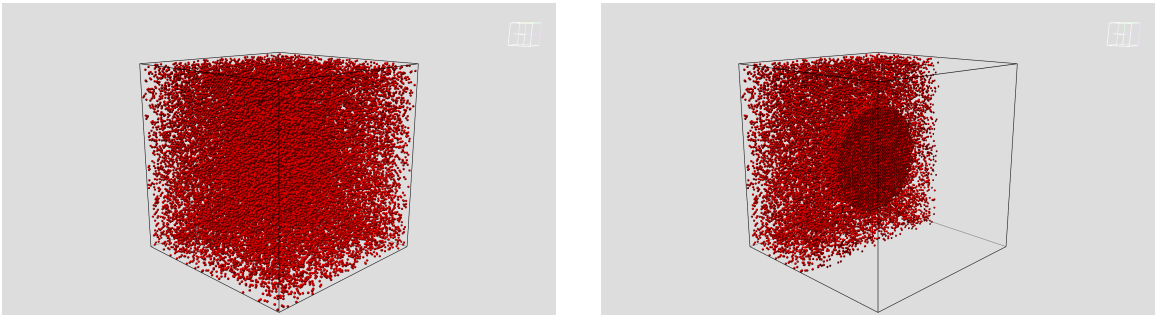
All examples will be compared with the *Domain Decomposition*, the *General Domain Decomposition* and the *kd Decomposition* using the old implementation, the new implementation and the old implementation in combination with the use of the SFCT, described in Chapter 3.

All simulations will run on 16 MPI-processes and use AutoPas, with the exception when the KDD is used. When using the KDD the simulations will run on 16 processes as well, but AutoPas is not compatible with this decomposition algorithm, so it will not be used - the default *LinkedCells*-datastructure will be used instead.

As an evaluation factor, the total runtime of the simulations(charts on the left) and the standard deviation(SD, charts on the right) of the processes, over 10 simulation timesteps each and 100 simulation timesteps in total, will be used.

5.1.1. mkesfera

The *mkesfera* simulation (Figure 5.1a) is a scenario that simulates a sphere of molecules surrounded by floating single molecules. The sphere has a higher density of molecules than the rest of the simulation, accordingly they are pulling each other together and the sphere stays almost in its form. Because the simulation runs with 16 processes, the number of molecules in each process domain is very unbalanced at the beginning. The reason for this is that the created domains are the same size before the first rebalancing - so the domains in the middle of the simulation area contain parts of the sphere with a high density, and the other ones contain molecules of the surrounding gas with a lower density.



(a) full simulation
Source: [mke17a]

(b) slice of simulation
Source: [mke17b]

Figure 5.1.: Picture of *mkesfera* simulation

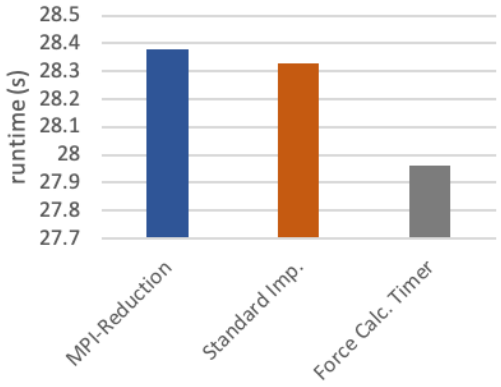
As you can see in Figure 5.2a and Figure 5.2b with the DD the mkesfera simulation performs almost the same for each implementation, this is because by using the DD the load of the processes do not get rebalanced at all. Only at the beginning of the simulation the subdomains are created and stay the same for all the timesteps.

When the GDD is used it can be seen that the new implementation, where the LTT gets reduced by the time spent inside MPI, achieves the same results as by using the SFCT. This can be seen in Figure 5.2c for the total runtime and also in Figure 5.2d for the standard deviation.

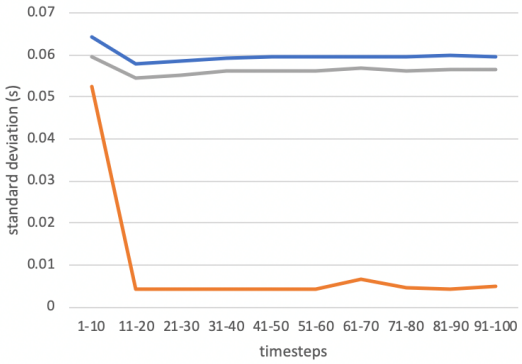
Using the KDD and the new implementation together does not lead to good results in this example (Figure 5.2e, Figure 5.2f) - the SD is not getting lower during the simulation, while the overall runtime increases.

5.1.2. Evaporation

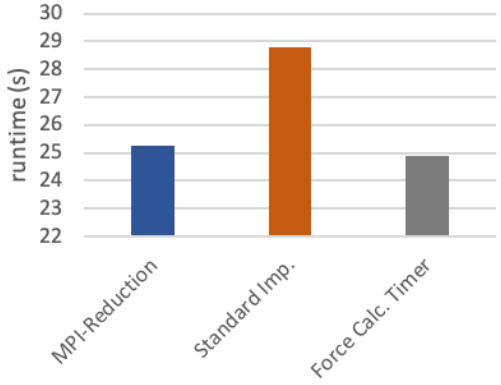
The *Evaporation* example simulates the evaporation of a liquid into the vacuum(Figure 5.3). The green-colored molecules are a liquid reservoir which is connected to the evaporating liquid (blue-red-colored) through a transition plane - and this liquid is connected to the vacuum through a planar interface [HV19]. The thermostat which is heating up the liquid is limited to the left 75% of the blue-red-colored liquid so that the transition between the liquid and vapor phase evolves naturally [HV19]. Throughout the simulation, the liquid is heating up until it starts to evaporate and turns into vapor.



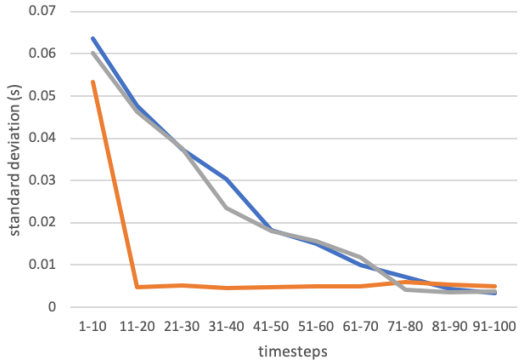
(a) Total Runtime - Domain Decomposition



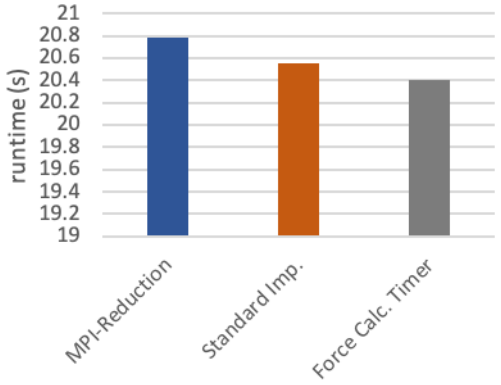
(b) Standard Deviation - Domain Decomposition



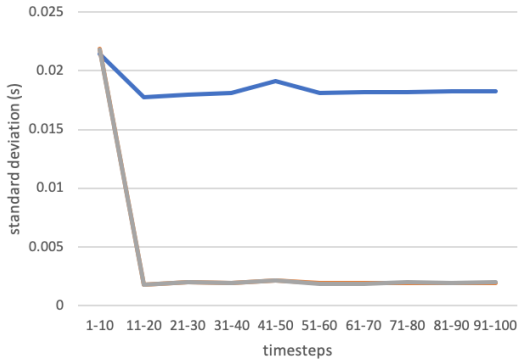
(c) Total Runtime - General Domain Decomposition



(d) Standard Deviation - General Domain Decomposition



(e) Total Runtime - kd Decomposition



(f) Standard Deviation - kd Decomposition

Figure 5.2.: Example: mkesfera - Comparison

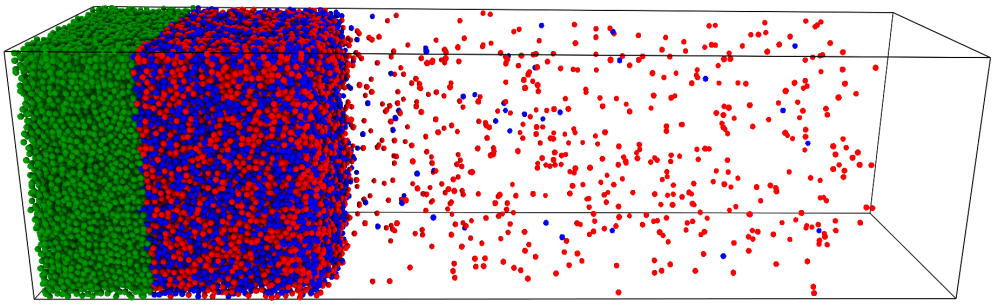


Figure 5.3.: Picture of *Evaporation* simulation
Source: [eva17]

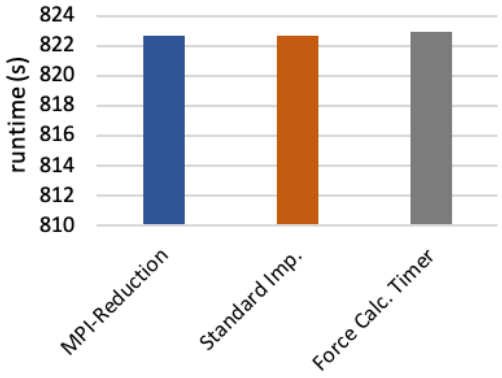
As stated in the previous example (Subsection 5.1.1) when using the DD the total runtime and the SD relies on how the molecules are distributed in the simulation space. Therefore, the overall runtime is pretty close to each other (Figure 5.4a), while the SD of the SFCT is higher than the others in general.

When using the GDD, the new implementation performs better than both, the old implementation and using the SFCT as a timer. The new implementation converges faster to an optimal load balancing, low SD, than the SFCT (Figure 5.4d), which then also leads to a better total runtime (Figure 5.4c).

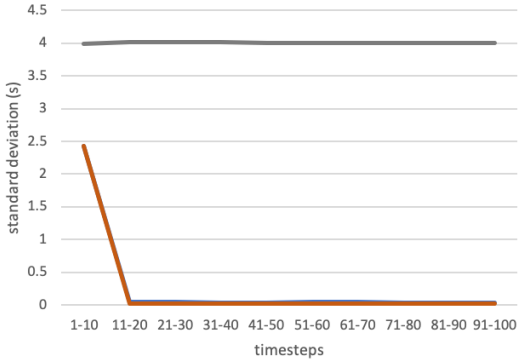
By using the KDD as the rebalancing strategy, the SD is higher and the overall runtime increases contrary to the old implementation and the SFCT in combination with the old implementation.

5.1.3. Vapor–liquid equilibrium

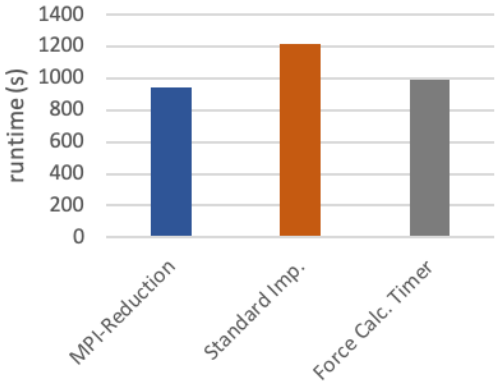
This *Vapor–liquid equilibrium* simulation simulates the scenario of thousands of *CO2*(carbon dioxide) molecules and their distribution in the given space between their liquid and vapor phases [Kis92](Figure 5.5). In the middle section of the simulation domain, the molecules with the high-density are in the liquid phase, the surrounding molecules represent the vapor phase of the *CO2*.



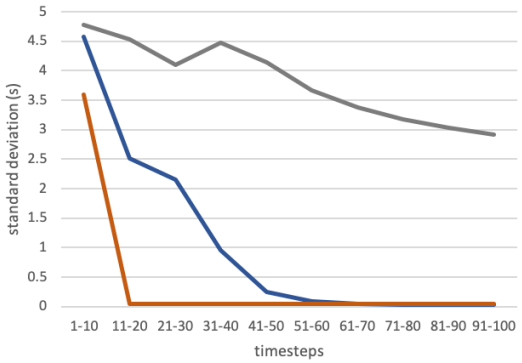
(a) Total Runtime - Domain Decomposition



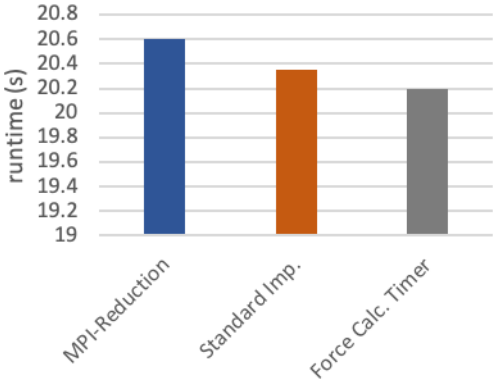
(b) Standard Deviation - Domain Decomposition



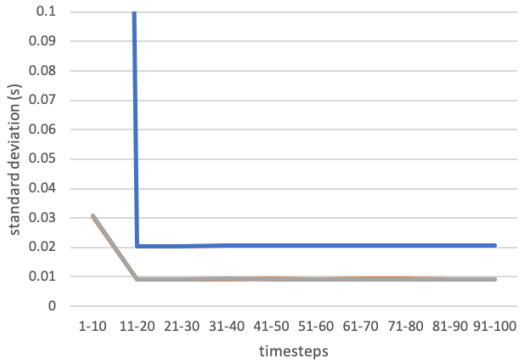
(c) Total Runtime - General Domain Decomposition



(d) Standard Deviation - General Domain Decomposition



(e) Total Runtime - kd Decomposition



(f) Standard Deviation - kd Decomposition

Figure 5.4.: Example: Evaporation - Comparison

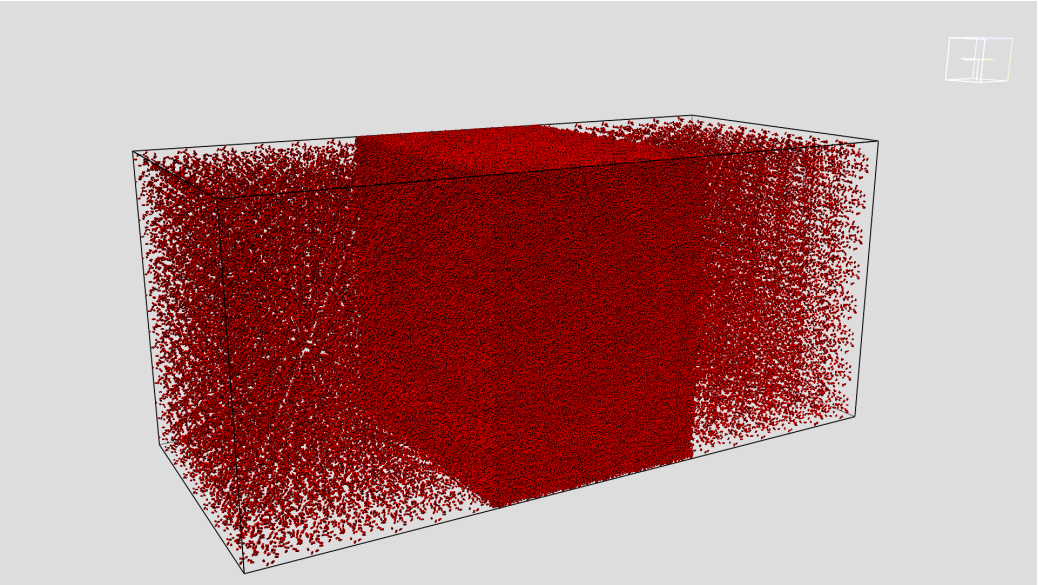


Figure 5.5.: Picture of CO2-VLE simulation
Source: [vle17]

As seen in the other two examples (Subsection 5.1.1 and Subsection 5.1.2), using the DD depends on the distribution of the molecules at the first timestep and how the subdomains are being created. Thus, the SD of all three stay pretty much the same (Figure 5.6b).

Using the GDD as a load-balancing-strategy, the new implementation behaves similarly to the old implementation as to the SFCT(Figure 5.6d) - And, the overall runtime is lower than both (Figure 5.6d). This leads to the result, that the new implementation can be used efficiently for this scenario.

The KDD gives as bad results as the other examples (Subsection 5.1.1 and Subsection 5.1.2) for the new implementation. The SD of the new implementation is approximately four times higher than the old implementation as well as the SFCT during the whole simulation (Figure 5.6f), which then leads to a higher overall runtime (Figure 5.6e).

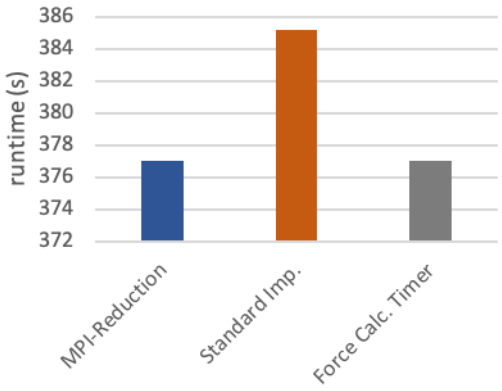
5.2. Result

To sum up the results of the examples presented in Section 5.1:

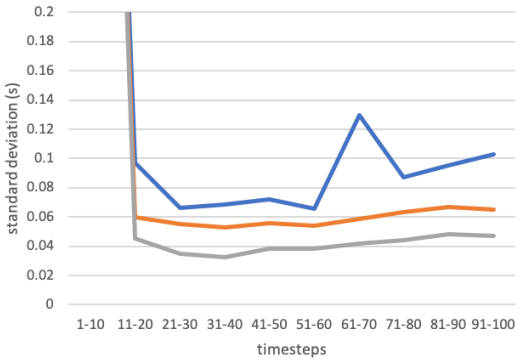
The new implementation is a good alternative for the SFCT used for the last traversal time. In general, by using the reduced time as the LTT the same or better results for the load balancing can be achieved. But this only applies to a simulation where the GDD is used as the load balancing strategy.

If KDD is used instead, this implementation should not be used due to the results. In all three examples, the load balancer cannot handle the new last traversal time as a rebalancing factor, if a rebalancing should be triggered or not. For finding the new load balancing the LTT is not used, therefore the new approach in combination with the KDD should not be used to optimize the load balancing of ls1-MarDyn.

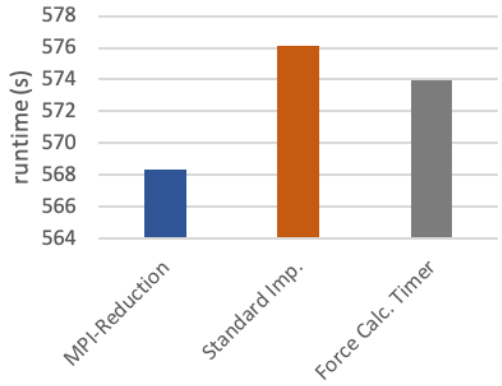
The reason why the same or better results can be achieved, when using the GDD, is that the time which is wasted due to the synchronization of the processes is not considered as a



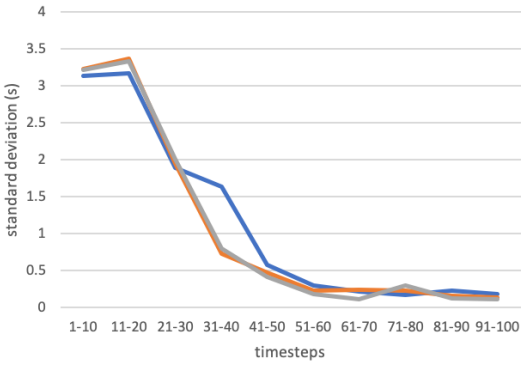
(a) Total Runtime - Domain Decomposition



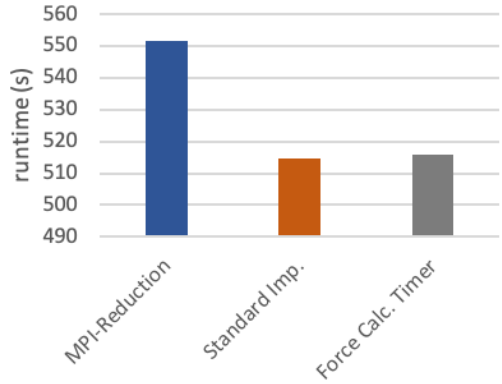
(b) Standard Deviation - Domain Decomposition



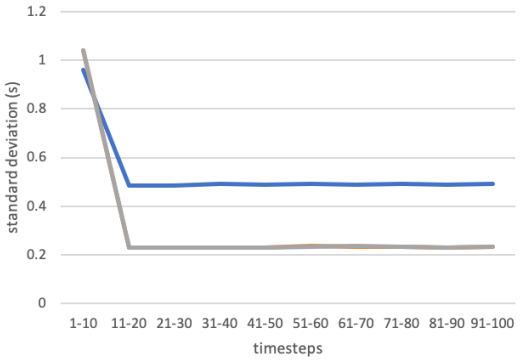
(c) Total Runtime - General Domain Decomposition



(d) Standard Deviation - General Domain Decomposition



(e) Total Runtime - kd Decomposition



(f) Standard Deviation - kd Decomposition

Figure 5.6.: Example: VLE - Comparison

factor for the load balancer anymore. This means that the input parameter is more precise and leads faster to an optimal load balancing. When using the SFCT, this time is also not considered for the load balancing, but it has the disadvantage that the time used for the plug-ins on each process is not considered as well. Thus, the new approach performs sometimes better than the SFCT.

	Origin LTT	**Time spent in MPI**	**Used LTT**	**Reduction (%)**
mkesfera	14709,22348	877,575299	13831,64745	5,97
Evaporation	243,2337441	58,20635789	185,0274022	23,93
VLE	5591,936039	486,658521	510,5277754	8,70

Table 5.1.: Reduction of the LTT using the time spent in MPI-calls

Also, it needs to be shown how much the LTT gets reduced by using the new implementation in the three different simulations. The times shown in Table 5.1 are the times measured by ls1-MarDyn and summed up over the whole simulation. The '*Used LTT*' is the time that is given as the parameter to the load balancer, which means this is the result of the '*Origin LTT*' minus the '*Time spent in MPI*'. This all leads to a reduction of $5,97\%$ in the *mkesfera*-simulation, $23,93\%$ in the *Evaporation*-simulation and $8,7\%$ in the *VLE*-simulation. That shows us that the time used for the load balancing was significantly falsified by the time spent in MPI-calls.

6. Conclusion and Outlook

The main goal of this thesis was to optimize the load balancing of the ls1-MarDyn project by adapting the used parameter, called *last-traversal-time*, due to the time spent inside of MPI-calls. This time should be reduced by the time needed for MPI-calls because the processes of the application need to be synchronous. This means that the processes have to wait for each other inside of some plug-ins, after the calculations of the temperature, pressure, etc., as well as after each timestep when all processes have finished their calculations for their subdomain. And this waiting time, amongst others, was one of the causes why the original LTT leads not to an optimal load balancing.

The solution stated in this thesis is reducing the LTT by the time spent in MPI-calls, thus this time needs to be measured. For this, a timer was implemented and used by a newly implemented profiling interface. This profiling interface overrides the used MPI-functions by reason that it can start a timer, trigger the wanted MPI-function, and stop the timer. Furthermore, it was implemented in a way, that this timer can also be used even if more than one MPI-function is needed at the same time. This measured time then gets subtracted from the LTT when it gets used as the parameter for the load balancer.

This modified LTT was then evaluated using different load balancing strategies and different example simulations. The result was that this approach can be used for a better load balancing in combination with the GDD. It achieves the same or better results than the SFCT-timer and has also the advantage that the used plug-ins are not ignored for rebalancing the load of the processes. Unfortunately, this does not apply to the KDD, the evaluation of this load balancing strategy leads on average to a higher standard deviation and to a higher total runtime than the old implementation or the SFCT-timer. Which leads to the result, that this approach should not be used in combination with the KDD.

In a retroactive view, the approach of this thesis turned out as a good foundation for further load balancing optimizations. Future works could be a full integration into ls1-MarDyn as a plugin, which then could be activated by the XML-input-file.

Acronyms

ALL A Load-balancing Library. 7, 9, 16

API application programming interface. 7

DD Domain-Decomposition. 8, 26, 28, 30

E-CAM extreme-scale computing for industry and academia. 7

GDD General-Domain-Decomposition. 7, 9, 13, 26, 28, 30, 33

KDD KD-Decomposition. 8, 13, 25, 26, 28, 30, 33

LTT last-traversal-time. v, 3, 14, 15, 22, 26, 30, 32, 33, 36

MPI message parsing interface. i, ii, iv, v, 2, 3, 6, 7, 12, 14, 18, 19, 20, 21, 22, 23, 25, 26, 32, 33, 35, 36

PMPI profiling message parsing interface. 19, 21, 23

SD Standard Deviation. 25, 26, 28, 30

SFCT SIMULATION_FORCE_CALCULATION-Timer. 12, 25, 26, 28, 30, 32, 33

VLE Vapor–liquid equilibrium. 30, 35

List of Figures

List of Tables

Bibliography

[BBF+03] Fabrizio Baiardi, Alessandra Bonotti, Luca Ferrucci, Laura Ricci, and Paolo Mori. Load balancing by domain decomposition: the bounded neighbour approach. *Proc. of 17th European Simulation Multiconference*, pages 9–11, 01 2003.

[CCWT08] Calvin Yu-Chian Chen, Yuh-Fung Chen, Chieh-Hsi Wu, and Huei-Yann Tsai. What is the effective component in suanzaoren decoction for curing insomnia? discovery by virtual screening and molecular dynamic simulation. *Journal of Biomolecular Structure and Dynamics*, 26(1):57–64, 2008. PMID: 18533726.

[DM11] Jacob D. Durrant and J. Andrew McCammon. Molecular dynamics simulations and drug discovery. *BMC Biology*, 9(1):71, Oct 2011.

[eva17] Snapshot of the evaporation simulation. `https://github.com/ls1mardyn/ls1-mardyn/blob/master/examples/Evaporation/stationary/png/sim02_run02.png`, 2017. [Online; accessed August 19, 2020].

[FAC00] Philippe Ferrara, Joannis Apostolakis, and Amedeo Caflisch. Thermodynamics and kinetics of folding of two model peptides investigated by molecular dynamics simulations. *The Journal of Physical Chemistry B*, 104(20):5000–5010, 2000.

[HV19] Matthias Heinen and Jadran Vrabec. Evaporation sampled by stationary molecular dynamics simulation. *The Journal of Chemical Physics*, 151:044704, 07 2019.

[Kis92] Henry Z. Kister. *Distillation Design*. 1992.

[mke17a] Snapshot of the mkesfera simulation. `https://github.com/ls1mardyn/ls1-mardyn/blob/master/examples/Generators/mkesfera/snapshots/mkesfera.png`, 2017. [Online; accessed August 19, 2020].

[mke17b] Snapshot of the mkesfera simulation cut in half. `https://github.com/ls1mardyn/ls1-mardyn/blob/master/examples/Generators/mkesfera/snapshots/mkesfera_cut.png`, 2017. [Online; accessed August 19, 2020].

[n217] Snapshot of a n2-vap simulation. `https://github.com/ls1mardyn/ls1-mardyn/blob/master/examples/Generators/ReplicaGenerator/heterogeneous/Standard-VLE/N2_118K/vap/n2_vap.png`, 2017. [Online; accessed August 19, 2020].

[NMC68] Isaac Newton, Andrew Motte, and Bernhard Cohen. *The mathematical principles of natural philosophy*. Dawsons of Pall Mall, 1968.

[oER19] E-CAM Centre of Excellence Revision. All (a load-balancing library), 2019.

[SL19] Germany Sebastian Lührs, Forschungszentrum Jülich GmbH. Best practice guide - modern interconnects. Feb 2019.

[vle17] Snapshot of the co2-vle simulation. `https://github.com/ls1mardyn/ls1-mardyn/blob/master/examples/Generators/ReplicaGenerator/heterogeneous/Standard-VLE/CO2_Merker_220K/vle/co2_vle.png`, 2017. [Online; accessed August 19, 2020].

[ZGH+18] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic load balancing based on constrained k-d tree decomposition for parallel particle tracing. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):954–963, 2018.