DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing Load Balancing of Heterogeneous Particle Simulations on Heterogeneous Systems

Simon Griebel

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Optimizing Load Balancing of Heterogeneous Particle Simulations on Heterogeneous Systems

Optimierung der Lastverteilung Heterogener Partikelsimulationen auf Heterogenen Systemen

Author:	Simon Griebel
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Steffen Seckler
Submission Date:	July 17, 2017

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

Munich, July 17, 2017 Simon Griebel

Abstract

In this thesis improvements to the k-d-tree based load balancing algorithm of the molecular dynamics simulation framework MarDyn will be presented. While already being highly scalable its load balancing algorithm still left room for improvement, because its central part, the load estimation function, was solely based on heuristics. For this work it was replaced with a function that actually uses time measurements for the estimation. These measurements then made it easier to expand the algorithm so it would be able to correctly estimate the load for simulations with more than one particle type and for simulations that are run on heterogeneous systems. The k-d-tree based load balancing algorithm itself was also modified. Some of these changes were needed for the support of heterogeneous systems, but in this thesis there is also a short look at how certain modifications to the splitting rules of the k-d-tree-nodes can influence performance.

Contents

1 Introduction

Simulations became an essential part of science and engineering, because they enable the test of new and interesting ideas without needing to create an expensive, time intensive test prototype, which might even be destroyed as part of the tests. Computer simulations can offer a cheap and flexible alternative, but it is normally quite hard to model a real life problem on the computer. The more detailed these models get, the more computing power is needed. This is a problem, since as the increase in speed of single processors/cores slows down, it becomes more and more important to parallelize the computer simulations, so that the power of multiple processors can be harnessed. This parallelization makes the program even more complex, because now the load distribution and the interaction between the different processors has to be considered.

One of these highly scalable simulations is ls1 mardyn, which models the interaction of molecules. It was created as part of the work for [Buc10]. While efforts were made to optimize the processor communication and the performance of the serial calculations, the core of the k-d-tree based load balancing algorithm was not changed much since its initial release. It is solely based on a heuristic function based solely on the number of particles. Even though it is quite limited it was used to also support heterogeneous hardware in the k-d-decomposition. The old load estimation was replaced with a new algorithm that takes time measurements, that are made before each simulation start, into account. This was then used to support more than one particle type in the load estimation, which was not previously done. Support for heterogeneous systems was also re enabled since the old algorithm could not easily be modified to use the time values. The k-d-tree load balancing algorithm itself was also modified, which allowed it to choose the splitting planes needed for the algorithm more freely.

This thesis will first give a short overview over the general theory behind molecular simulations, with a look at the needed calculations interactions and general ideas for balancing the resulting load on many processors. Then in chapter 3 this thesis will go into more detail about the implementation of ls1 mardyn, followed in chapter 4 by the improvements to the existing load balancing algorithm that were made as part of this work and their performance effects. The thesis will end with a short conclusion and a chapter with ideas for future improvements.

2 Molecular Simulations

2.1 Force Calculation

As described in [Buc10] the basis of every molecular simulation is the n-body problem, which describes all simulations where forces between many particles have to be calculated. Since these molecules can become quite complex it is advisable to reduce calculation times by reducing the force exerting parts of the particles to certain force centers, which don't necessarily correspond to single atoms. In the so called soft-sphere models they define continuous potentials. The most important one is the Lennard-Jones potential which was first described in [Jon24a] and [Jon24b] and today is often given in the following form:

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

$-\epsilon$ is the value of the minimal potential, σ the distance to the center where the potential is zero, where the attractive and repulsive forces are equal. r_{ij} is the distance between the i^{th} and j^{th} particle. Examples for Lennard-Jones-Potential graphs for different values of ϵ and σ can be seen in Figure 2.1. This potential models the interaction of the attracting and repulsive forces of two normal particles. The positive term models the repulsive forces, that are exerted by overlapping electron hulls of two different molecules. The negative term models the attractive van-der-Waals potentials. The Lennard-Jones-Potential alone isn't enough to model particle interactions. More complex molecules also contain dipoles and quadrupoles, which have to be taken into account. This is still not enough, since these two only represent unequal electron distributions in overall neutral molecules, so additionally real charges (additional/ missing electrons) have to modelled as well. With these potentials the forces and the resulting movements of the particles can be calculated with differential equations, which cannot be solved analytically, but need to be discretized. This can for example with the Störmer-Verlet method described in [Ver67]. Normally it is enough to only calculate pairwise interactions between the the different potentials. This means that there are $n \cdot (n-1) \in O(n^2)$ Molecule interactions to consider. This is still quite a lot, especially for large systems, but the structure of the Lennard-Jones potential allows for

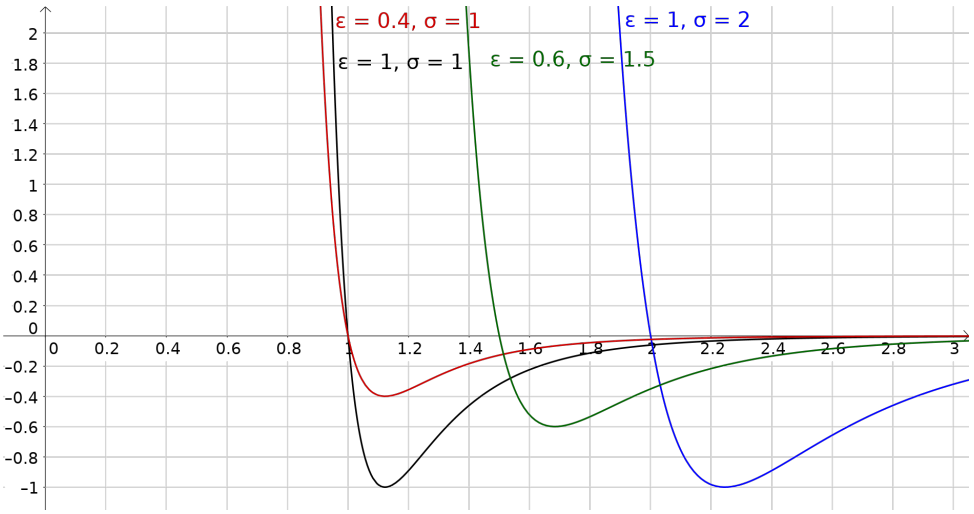

Figure 2.1: The Lennard-Jones potential for different values of ϵ and σ

a simple optimization. It can be easily seen that the resulting forces of the potential go quickly to zero if the molecule distance gets bigger. To exploit this fact one can simply stop calculating the interactions if two molecules are further apart than a certain defined distance, called the cut-off radius. This is used in the linked cells data structure, where the simulation domain is split into cubic cells with a side length that is greater or equal to the cut-off radius (see Figure 2.2). This means that a particle can only interact with the particles in its cell and the neighbouring cells that share at least a corner with its cell. Furthermore even if a particle is in a neighbouring cell, the forces need to be only calculated if it is actually in the cutoff radius, otherwise only the distance between the two molecules needs to be calculated (see Figure 2.2). To calculate all forces in a simulation it is necessary to iterate over all cells. For every molecule in such a cell first the interactions with the other molecules in the same cell (further called *inner cell interactions*) and then the interactions with the molecules in neighbouring cells have to be calculated (further called *neighbour interactions*). If one would calculate the latter for all particles in all neighbours, it would lead to unnecessary calculations. Newton's Third Law states, that if an object exerts a force on another one, that the object itself experiences a force equal in magnitude and in the opposite direction of the exerted force. That means for every particle pair you only need to calculate one force, because the other one can be directly calculated from it. This is easy to do for inner cell interactions, because you simple iterate over all pairs only once, but for neighbour interactions this is not as easy. Here it has to be guaranteed that for each cell the interactions are only calculated between it and half of its neighbours and not

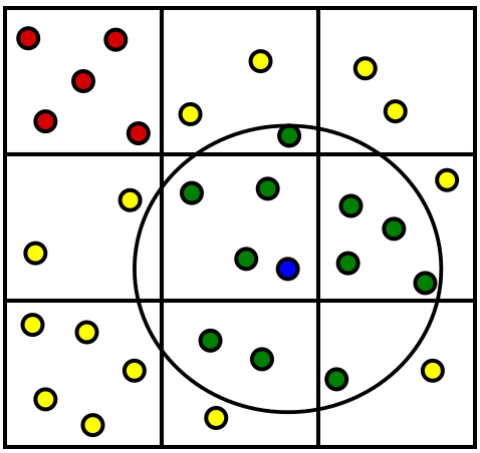

Figure 2.2: A small 2D linked cells structure. For each molecule the interactions with all molecules inside of the cutoff radius have to be calculated. If a part of the cells is inside of the cutoff radius, then for all particles at least the distances have to be calculated. Particles in cells that are completely outside of the cutoff radius can be ignored.

all of them. An example of how that looks can be seen in Figure 2.3. Since simulations domains cannot be infinite, an important design decision that has to be made, is how to deal with the border of the simulation area. One idea would be to let the particles bounce back from the wall, while another would that particles that leave the simulation on one side come back in on the other side, which can be achieved by simply linking the cells on opposite simulation boundaries. This creates a periodic domain.

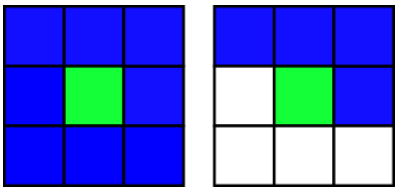

Figure 2.3: While naive method (left) calculates all interactions with all neighbours, the better method on the left only calculates the interactions with half of neighbours (in blue) of the currently iterated cell (green), so that double force calculations are avoided.

2.2 Parallelization & Load Balancing

2.2.1 General

As already stated in the introduction, molecular simulations can get relatively big and complex, so it is advisable to distribute the calculations on multiple processors. The linked-cells data structure offers a relatively easy way to achieve this, because the load can be distributed by distributing the cells in a way that partitions the simulation domain. As long as a cell only borders cells that are also handled by the same processor, the force calculations can be computed as in the sequential case. The problems arise, when forces between particles that are stored on different processors have to be calculated. The simplest way to do this would be for each processor to store a so called halo, which contains all cells of other processors that are neighbours of cells owned by the processor in question (see Figure 2.4). Then to calculate the neighbor

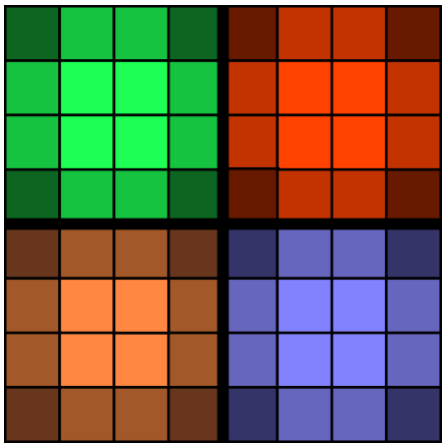

Figure 2.4: Simple linked cell structure with periodic boundary conditions divided between four processors (indicate by the different colours). If no forces are exchanged between the processors, the darkest cells of each colour have to be shared by four processors, the medium dark cells by two and the lightest cells are only stored on one processor.

interactions in border cells this halo information can be used. No forces are calculated for particles in the halo, since this is done by the processor who shared the particles with the other processors. It is important to note that this approach does not make use of Newton's third law, so the neighbour interactions across processor boundaries have to be calculated twice. To avoid these redundant interactions it is possible to not store the full halo but only half of it (similar to the way it was done for linked cells). Then only one of the involved process calculates the boundary crossing forces but these have

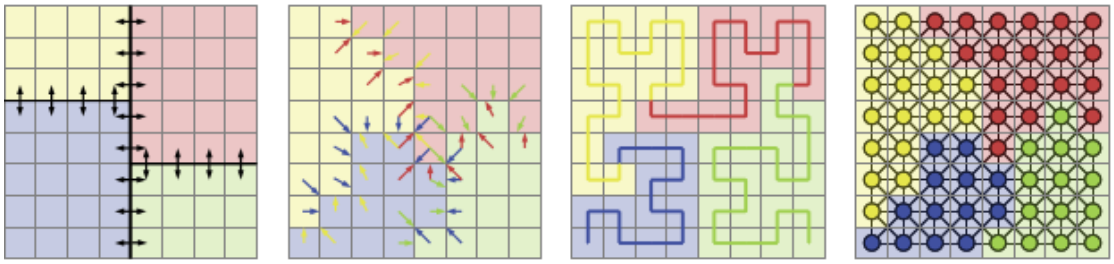

Figure 2.5: The four different presented load balancing algorithms, k-d-decomposition, diffusion, space filling curves and graph based algorithms; image taken from [Buc10]

to be communicated to the second involved process. For further details one can look at [Bro+93]. In both cases it is advisable to keep the surfaces of the processor areas small, because bigger surfaces lead to more communication overhead in both cases and to more calculation load (from the unecessary double calculations) in the former case. The sum of the communication cost and cost of the additional calculations (if there are any) will be further called separation costs. An example for a good cell distribution, which keeps the surfaces of the processor areas as small as possible, would be a decomposition made out of cubes or at least cuboids that are as close as possible to cubes. These have the best volume to surface ratio (meaning more volume per surface area). It is easy to see, that such a distribution still isn't great, if it doesn't take the molecule distribution into account, when heterogeneous molecule systems are considered. Here the density between two parts of the simulation can differ by a lot, so it might be possible that only a few processors get high density areas while the rest get only low density areas. The few processors would then heavily slow down the simulation. So somehow it has to be ensured that the load of the processors is as equal as possible, which in a dynamic simulation can only be achieved by dynamic algorithms. Generally one can divide these solutions into global and local algorithms. Global algorithms store a central partitioning data structure, while local algorithms store as little information as possible and only exchange load with their neighbours. The former are generally more expensive so they should be used less often than the local algorithms, but since they have access to information about the whole domain, they can theoretically create better partitions. For the algorithms to be able to partition the domain so that every processor gets the same load there needs to be some kind of load estimation for each cell, which estimates the time a processor needs to iterate over it. This can be a simple heuristic function which is solely on the amount of molecules in the cell (and its neighbors), but it can also make use of time measurements of previous iterations. If not otherwise mentioned the information in this section is based on [Buc10].

2.2.2 Diffusion

Diffusion is a completely local algorithm. The idea is that every processor only knows its dynamically changing part of the complete domain. If two neighbouring processors take a different amount of time for their respective area, some of the border cells can be exchanged to equalize the load. This exchange needs to take differences in separation cost between the old and the new decomposition into account. These stem from the fact that every exchange generates a new border, where old border cells are now completely inside of one process, while other cells become new border cells and this of course also changes the costs. One way to handle diffusion is the so called sender initiated exchange. If a processor detects that it has to much load, the border cells, that neighbour processors with less load, are sorted, so that the cells with the highest expected load reduction (the sum of the saved separation cost and the lost force calculations) are sent to the neighbouring processes until the load of the sending processor is equalized. In the receiver initiated exchange the receiver is the main actor. Of course you somehow need to define a initial decomposition. This can either be simply a static, molecule unaware decomposition, since the load differences should be amended relatively quickly or one of the other presented load balancing algorithms could be used. This doesn't necessarily only need to happen at the beginning of the simulation. It is also possible to repeat this process, so that in-between the global decomposition the diffusion algorithm ensures a good load distribution, which then makes up a hybrid approach. This idea was described in [HPG16]. A big problem with this approach are the dynamically changing processor areas, that are not as clearly defined as in the other algorithms. This means that this this approach takes a lot of space to represent these areas in memory and it takes more time to iterate over the cells. A normal way to store the owned cells would be a octtree. A big advantage of this approach is that it can run on heavily heterogeneous architectures, without needing changes in its basic algorithm because the load calculation is only based on the time the processors needed and not on their theoretical performance.

2.2.3 Space-Filling Curves

As their name suggests, space filling curves can be used to linearise multidimensional space, so that each point in space (here each discrete cell) can be bijectively mapped to a point on a curve. This is very useful for load balancing, since it effectively reduces the dimensions of the problem from three to one. To distribute the load, each processor gets a continuous part of the curve. This ensures the locality of the cells and also requires little memory, since the processor only stores the begin and end of its domain on the curve as two indices. With these two values the actual the function that defines

the curve is enough to calculate the owned part of the domain. If you assume that each cell has an estimated load associated with it, it becomes relatively easy to distribute the processors. A version of how to do this in parallel was described in [HPG16]. If the rebalancing is done often enough it can happen that only processors that are neighbours on the curve actually need to exchange cells, which makes space filling curves a hybrid between local and global algorithms. Space filling curves can only fill finite quadratic/ cube-shaped linked-cell structures with sides lengths that are powers of two. To use the for all cuboid structures, one fills it up to the next full cube with power of two side lengths, which means the curve goes through "empty space". This space can be either filled with empty cells or be ignored in a more efficient way. The biggest downside of space filling curves are their restrictive nature, which might not allow close to optimal load distributions.

2.2.4 Graph-Based Structures

The graph based approach differs from the other algorithms because here the load balancing problem is reduced to another seemingly, completely different problem: the graph partitioning problem. One version of it goes as follows: given a graph with node and edge weights and the number of partitions k, partition the graph in such a way, so that the number of edges that cross partitions is minimal, while each the sum of the node weights in each partition is as equal as possible. This is a NP-hard problem. The first idea that it would make sense to model the cells as graph nodes is probably also the best. The edges connect the nodes corresponding to neighboring cells and the edge weights represent the separation costs. Concretely this could be represented by the number of separated particle pairs. The node weights contain the load estimation. The problem with this approach for general linked-cells-based that while the separation costs are minimized globally, it might be unevenly distributed on the different processors. That this method can still lead to good despite of this problem was shown in [HPG16]. A framework which uses these structures is ParMETIS which is partly described in [SKK00].

2.2.5 k-d-Tree

A k-d-tree is a geometric tree structure. It recursively splits the domain into two halves separated by an axis-aligned hyperplane. In the beginning there is only one node which encompasses the whole domain. To use this structure for load balancing, a list of all processors is assigned to this node. Now the domain is split, so that the load of the domain is roughly halved. This creates two child nodes, where each of them gets half of the processors of the parent node. In every step each of the processors calculates

the split of the node it is assigned to. This is done recursively until each node contains only one processor. These are the leaves of the tree. If there are still a lot of processors left, it is not necessary to half the load, it would simply suffice to split the domain anywhere and then assign different numbers of processors to each half, so that the load per processor remains roughly the same. Additionally if there are only a few processors left it might become feasible to test every possible splitting plane, since there could be others apart from the load halving one, that lead to a better distribution of processors, where the loads per processor are as equal as possible. In each recursion level it has to be decided in which dimension the tree is split. If this is too expensive there are two possibilities: Either switch dimensions after each split, so that the domain is split along all of the axes in a rotating fashion or simply always split along the axis, along which the domain is biggest. The latter is probably better since the resulting nodes should be more cubic than in the former case, which reduces the overall separation costs.

As in the other algorithms it could be quite problematic, if the hyperplanes always cut through high density areas. To avoid this it would be possible to add the separation costs to the cell loads. The problem here is that a cut in an earlier recursion level might require that there are cuts through high density areas in later recursion levels. Either one can ignore this completely and hope that a good cut in a higher dimension, doesn't lead to problems later on or one can fully calculate the decomposition in every recursion step for every candidate cut, which means that for every cut the best decomposition is calculated. This is of course problematic if there are a lot of possible cuts in each recursion level, because the the number of operations scales exponentially in this amount but it would also generate a perfect k-d-decomposition. A compromise between the two options is to implement a processor threshold. Only if a cell contains less cells than this thresholds the full search is activated.

3 MarDyn

3.1 General

As mentioned MarDyn is a relatively big framework used solely for molecular simulations. Its initial state was extensively described in [Buc10] though since then a variety of changes were made. It uses a linked cell data structure and every processor stores the full halo of its area, which means there are redundant force calculations (as mentioned in the previous section). The domain is normally periodic. The domain decomposition algorithm is a dependency of the simulation and can be relatively easily replaced. The parallelization strategy for this work was solely based on mpi even though there were some efforts made to include OpenMp parallelization. The framework also supports extensive output for example time measurements of the whole simulation and for each processor. The processor times measured per processor are split into the force calculation time and the communication time. The later also includes the time that is needed to create the domain decompositions. MarDyn also supports vectorization of the force calculations. MarDyn is generally very modular and under constant development, but the above mentioned attributes describes the configuration that is assumed for this work.

3.2 Load Balancing

3.2.1 Default Domain Decomposition

MarDyn supports a standard, static domain decomposition, which divides the simulation area in equally sized cuboids which are assigned to the processors. The size of these cuboids only depends on the domain size and not on the distribution of the molecules. Since this is the most basic domain decomposition that also doesn't require any rebalancing it is used as baseline for the performance tests.

3.2.2 k-d-Tree Decomposition

The main decomposition algorithm is an implementation of a k-d-tree, even though it doesn't fully implement it in the way described in the previous chapter. For ex-

ample it turned out, that the communication cost that comes from the partitioning of the cells is negligible so it isn't calculated. One of the reasons for that is because MarDyn uses the so called overlapping communication strategy mentioned shortly in [Nie12], where asynchronous communication used to minimize the communication cost. The processors initiate the communication and while the data is transferred over the network they calculate the inner cell particle interactions. If they are done they check if the communication is already finished and if that's the case they continue with their calculations. Only when the communication actually takes longer than the force calculation there is a measurable performance degradation because of the communication. Even though it uses a scheme where only particles and not the forces between border cells are communicated, the other part of the separation costs, the costs for the double calculations, aren't treated in a special way either (see the next section for more details). Despite of that the decomposition can still recursively evaluate all possible decompositions for each possible subdivision, when considering its quality. This is only used to reduce the differences in the processor loads, where in Section 2.2.5 the main motivation was to reduce the separation costs. At the beginning of the simulation there are some parameters which can be defined. The most important one is probably the number of iterations between two rebalancing steps. Additionally it could be defined whether the algorithm splits tree nodes along the axis, along which the node in question is still the biggest or if it should always search in the direction of every axis. Additionally one could define the processor threshold, below which the all decompositions for all subdivisions are calculated (further called full search threshold). There is also already an extensive infrastructure for outputting decomposition and molecule data.

3.3 Cost Estimation

The cost estimation function currently used in MarDyn is pretty basic. It assumes that the time needed for calculating the forces only depends on the squared number of molecules. Let i be a certain cell, let $n(i)$ be the number of molecules in i, and $K(i)$ be the neighbours of i, then the load calculation function for each cell i is:

$$n(i)^2 + \frac{1}{2} \sum_{d \in K(i)} n(i) * n(d) \tag{3.1}$$

This calculation is a heuristic estimation for the the needed force calculations. There are at least two possibilities of how to interpret this equation. In [Nie+14] it was said that a slightly modified version, where the first term was multiplied by $\frac{1}{2}$ and not by 1, should represent the number of distance calculations. Another interpretation and

which is further assumed in this paper uses the exact version of Equation 3.1 and it was mentioned in [Sec+16]. Here the $\frac{1}{2}$ represents the fact, that for every molecule in a cell, some of the molecules in the neighbouring cell will be outside of their cut off radius. This means that only the distance between the two particles, but no forces have to be calculated, which as mentioned is normally cheaper than the force calculations. This seems to be preferable to the first interpretation, since for bigger molecules the force calculations are far more expensive than the distance calculations. Still this heuristic is overly simplistic since for one it doesn't take into account, whether a neighbouring cell shares a face, an edge or only a corner. In the latter two cases there are generally a lot less particles inside of the cutoff radii of particles in the original cell. This can be seen in the previous Figure 2.2, where the blue particle is close to the corner but the circle defined by the cutoff radius still covers more area of its origin cell and its edge neighbours than of the corner neighbours. The heuristic also doesn't take into account that for cells inside of a processor area particle interactions are only calculated once for both involved particles as described in the previous chapter. The implementation technically didn't follow either model, because here $i \in K(i)$, which meant that the interactions inside of a cell were counted one and a half times. Since this behaviour was documented nowhere, it was assumed to be a bug. It turned out it didn't really affect the quality of the load balancing algorithm. Despite of that, the bug was fixed for the measurements later in this thesis.

3.4 Existing Support for Heterogeneous Hardware

The load balancing algorithm of MarDyn already supports heterogeneous hardware. This was a somewhat recent addition described in [Sec+16]. To make the support of arbitrary heterogeneous hardware easier it is necessary to define a constant ratio between the performance of the different processors. This is done by measuring the performance of each processor. If the domain should be split at a load ratio of x%, then the processors have to be assigned in such a way, that the quotient of the sum of the performances of one side and the sum of the performances on the other side is as close as possible to x%. The performance measurements can either be done beforehand and simply be provided to the simulation at the start or they can be calculated dynamically. Only measuring the time in the dynamic case isn't enough, since this would mean that a faster processor with more load is as fast as a slower processor with less load. So somehow the load has to be taken into account, and this was done here with a FLOP counter, so the performance was measured in FLOPs per second. But even then the dynamic version still has a problem: It can be generally observed that the performance of a processor that has only a few particles is lower than that of the

same processor with more particles. This can lead to a negative feedback loop where processors with few particles to begin with, have slower flop rates than they would have with more load, so they get even less particles, which makes them even slower, and so on. It is important to note that this effect is more dominant in structures with very high differences in performance, which means that the slower processors already get only very few particles in the initial decomposition. So it is better to calculate the performance in the beginning. Since the initial performance is still only transformed into a constant, the lower FLOP rate in a processor with only a few particles is not taken into account.

4 Improvements to the Existing Load Balancing Algorithm

4.1 General

The main focus of this work lay on improvements of the load estimation though some improvements were made to the general algorithm as well. For the load calculation function it can be observed that it would be perfect if the load l and the time t it actually took to do the calculations in an area are directly proportional, meaning that there is some constant c so that $t = c \cdot l$. This is because the algorithm would obviously be perfect if the load estimation function would actually return the time needed and because the algorithm would work the same way, if you replaced the load values l with $k = \frac{l}{c}$. As mentioned before this only relates to the actual calculation costs, the separation costs are not modelled in the load calculation function found in MarDyn and this was also not changed in the work for this paper.

Used Scenarios

For the benchmarks of the improvements several scenarios were used. One was already given and the others were created with the help of so called scenario generators. The given scenario was a domain filled with clusters made of Ethane (as can be seen in Figure 4.1). This is also the smallest scenario used with only 512,000 molecules and roughly 13,000 cells. It is further labelled *ethan*. The first generator generates a sphere of higher density inside of an otherwise homogeneous lower density domain (as seen in Figure 4.1). With it two concrete scenarios were created, one where the sphere is directly in the center (*mid*) and one where it is in the corner of the simulation domain (*cor*). Both have roughly 350,000 cells and 4.2 million particles. The second generator could generate two different phases with different densities, which are separated by two axis aligned planes (as can be seen in Figure 4.1). It can be parametrized with the starting point of one of the phases relative to the size of the full domain along an axis and the size of the phase, also relative to the domain length along the same axis. The names of the concrete scenarios generated contain these two parameters in their name, so in the scenario *40_50* the first phase starts at 40% of the length along the y axis and

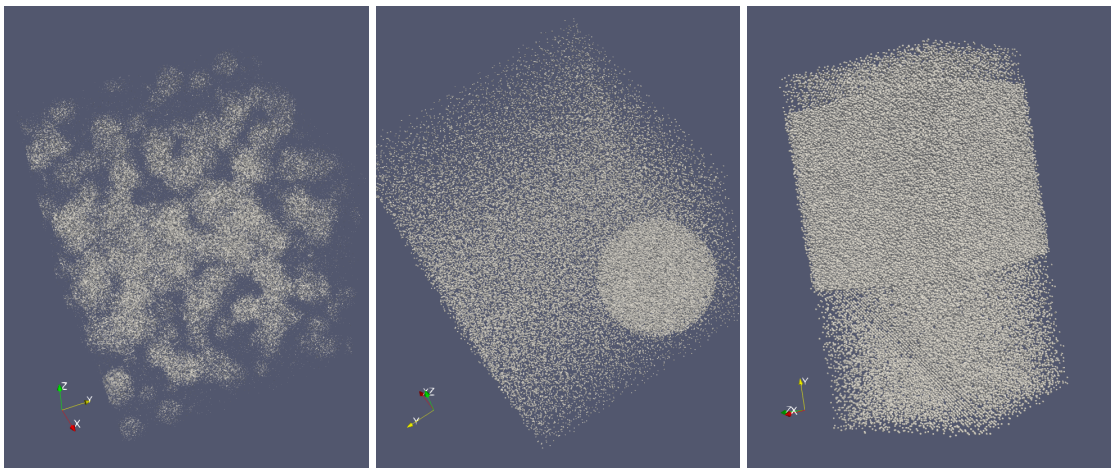

Figure 4.1: The scenarios *ethan*, *cor* and *40_50*

goes to $90\% = 40\% + 50\%$ of the domain length (see Figure 4.1). The other phase would then go from 90% to 100% and from 0% to 40%. Several scenarios were generated with it and they all contain ca. 280 million particles in around 65,000 cells. As part of this work this generator was also improved so that the two phases can contain different molecule types (but still only one type per phase). Only one scenario was generated with this improved version, since the generator was generally not suited for molecule types that consisted of more than one potential center. Since it doesn't take the sizes of the molecules into account, it could happen that these molecules were generated too close to each other which would lead to problems in the simulation. The generated scenario is further labelled *2mol* and it contained more than 5 million molecules in 160,000 cells. There were three molecule types used, Argon, Ethane and Cyclohexane. Argon is the easiest molecule with only one Lennard-Jones-Center and it is the molecule present in the generated simulations with only a single molecule type. Ethane is a bit more complicated but still consists of only two Lennard-Jones-Centers and it is only present in the *ethan* scenario. Cyclohexane was used as the second type (in addition to Argon) in *2mol*, since it is far more complex, with 6 Lennard-Jones-Potentials and one Quadrupole.

Result Format

For the representation of the results up to four times will be given in the result tables: the total simulation time, the maximum force calculation time, the minimum force calculation time and the average waiting time. The total simulation time is the time needed in the main loop of the program. This excludes the pre-processing (things like

loading the molecule information, and the time needed for the vectorization tuner) and post-processing, which is mostly information output. The force calculation time of a processor is the time it needed for calculating all of its interactions. The waiting time of a processor is the difference between its calculation time and the maximum calculation time. The average can be calculated by subtracting the average calculation time from the maximum calculation time and it is a measure for the maximum possible performance improvement. This can be illustrated with an example: Assume that half of the processors have to wait ten seconds, while other half all take the maximum force calculation time of twenty seconds. Then the average wait time is five seconds. To balance out the load, one could imagine that the faster processors each transfer five seconds of their calculations to the slower processors, which would mean that every processor takes the same amount of time, and the simulation overall is five seconds faster. Such a precise load transfer is of course not possible and transferring load might even incur new overhead (see separation costs), which would change the overall load. Still this thought experiment shows that the average waiting time is a rough estimate for an upper bound for the possible performance improvement.

The load balancing algorithms that already existed and are used as baselines are the standard domain decomposition (*DD*) described in Section 3.2.1, the old k-d-decomposition (*KDD*) described in Section 3.2.2 and the heterogeneous k-d-decomposition described in Section 3.4 (*hetero KDD*). The improved k-d-decomposition algorithms that were used and that are described in the following sections, are the one based on the quadratic model (*QM*), the one based on the vectorization tuner for one and two molecule types (*tuner*) and the one based on the vectorization tuner for heterogeneous systems (also for one and two molecule types) (*hetero tuner*). All of these were compiled with support for the AVX-vectorization for the force calculations implemented in MarDyn and if not otherwise mentioned they use a full-search-threshold (see Section 3.2.2) of 8 processors and a sb-threshold (see Section 4.5) of 4, which turned out to be good default values.

All mentioned times are given in seconds.

4.2 Single Molecule Type

4.2.1 General

To improve the present load estimation function in MarDyn a first idea would be to simply change the already existing formula (Equation 3.1) which was based solely on the molecule count in each cell. It would be a lot better if one could use the actual time processors need to calculate a certain amount of interactions to achieve a better performance. This is especially important for simulations with more than one molecule

type on heterogeneous hardware since an adjustment of the original formula would require run time information anyway to measure the performance difference between processors and the different calculation costs for each molecule type.

4.2.2 Quadratic Model

Without Linear Parameter

The simplest way of integrating measurements into the simulation is to look at the results of a simulation run. MarDyn already outputs the size of the area and the number of particles each process works on, when the simulation finishes. With little change it then also supported the time it needed to calculate the molecule interactions per process. Since the basic load estimation is based on (linked-)cells and not on the k-d-tree nodes, the measured node values (particle count and needed time) have to be divided by the number of cells owned by that process, so that one gets at least an average value for the cells. The simplest model, that goes beyond the original model described in Equation 3.1, is to assume that every cell needs an additional constant time to be iterated over, so there is already a load c associated with an empty cell. This can be represented by following equation (base on Equation 3.1, which is further called quadratic model:

$$c + n(i)^2 + \frac{1}{2} \sum_{d \in K(i)} n(i) * n(d) \tag{4.1}$$

This assumption was supported by the fact, that in the old decomposition processors that got bigger domain areas took longer than the their load suggested. c now has to be somehow determined. If l is the calculated load of a process (the sum of Equation 3.1 for all cells in the process), t the time a processor actually needed (measured after a simulation run) and #cells the number of cells in a processor, then the following equation should hold, if the new model would be perfect:

$$\frac{t}{\#cells} = m \left(\frac{l}{\#cells} + c \right) = m \frac{l}{\#cells} + m \cdot c \tag{4.2}$$

While t denotes a time, l and c represent the load. As mentioned before, if the decomposition is perfect, time and load should only differ by a constant factor, which is here represented by m. Since this equation can be calculated for every processor it made sense to use the values of all processors and use these to generate a regression line (with $c \cdot m$ and m as unknown parameters). Plotting these values showed promising results as can be seen in Figure 4.2. The results of the regressions are given in Table 4.1. The problem with this model is that it cannot easily be determined at run time (at least not

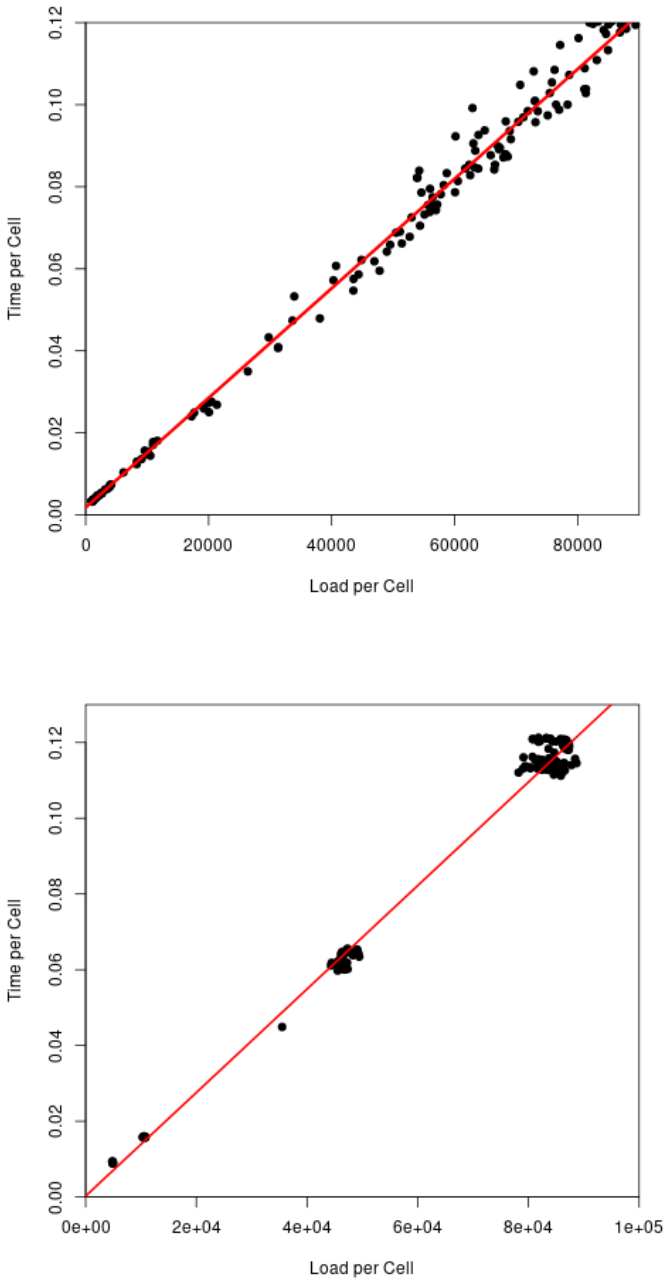

Figure 4.2: Two examples for the regression lines. The first image shows the results line for the *cor* scenario, where the regression is the most stable, while the second image shows the result for the scenario *40_50*, where the std. errors are far bigger (see Table 4.1). The interception point with the y-axis is $m \cdot c$ the slope is m.

Scenario	Procs.	$m \cdot c$	std. error	m	std. error	constant $c = \frac{mc}{m}$
mid	140	$2.42 \cdot 10^{-3}$	$1.32 \cdot 10^{-6}$	$1.32 \cdot 10^{-6}$	$4.85 \cdot 10^{-9}$	1840
cor	140	$1,81 \cdot 10^{-3}$	$5.15 \cdot 10^{-4}$	$1.34 \cdot 10^{-6}$	$8.51 \cdot 10^{-9}$	1380
25_50	112	$1.34 \cdot 10^{-3}$	$6.75 \cdot 10^{-4}$	$1.36 \cdot 10^{-6}$	$8.54 \cdot 10^{-9}$	1030
40_50	112	$2.38 \cdot 10^{-4}$	$8.64 \cdot 10^{-4}$	$1.37 \cdot 10^{-6}$	$1.19 \cdot 10^{-8}$	174
ethan	84	$1.63 \cdot 10^{-2}$	$6.27 \cdot 10^{-3}$	$2.82 \cdot 10^{-6}$	$6.63 \cdot 10^{-8}$	5800
ethan	56	$7.28 \cdot 10^{-3}$	$4.83 \cdot 10^{-3}$	$2.97 \cdot 10^{-6}$	$4.52 \cdot 10^{-8}$	2450

Table 4.1: Different results for the regression lines; the high std. errors for the c constants show, that only the result of the first regression is somewhat stable. $m \cdot c$ is a time, m is time per load and with that c is a load value.

in the current structure of the program) so the constants would have to be calculated manually for every particle-processor-combination and then provided at the start of every new run (or at compile time). Another problem that can be seen in Table 4.1 is that load constants differ wildly from scenario to scenario. $m * c$ should technically be independent from the particle type, since the constant time needed to iterate over a cell without calculating anything should be independent from the contained particle type and the conversion factor m should only be dependent on the molecule type used in the simulation, because the load time ratio only depends on the type of the used molecules. A clear explanation for why this is not the case in this table is the growing standard error in the result of the regression. This is shows the biggest weakness of this approach. The big standard error can be explained by the fact, that the averaged time results are not always as spread out as in the first graph of Figure 4.2 but are clumped together as in the second graph of the same image. For the performance evaluation still some constants had to be picked. For argon this constant was defined as 1650 and for ethan 5200. These values were chosen because they relatively close to the value in Table 4.1 with the least amount of error, but they also take into account, that the other regression results tend to lower values for c, even though their std. errors increase drastically. When used in the simulations these values also lead to very good results (see Section 4.2.4).

With Linear Parameter

Since the simulation also contains operations that are linear in the amount of particles per cell (for example the velocity update with the calculated forces in each iteration), it was only natural to add a parameter that is linear in the number of molecules to the above model. So p in the following equation is the amount of particles in a processor area, while n is the new unknown constant converting the load associated with each

particle into time. The other constants are defined as above.

$$c + p \cdot n(i) + n(i)^2 + \frac{1}{2} \sum_{d \in K(i)} n(i) * n(d) \tag{4.3}$$

The equation for calculating p and c is the following, when o is the number of particles for the process in question and the rest of the variables are defined as in the previous section:

$$\frac{t}{\#cells} = m \left(\frac{l}{\#cells} + p \frac{o}{\#cells} + c \right) = m \frac{l}{\#cells} + m \cdot p \frac{o}{\#cells} + m \cdot c \tag{4.4}$$

While at first sight it might seem that this equation can model the reality in a better way, if one looks at figure 4.2, the previous model was already pretty good. The big standard errors are probably a result of the bad sample points for the regression and not because of the bad model. So in the end the additional parameter seems unnecessary and this was supported by the results of the regression of the new model, where the results were far less stable than in the original one. It even lead to a negative cell constant c, which of course contradicts the modelled reality where a processor cannot take zero or less time for iterating over a cell. Plugged into the simulation it turned out that as expected this model gave worse results than the model without the linear parameter which is why this approach is not further discussed here.

4.2.3 Vectorization Tuner

As mentioned before the load estimation is done on a cell level, so it would also be better to use times measured at a cell level and not at a tree-node level as done in the previous section. Since starting and stopping timers incurs an overhead, it doesn't seem feasible to measure the times for each cell, while the simulation is running. A better approach would be to measure the time each processor needs to calculate the forces between a certain amount of particles in a cell before the simulation starts and use them as it goes on. MarDyn mostly implemented this functionality already in a class called vectorization tuner. It can generate cells which can be placed arbitrarily in a simulation area. These cells can be filled with an uniformly randomly distributed molecules of different types. Then the tuner can calculate the inner interactions for a single cell or the neighbour interactions for two different cells (not both at the same time) repeatedly for a given amount of iterations and return the time needed. This was used to generate four different time profiles, which contain these measurements for different amounts of particles in the cells. One contains the measured times for the inner cell interactions, one for the interactions between two neighbours that share a face, one for two neighbours that share an edge and one for neighbours that share a

corner. It is important to differentiate these, since if you assume that all molecules are uniformly randomly distributed[1] in the domain, it is more likely for a certain molecule to interact with molecules in a neighbour that shares a full face instead of only an edge or a corner. This can be seen in Figure 2.2 for the two dimensional case, where the blue molecule is close to the corner but its cutoff radius still encompasses more space of its own cell and its edge neighbours than of its corner neighbours. This means it normally also takes less time to calculate the interactions with a face neighbour than an edge neighbour with the same amount of molecules, which makes the separation necessary. The difference in time can be seen in Figure 4.3, where the different tuner profiles are compared.

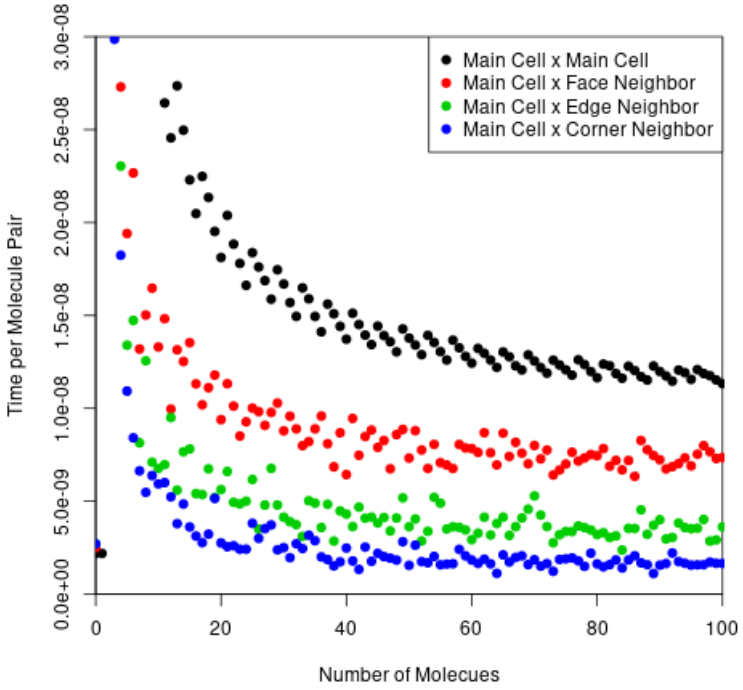

Figure 4.3: A typical tuner profile for Argon on the Intel Xeon E5-2697 v3 for all different neighbour types. The 'zig-zag' in the black measurements is a result of the vectorisation of the calculations.

For the inner interactions it is clear that a sensible time profile should consist of the time a processor needs for every amount of particles from 0 to a certain amount x. x

[1] This is the only sensible assumption if nothing is known about the particle distribution beforehand.

should be bigger than the maximum number of particles in a single cell in the whole simulation, so that for every possible molecule count a time is stored. For the neighbour interactions a similar idea would be to measure the time for every possible combination of molecule amounts less than x in the two cells. This of course takes quadratic space and even worse quadratic time. To avoid this overhead the implementation reduces the case where two neighbouring cells have particle counts x and y, with $x \neq y$ to a case where both have the same particle count by assuming that both have $\lfloor \sqrt{x * y} \rfloor$ particles. If x is roughly y, this should never be a problematic assumption, but even if this is not the case it should also work, at least if a smooth particle distribution is assumed. It can be observed that the maximum number of possible interactions between neighbouring cells is $x \cdot y$, which is of course roughly $(\lfloor \sqrt{x * y} \rfloor)^2$, the amount of interactions implied by the simplification.

So the resulting profiles can then be stored in four different, single dimensional arrays. When the load needs to be calculated for the inner cell interactions, the load value is simply the entry with the same index as the number of particles in the cell. For the neighbour profiles the above mentioned simplification ($\lfloor \sqrt{x * y} \rfloor$) is used as a look up index. As the number of interactions and with that the time the vectorization tuner needs to measure these values increases quadratically in the number of particles, it doesn't make sense to calculate the times for too many particles, especially if there are no cells that have that many particles in the entire simulation domain. Since this number is generally not known before the simulation, an out of bounds access on the arrays has to somehow be avoided. The simplest but not the best way of doing that, is to cap the number of particles, meaning that if there would be an out of bounds access the largest stored value is returned. This is only acceptable if this value is never exceeded by a large margin and/or if this doesn't happen too many times in the simulation. A better solution can be found when looking again at the graphs of the time profiles (Figure 4.3). It seems that time needed per (unordered) molecule pair converges to a constant c, when the number of molecules/molecule pairs increases. The overall time t needed for a certain amount of particles is extracted from the profiles, by calculating the median of the last 10% of the measured values. To calculate the constants from it the following equations can be used: Let x be the index where t was found in the array, which is equal to the number of particles in the main cell (and its neighbour). Then if the model would be perfect, the relation between t and c can defined as follows:

$$t_{inner} = c_{inner} \frac{x(x-1)}{2} \tag{4.5}$$

$$t_{neigh} = c_{neigh} cxy \tag{4.6}$$

To calculate c one simple solve the above equations for it. The factors one the right side are the number of possible interactions/the number of (unordered) molecule pairs. If

one now wants to extrapolate load values, the same equations can be used to calculate the now unknown t, which represents the load in this case, by using the number of particles, that made the extrapolation necessary, (or the simplification for two cells) and the previously calculated c. It is important to note, that c_{neigh} can differ for different neighbour types, so in the end one constant is needed for each of the four profiles.

4.2.4 Performance Results

Scen. (Procs.)	DD	KDD	QM	tuner
mid (196)	193/157/138	81.9/46.4/25.7	46.2/25.7/5.19	43.4/25.8/5.23
cor (196)	199/163/147	75.6/39.1/21.8	41.1/23.6/6.42	41.8/23.7/6.47
25_50 (140)	76.2/53.7/26.9	57.6/34.4/6.6	53.2/33.6/6.09	52.9/33.7/6.06
ethan (84)	67.7/62.6/28.2	56.8/47.9/11.8	57.1/47.6/11.4	56.6/47.8/11.4

Table 4.2: Times measured for simulations with 300 iterations and only one rebalancing step at the start in the format *total simulation time/maximal force calculation time/average wait time*

A comparison of performance results for the previously mentioned models can be found in Table 4.2. It shows that the quadratic model can improve the performance in some scenarios by roughly 45%. In others not much changed, though as the average wait time indicates, the old k-d-decomposition was already so good that there wasn't much room for improvement. This was the case when there weren't any processors that had an area that was much bigger than the other processor areas. In all cases the result of the quadratic model and the vectorization tuner are pretty much identical, which is quite surprising, considering the simplicity of the model. Even in this case the vectorization tuner is always preferable, because the model was too error prone in some cases and also because the tuner is easier to adapt for heterogeneous systems and multiple molecule types.

Until now the vectorization tuner calculated enough values, so that the array corresponding array was always bigger than the maximum number of molecules in a cell. That means that no load values had to be extrapolated. This is normally quite feasible since the the calculation for 0 to 350 Argon particles takes less than 3 seconds. Additionally these profiles can be saved and reused between two simulation runs, though it should be ensured, that the same molecule types are used and that the simulations are run on the same hardware. Still it is quite interesting to look at the results when some values have to be extrapolated. As can be seen in Figure 4.3, it is clear that below 50 molecules the graph didn't yet converge to a constant, so it wouldn't make sense to calculate less values than that. For this measurement it is also important to look at

how many particles were in the cells, so that it can be seen whether extrapolations were even needed. The results are shown in Table 4.3.

Scen. (Procs.)	max. Mols.	50	75	100
cor (196)	129	45.9/23.5/6.18	41.5/23.5/6.28	40.9/23.6/6.32
mid (196)	118	52.4/26.6/5.82	48.2/25.9/5.23	46.3/26.0/5.41
25_50 (140)	86	55.3/33.6/5.91	53.6/33.3/5.56	53.3/33.7/5.99
ethan512 (56)	255	80.7/68.9/14.8	80.8/68.9/14.9	81.1/69.0/15.0

Table 4.3: Times measured for simulations with 300 iterations and only one rebalancing step at the start in the format *total simulation time/maximum force calculation time/average wait time*. The column 'max. Mols.' denotes the maximum number of molecules in a cell for all processor. In all scenarios nearly all processors owned at least one cell which came close to this amount. The numbers in the following columns denote the maximum molecule count for which the tuner values were measured (so for bigger molecule counts they had to be extrapolated)

As can be seen the results for only using profiles up to 50 molecules are already pretty good. For the *cor* and *mid* scenario the speed up was minimal. Also for the scenarios with even more molecules per cell there wasn't even a detectable speedup, which can probably be partly explained by the fact, that their k-d-tree structures are relatively stable, meaning they change only little, when the load balancing values change slightly.

These test cases used only few hundred iterations with only one rebalancing step, but in a real scenario the simulation should probably run for a few thousand iterations and since the molecules can move quite a bit during that time, the tree should also be rebalanced quite a few times. To see if that would affect the speedup, additional benchmarks were created, where the simulation ran for 5000 iterations. Here it turned out that all of the used scenarios were pretty static, so the runs with less than 1000 iterations between iterations where a bit slower than simulations with 1000 iterations between rebalancing steps. This can be seen in Table 4.4. These results show at least, that the rebalancing steps don't take a lot of time because even an 50-fold increase in total rebalancing steps only increased the total running time by a small amount.

4.3 Multiple Molecule Types

4.3.1 General

While the overall simulation already supported more than one molecule type, the k-d decomposition and the vectorization tuner only assumed that one type was present.

Scen. (Procs.)	it. between reb. steps	KDD	tuner
cor (196)	1000	1322/670/319	738/411/318
cor (196)	200	1322/663/320	744/403/318
cor (196)	20	1344/662/320	796/402/317
ethan (84)	1000	957/807/612	954/805/613
ethan (84)	200	974/814/614	975/804/614
ethan (84)	20	977/803/614	974/802/615

Table 4.4: Times measured for simulations with 5000 iterations in the format *total simulation time/maximal force calculation time/average force calculation time*

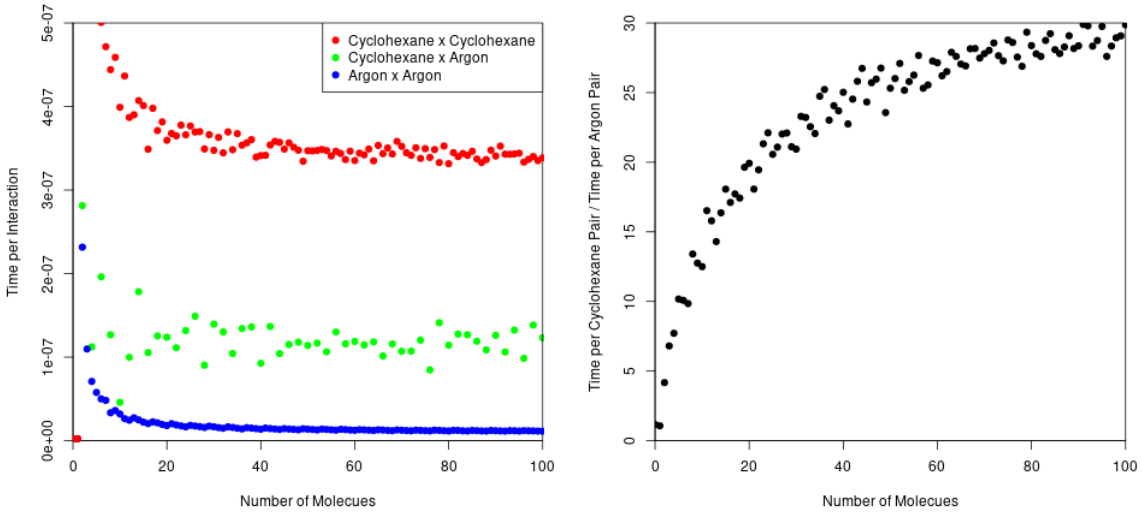

Figure 4.4: The first graph shows a tuner profile for a cell with the two molecule types Argon and Cyclohexane on the Intel Xeon E5-2697 v3 concerning only the inner cell interactions. For the values of the mixed interaction there are always as many argon molecules as Cyclohexane molecules and the number of molecules is their sum. The second graph shows the quotient of the measured times for the Cyclohexane (red) and the Argon interactions (blue).

This is of course problematic since the time needed to calculate the forces can differ heavily from particle to particle. This can be seen in Figure 4.4 where the cost of different interaction types is compared. To support more than one particle type, every cell needs to store the number of contained particles for each type instead of only storing the amount of total molecules in the cell. This means the memory requirements increase for every additional type ($O(n * m)$, where n the number of types and m is the number of cells). This gets even worse for the vectorization tuner since the number of measured times grows exponentially in the number of particle types, at least if you measure the times of every particle configuration, when there can be up to $x_i > 0$ particles in the cell for each particle type i. The measured times are then stored in an n-dimensional array, where the index in the i^{th} dimension corresponds to the number of molecules of type i in the cell. Analogous to the vectorization tuner for single molecule types it is assumed that the number of molecules of the same type is equal in two neighbouring cells. Again for the look-up if you have actually have x_i molecules of type i in one cell and y_i in the other than the look-up index in the i^{th} dimension is $(\lfloor\sqrt{x_i * y_i}\rfloor)$. The number of measured times can again be reduced by using the same strategy as mentioned in Section 4.2.3, where one uses a constant to estimate costs when the number of particles in a cell gets too big. It is important to note that if you have more than one particle types there needs to be a constant defined for every interaction involving different molecule types. If you assume that one has molecules of type A and B, then the interactions of A and A, B and A and B and B all (generally) take a different amount of time to be calculated (this can also be seen in Figure 4.4). The interactions between molecules of a single type are further called homogeneous interactions, the interactions between molecules of two types heterogeneous interactions. The total number of different interactions and with that the number of needed constants for each neighbour type is:

$$\#constants = n + \binom{n}{2} = n + \frac{1}{2}n(n-1) \tag{4.7}$$

n is the number of particle types and in the sum it represents the number of homogeneous interactions. First the constants for the homogeneous interactions c_{ii}, $i \in [n]$ have to be calculated, followed by the constants for the heterogeneous interactions c_{ij} $(= c_{ji})$, $i, j \in [n], i \neq j$. The former can be done exactly the same way as in 4.2.3, using the measured times of scenarios where only molecules of type i are present. The heterogeneous interaction constants are more problematic, since the can only occur, when there are particles of multiple types in a cell. In such a cell there are always homogeneous and heterogeneous interactions. In the simplest case where only two molecule types i and j are present, the total amount of time t_{total} can be approximated

by:

$$t_{total} \approx t_i + t_j + t_{ij} \tag{4.8}$$

t_i (t_j) is the time needed to calculate the homogeneous interactions between all molecules of type i (j) and t_{ij} is the amount of time needed for heterogeneous interactions. For simplicity other factors that increase the time are ignored. Assume that x_i (x_j) is the number of particles of type i (j) in this cell. Then t_i (t_j) can be approximated by the time the tuner needs for a cell that contains only x_i (x_j) molecules of type i (j) and nothing else. These times are further called u_i and u_j. With this approximation t_{ij} can be approximated by:

$$t_{ij} \approx t_{total} - u_i - u_j \tag{4.9}$$

To now extract the constant c_{ij} from t_{ij} one needs to look at the number of possible heterogeneous interactions (for two molecule types), again under the assumption that two neighbouring cells have the same amount of particles of each of the two types.

$$\text{Number of heterogenous inner cell interactions: } x_i x_j \tag{4.10}$$

$$\text{Number of heterogenous neighbour interactions: } 2x_i x_j \tag{4.11}$$

For the inner cell interactions all of the molecules of type i can interact with all molecules of type j. For neighbour interactions this term has to be multiplied by two, because all molecules of type i in the main cell can interact with all molecules of type j in the neighbour cell, while all molecules of type j can interact with all molecules of type i in the neighbour cell. Now we assume that t_{hetero} is directly proportional to the number of possible interactions, which c_{ij} as a conversion constant:

$$t_{ij}^{inner} \approx c_{ij}^{inner} x_i x_j \tag{4.12}$$

$$t_{ij}^{neigh} \approx c_{ij}^{neigh} 2x_i x_j \tag{4.13}$$

With these approximations and the Approximation 4.9 c_{ij} can now be calculated as:

$$c_{ij}^{inner} \approx \frac{t_{ij}^{inner}}{x_i x_j} \approx \frac{(t_{total}^{inner} - u_{ii}^{inner} - u_{jj}^{inner})}{x_i x_j} \tag{4.14}$$

$$c_{ij}^{neigh} \approx \frac{t_{ij}^{neigh}}{2x_i x_j} \approx \frac{(t_{total}^{neigh} - u_{ii}^{inner} - u_{jj}^{inner})}{2x_i x_j} \tag{4.15}$$

When all of the $c_{ij}, i, j \in [n]$ have been calculated out of bounds accesses to the multi dimensional array can be handled correctly. If you assume that the access indices where

x_i for all dimensions $i \in [n]$, then the time/load t_{extra} is extrapolated as follows:

$$t_{extra}^{inner} = \sum_{i=1}^{n} c_{ii}^{inner} \frac{x(x-1)}{2} + \sum_{i=1}^{n} \sum_{j=i+1}^{n} c_{ij}^{inner} x_i x_j \tag{4.16}$$

$$t_{extra}^{neigh} = \sum_{i=1}^{n} c_{ii}^{neigh} x_i^2 + \sum_{i=1}^{n} \sum_{j=i+1}^{n} 2c_{ij}^{neigh} x_i x_j \tag{4.17}$$

The first sum in both equations represents the time needed for the homogeneous interactions, while the second nested sum represents the time needed for all heterogeneous interactions. Even though the above model would support arbitrary many molecule types, MarDyn right now supports only two different types in the load balancing.

4.3.2 Performance Results

Scen.	Procs.	DD	KDD	tuner
2mol	252	255/236/14.8/116	281/261/14.1/143	186/149/106/29.4

Table 4.5: Times in seconds for a simulation with 300 iterations and only one rebalancing step at the start in the format *total simulation time/maximal force calculation time/minimal force calculation time/average wait time*

As explained earlier only one scenario with two different molecule types was generated. As Table 4.5 shows, the traditional load function in this case is even worse than the static decomposition. A reason for that is probably the big difference in calculation time between Argon and Cyclohexane (see Figure 4.4), which renders the load estimation useless. The vectorization tuner improves the total performance (including the communication and decomposition times) by roughly 37% compared to the static case and by 51% compared to the old k-d-decomposition. Despite of that the average waiting time indicates, that the overall simulation time can still be improved by roughly 16% even when the vectorization tuner is used.

It is important to note, that in the used benchmark the size of the two phases where equal. If the Cyclohexane molecules were replaced with the cheaper molecules like Argon, so that there would still one processor left with only Cyclohexane molecules, then the expected speedup would be a lot bigger. The reason for that is, that the total simulation time would still be roughly the same as in the scenario above, because the Cyclohexane processor would still need ca. 260 seconds, while the total load and with that the time needed in a perfect decomposition would be drastically lower. Because of the limits of the scenario generator this could not be fully exploited.

4.4 Heterogenous Hardware

4.4.1 General

The existing way of dealing with heterogeneous systems in MarDyn was already presented. To fully make use of the more detailed values the vectorization tuner provides, a simplification of the hardware structure was needed. It has to be separable into two clusters by an mpi-rank *k*, so that all processor with a lower rank than *k* are of the same type, and all processors with an equal or bigger rank than k also are of the same type. This means that only two different processor types are allowed. The decomposition splits the domain in the first recursion level in such a way, that each of the nodes gets all processors of one of the clusters as processor list. If that is the case the decomposition algorithm can work as if there weren't different clusters to begin with. So the only addition to the algorithm that is needed, is to find an initial split. Assume that there is a list of split candidates, then the cluster with the lower mpi ranks (the right cluster) calculates the loads of the right sides of all possible cuts with their vectorization tuner values, while the cluster with the higher ranks does the same for the left side. So if the right processors are slower the right side will have a generally higher load than the left side for the same amount of particles. So this is the point in the algorithm were the different processor speed come into play. Now from all split candidates the one is chosen, where the ratio between the left and the right load is closest to the ratio between the number of left and right processors.

4.4.2 Performance Results

The benchmarks that can be seen in Table 4.6 were run on cluster consisting of two different processor types, one was the SandyBrigdge-EP Xeon E5-2670 with 8 cores per socket and the other was the AMD Bulldozer Opteron 6274) with 16 cores per socket. For every physical core there was one mpi process. Even though these processors have different architectures, it could be observed (see Figure 4.5) that quotient of time it takes for one of these cores to calculate a interaction converges to a constant relatively

Scen.	Procs.	KDD	hetero KDD	hetero tuner
cor64	64/128	65.4/43.7/16.6/27.3	53.5/36.4/13.9/22.5	31.5/22.0/10.1/6.40
2mol	96/384	183/174/8.99/109	162/155/6.18/94.6	83.5/75.6/37.9/14.1

Table 4.6: Times measured in the format *total simulation time/maximal force calculation time/minimum force calculation time/average wait time*; the number of processors is given in the format *Number of Xeon Processors/ Number of Bulldozer processors*

quickly. Interestingly this constant depends on the kind of interaction (if its an inner

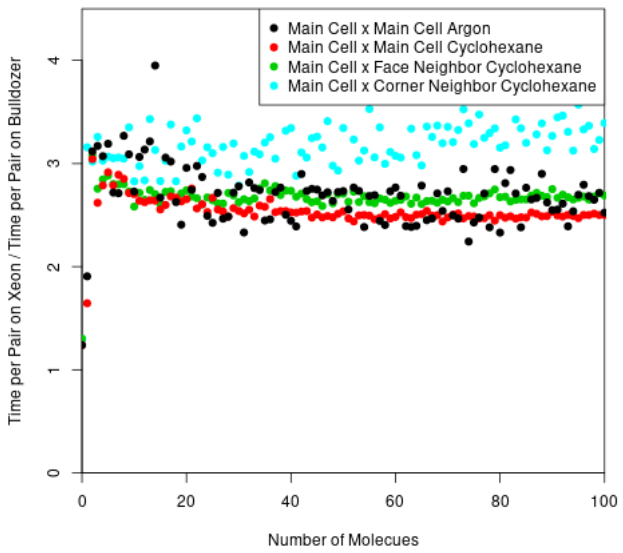

Figure 4.5: The quotient of the time needed to calculate a homogeneous interaction on a Xeon processor and on a Bulldozer for different interaction types (different neighbour types and different involved molecules)

cell interaction or an interaction with a face, edge or corner neighbour) and as it turns out the Cyclohexane results are more stable than for Argon, which makes it hard to tell whether the constants for ethan and Cyclohexane would be exactly the same. Technically the fast convergence to a constant should imply that the approach of assigning a single performance to a processor should work pretty well, but still the heterogeneous k-d-decomposition didn't fare well compared to the tuner (even in the case with only a single molecule type). One reason for that could be that it still uses the old load balancing algorithm which brings its performance down. It is important to note that the maximum improvement in performance for the algorithm that takes the heterogenous hardware into account compared to the hardware oblivious algorithm depends heavily on the performance difference between the used processors. How to calculate the maximum improvement was discussed in more detail in [Sec+16]. This means that in the presented case the comparison is not as interesting, since the processors are quite close performance wise. It would be more interesting to compare standard processors with accelerators or with special processors like the Xeon-Phi. It is also interesting to note that one can reduce the case with more than one particle type

to the same principles, since for outside viewer that only sees the total runtime of the processors, it is indiscernible if the processor performs worse than another one or if it performs the same but contains more of the expensive molecules.

4.5 Changes to the Splitting Rules of Decomposition Algorithm

4.5.1 General

As mentioned before one could previously only globally decide whether the simulation splits along the axis, where the k-d-tree node is still biggest or whether it should always search in every dimension. The latter technically should result in slightly better decompositions but the problem was that that it also created processor areas that were very thin in one direction (see Figure 4.6), which should result in higher separation costs. A better solution is to add a threshold (the sb-threshold), so that the simulation is only allowed to search in every direction if the number of processors in a node is below that threshold. This avoids the long thin nodes but it also means that the performance probably won't be heavily affected, since in the last few nodes the decomposition is already pretty determined.

4.5.2 Performance Results

As can be seen in Table 4.7 the effect of allowing a split in all directions is minimal even at higher thresholds. Even though no big performance improvement is visible, the result is also a very interesting. While it is shown that splitting the domain only along the biggest axis is sufficient most of the time, the results also clearly show that the separation costs don't have too much influence on the simulation, since otherwise higher thresholds should lead to a lower performance. It is also possible that, the slightly better load distributions are achieved, which offset the increase in separation costs. In the only case where at least a speedup of 10% in the force calculation could be measured, it can be observed that the generated nodes aren't as thin as in the other scenarios.

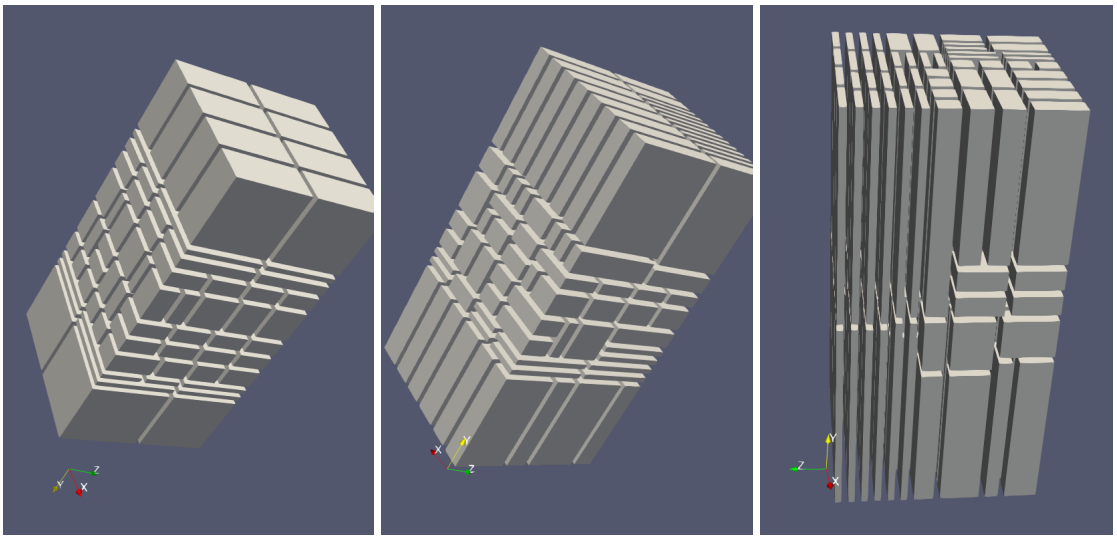

Figure 4.6: The decomposition of 25_50 at with different thresholds. In the first image every tree node is only split along the axis where it is still biggest. In the second the algorithm was allowed to split in all directions, when a node had less than 8 processors and in the last image the algorithm was allowed to always split in any direction.

Scen.	Procs.	Threshold					
		0	2	4	16	32	infinite
ethan512	56	79.5/69.4	78.1/67.7	79.5/67.4	73.8/63.2	72.7/62.8	73.2/61.8
ethan512	84	56.0/48.6	56.7/47.9	56.8/47.9	54.9/45.6	54.4/46.5	–
cor	196	41.0/22.3	40.7/20.9	41.8/23.7	41.9/20.8	42.4/20.2	–
mid	196	43.1/26.2	43.4/26.1	43.4/25.7	44.4/25.5	44.9/25.5	–
25_50	140	63.5/43.8	65.4/443	52.7/33.6	53.7/30.4	52.6/30.1	57.2/31.6

Table 4.7: Times measured for different thresholds with the vec. tuner in the format *total simulation time/maximal force calculation time*; the '–' entry indicates that there was an error in the domain decomposition, which crashed the simulation

5 Conclusion

This work presented an efficient way of integrating time measurements into the load estimation function of a standard k-d-decomposition algorithm. While it was shown that a modified heuristic can be as good performance wise, the time measurements were still more flexible and more robust.

This was shown when these measurements were used to support additional molecule types, which is hardly doable without some time measurements at some point. While the decomposition is still not perfect, it performed much better than the old, molecule type oblivious algorithms. It turned out that here a bad load estimation can be even more harmful to the running time than not doing any load estimation.

The time measurements were also used to reimplement the support for heterogeneous systems, which already existed for the old load estimation function. Despite its restrictions on the underlying hardware architecture, it showed that at least in these cases it could outperform the old decomposition for arbitrary heterogeneous systems.

Last but not least changes to the splitting algorithm of the decomposition algorithm were made. While they didn't increase the performance they showed that even k-d-decompositions with a lot of nodes with a bad volume to surface ratio can still perform well.

6 Future Work

Even though the cost estimation based on the vectorization tuner improved load balancing by quite a bit there are still some things left to do.

One of them is to smooth the vectorization tuner profiles, since currently there are quite some differences between two consecutive measurements (as can be seen in Figure 4.3). One way to achieve this would be to exchange the results of the vectorization tuner between all processes of the same type, so that every processor can calculate the average over all these values. This was already implemented but was removed again, when the support for heterogeneous architectures was introduced. Here it has to be ensured that processors of different types don't exchange their tuner data. Before that happens though, it is important to change the uniform molecule generator to use different seeds for different processors, otherwise all processors measure very similar values. Another way to smooth the vectorization tuner values would be to apply a filter which averages the values of each measurement with its neighbours.

The part of the load balancing algorithm that still needs the most changes is probably the algorithm for heterogeneous systems. The heterogeneous decomposition right now only supports two clusters of different types as descried above. This is quite hard to change since the algorithm gets more complex when more clusters are present.

A far simpler version of this algorithm could also be used for homogeneous architectures. Here the splitting planes should cut the domain in such a way, that two processors that are in the same node/island are separated as late as possible. This might improve communication times, though it is questionable whether the improvement in communication time will offset the more complex partitioning algorithm. Because of the overlapping communication this also will only help in architectures were the different island are separated by a slow connection.

It is also possible to re enable heterogeneous support for arbitrary heterogeneous structures. Since the graphs (see figure 4.5) show that at least for the processors used for this work the difference in performance converges to a constant relatively quickly , these constants can be used to adapt the already existing support for heterogeneous hardware to the vectorization tuner values. Compared to the new algorithm for the heterogeneous system support this has the disadvantage, that the tree can now cut through cluster boundaries again, though it has to be further investigated if this is really a problem with overlapping communication.

Bibliography

[Bro+93] D. Brown, J. H. Clarke, M. Okuda, and T. Yamazaki. "A domain decomposition parallelization strategy for molecular dynamics simulations on distributed memory machines." In: *Computer Physics Communications* 74.1 (1993), pp. 67–80. ISSN: 0010-4655. DOI: http://dx.doi.org/10.1016/0010-4655(93)90107-N.

[Buc10] M. Buchholz. "Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen." Dissertation. Technische Universität München, 2010.

[GZK07] M. Griebel, G. Zumbusch, and S. Knapek. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. eng. Vol. 5. Texts in Computational Science and Engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. ISBN: 978-3-540-68094-9.

[HPG16] S. Hirschmann, D. Pflüger, and C. W. Glass. "Load Balancing for Molecular Dynamics Simulations on Heterogeneous Architectures." In: *2016 IEEE 23rd International Conference on High Performance Computing Workshops*. Dec. 2016, pp. 130–141.

[Jon24a] J. E. Jones. "On the Determination of Molecular Fields. I. From the Variation of the Viscosity of a Gas with Temperature." In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 106.738 (1924), pp. 441–462. ISSN: 0950-1207. DOI: 10.1098/rspa.1924.0081. eprint: http://rspa.royalsocietypublishing.org/content/106/738/441.full.pdf.

[Jon24b] J. E. Jones. "On the Determination of Molecular Fields. II. From the Equation of State of a Gas." In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 106.738 (1924), pp. 463–477. ISSN: 0950-1207. DOI: 10.1098/rspa.1924.0082. eprint: http://rspa.royalsocietypublishing.org/content/106/738/463.full.pdf.

[Nie+14] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and M. Horsch. "ls1 mardyn: The massively parallel molecular dynamics code for large systems." In: *CoRR* abs/1408.4599 (2014).

[Nie12] C. Niethammer. “Performance evaluation and optimization of the ls1-MarDyn Molecular Dynamics code on the Cray XE6.” In: *Proceedings of the Cray User Group 2012*. 2012.

[Sec+16] S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann. “Load Balancing for Molecular Dynamics Simulations on Heterogeneous Architectures.” In: *2016 IEEE 23rd International Conference on High Performance Computing*. This work represents IEEE-copyrighted material. Dec. 2016, pp. 101–110. ISBN: 9781509054114.

[SKK00] K. Schloegel, G. Karypis, and V. Kumar. “Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning.” In: *Euro-Par 2000 Parallel Processing: 6th International Euro-Par Conference Munich, Germany, August 29 – September 1, 2000 Proceedings*. Ed. by A. Bode, T. Ludwig, W. Karl, and R. Wismüller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 296–310. ISBN: 978-3-540-44520-3. DOI: 10.1007/3-540-44520-X_39.

[Ver67] L. Verlet. “Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules.” In: *Phys. Rev.* 159 (1 July 1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98.