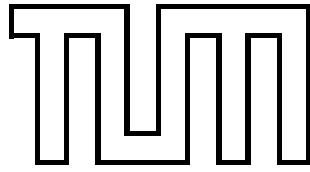


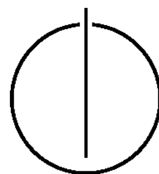

Department of Informatics

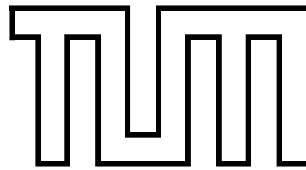
TECHNISCHE UNIVERSITÄT MÜNCHEN


Bachelor's Thesis in Informatics

# Parallel Cluster Detection in Nucleation Scenarios


Michael Obersteiner



# Department of Informatics

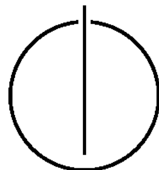
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

## Parallel Cluster Detection in Nucleation Scenarios

## Parallele Keimerkennung in Nukleationsszenarien


| | |
|---|---|
| Author: | Michael Obersteiner |
| Supervisor: | Univ.-Prof. Dr. Hans-Joachim Bungartz |
| Advisor: | M.Sc. Nikola Tchipev |
| Date: | September 15, 2014 |

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

Munich, September 24, 2014 Michael Obersteiner

# Acknowledgments

In this chapter I want to thank the people who have contributed to this thesis. First I want to express my gratitude to my supervisor Prof. Dr. Bungartz for giving me the possibility to write a bachelor thesis at the chair of Scientific Computing. Moreover I thank Wolfgang Nicka for the development of the scenario generator used in this work which is able to create MarDyn inputs for specified physical parameters, such as pressure and temperature. Finally I want to thank my advisors Wolfgang Eckhardt and Nikola Tchipev for the support I have received during the bachelor thesis. Especially Nikola Tchipev should be further mentioned as we developed together the new coarsening strategy.

# Abstract

Molecular dynamics has emerged as an important technique for simulating nucleation processes. However, an efficient implementation requires suitable parallel algorithms for the cluster detection. In this work a parallel cluster algorithm is introduced based on two depth-first searches using the geometric criterion. The first step is a local depth-first search which is followed by a new coarsening strategy based on the global graph. This coarse graph information is then processed by a master processor in a second depth-first search. In addition several performance measurements were conducted on the resulting implementation showing the good potential of the method and current drawbacks of the sequential part of the algorithm. Finally the new method is evaluated and possible improvements and extensions are illustrated which would allow a very efficient and reliable implementation.

## Zusammenfassung

In den letzten Jahrzehnten haben sich molekulardynamische Simulationen als wirksame Methode für die Vorhersage von Nukleationsszenarien herausgestellt. Für eine effiziente Umsetzung sind jedoch geeignete parallele Clustererkennungsalgorithmen nötig. In dieser Arbeit wird ein paralleler Algorithmus vorgestellt, welcher mithilfe von zwei Tiefensuchen eine effiziente Implementierung der Clustererkennung ermöglicht. Die erste lokale Tiefensuche ermittelt alle lokal ermittelbaren Clusterteile und erstellt somit eine vergröberten Graphen. Basierend auf dieser neu entwickelten Vergröberungsstrategie wird der Graph mithilfe eines "Masterprozessors" in einer zweiten Tiefensuche ausgewertet. Die Performanceanalyse zeigt das Potential der neuen Methode, jedoch werden auch die derzeitigen Probleme durch den sequentiellen Teil des Algorithmus deutlich. Weiterhin werden mögliche Erweiterungen und Lösungen für die derzeitigen Probleme aufgezeigt, welche eine sehr effiziente und zuverlässige Implementierung ermöglichen.

# Contents

**Acknowledgements** **vii**

**Abstract** **ix**

**Zusammenfassung** **xi**

**1. Introduction** **1**
1.1. Overview . . . . . 2

**2. Theory** **3**
2.1. Nucleation . . . . . 3
2.2. Molecular dynamics . . . . . 4
2.2.1. Force calculation . . . . . 5
2.2.2. Linked cell algorithm . . . . . 6
2.2.3. Periodic boundary condition . . . . . 7
2.2.4. Parallelization . . . . . 7
2.3. Clustering criteria . . . . . 9
2.3.1. Energy criterion . . . . . 10
2.3.2. Geometric criterion . . . . . 10
2.3.3. Hybrid criterion . . . . . 11
2.4. Cluster identification . . . . . 12
2.4.1. Sequential cluster identification . . . . . 12
2.4.2. Parallel cluster identification . . . . . 13
2.5. MarDyn . . . . . 15
2.6. Relation to connected component labeling . . . . . 15

**3. Implementation** **17**
3.1. Sequential cluster detection . . . . . 17
3.1.1. Implementation of the geometric criterion . . . . . 17
3.1.2. Depth-first cluster detection . . . . . 18
3.2. Parallel cluster detection . . . . . 19
3.2.1. General concept . . . . . 20
3.2.2. Evaluation of the geometric criterion in halo regions . . . . . 22
3.2.3. Local depth-first search . . . . . 23
3.2.4. Parallel cluster communication . . . . . 24

3.2.5. Global depth-first search . . . 24
3.2.6. Optimization . . . 25
3.2.7. Validation . . . 27

**4. Results** **29**
4.1. Simulation setup . . . 29
4.2. Visual verification . . . 30
4.3. Physical verification . . . 31
4.4. Scalability . . . 34
4.4.1. Strong scaling with argon . . . 34
4.4.2. Weak scaling of argon . . . 37
4.4.3. Performance analysis with ethane . . . 39
4.5. Evaluation of the clustering algorithm . . . 42
4.5.1. Problems . . . 42
4.5.2. Advantages . . . 45

**5. Summary and outlook** **47**
5.1. Summary . . . 47
5.2. Outlook . . . 47

**Appendix** **51**

**A. Source Code** **51**
A.1. Parallel cluster communication . . . 51
A.2. Local depth-first search for the parallel algorithm . . . 53
A.3. Create graph . . . 57

**Bibliography** **61**

# 1. Introduction

Nucleation is a process which can be observed everywhere in the nature. It is responsible for the formation of clouds in the sky, caused by the condensation of water in the air, as well as the crystallization process, which for example appears in supersaturated solutions. In general nucleation processes occur during the change between different thermodynamic phases, e.g vapor to liquid.

Since this spontaneous process often takes place in very short time periods (order of ns) and the nucleation sites are often very small (nm), the possibility to study nucleation via experiments is limited. For this reason numerical simulation is used to predict nucleation processes which are of high interest for industrial applications such as chemical engineering.

The challenge in identifying clusters during a numerical simulation is the detection of the phase change. A common method searches for "bounded" particles in the domain to detect a vapor to liquid change. A connected component of bounded particles forms a cluster which represents a liquid drop. The difficulty is the exact determination of the bounded particles, as there does not exist any exact physical description of a cluster.

A quantity of interest for industrial applications is the nucleation rate. This number describes the rate at which clusters form in comparison to the time and the domain volume of the experiment. Numerical simulation can be used to predict the formation of clusters and can, therefore, give an estimate for the nucleation rate.

However, one problem of the numerical simulation is the high computational effort. Therefore only very small particle numbers can be simulated. To calculate nucleation processes for bigger scenarios massive parallelization is required. Unfortunately, common cluster detection algorithms, such as depth-first search, are hard to parallelize [12].

In this work the well-established molecular dynamics framework MarDyn [8], used for chemical engineering, is extended by a clustering detection. The existing methods developed by Kible [18], Walter [28] and Schluttig [21] did not provide an efficient algorithm for parallelization, whereas the usage of the boost library [1] provided a good parallel efficiency but a significant overhead. Hence, a parallelization method based on two depth-first searches combined with a new coarsening strategy is introduced which is specially designed for the parallel cluster detection and therefore uses further application-related assumptions to improve the parallel performance of the known methods. This method is then further analysed.

## 1.1. Overview

In chapter 2 the theoretical basics of the nucleation process and numerical dynamics are introduced. Furthermore, the basic criteria for identifying bounded particles are presented and compared. Finally, an introduction to the molecular dynamic simulator MarDyn is given which we used for the molecular dynamic simulations. In chapter 3 the implementation of the cluster detection is outlined using the geometric criterion. The first section describes the basic concept for a sequential algorithm. This concept is then extended by a parallel algorithm which is described in detail in the next part of the chapter. In addition some optimizations of the whole algorithm are illustrated. In chapter 4 the results of the clustering algorithm are presented and analysed. A visual verification, a physical verification and a scalability analysis are included. Furthermore the method is evaluated and current problems and advantages are shown. In chapter 5 a conclusion is given by summing up the most important ideas and results. Finally, an outlook for future works is presented.

# 2. Theory

## 2.1. Nucleation

The nucleation process was first described in the Classic Nucleation Theory (CNT) which was introduced by Gibbs (see [17] and references therein). The theory describes the phase transition between vapor and liquid as well as between liquid and solid. In general, nucleation theory can be divided into two classes: homogeneous nucleation [5] theory which describes nucleation processes of a single chemical species, e.g. Argon, and heterogeneous nucleation (see [22] and references therein) where multiple chemical species interact. In this work, we consider the homogeneous nucleation of a vapor-to-liquid transition.

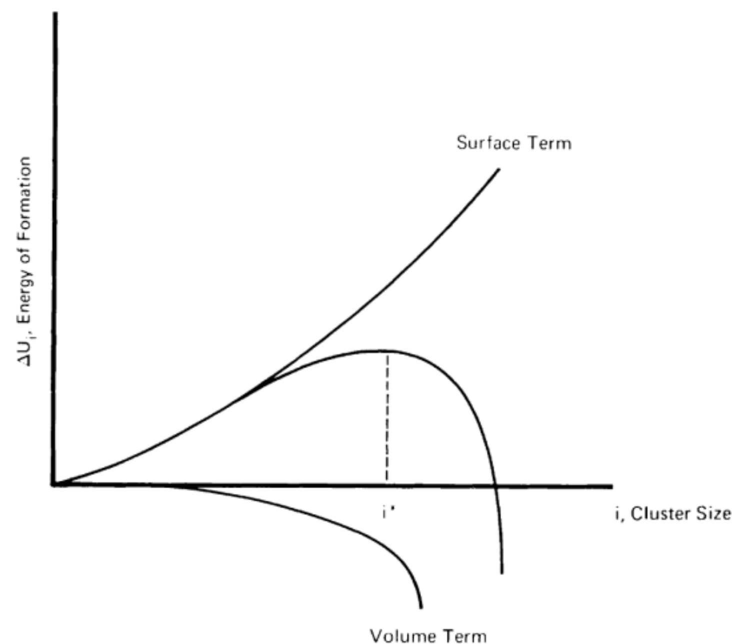



Figure 2.1.: Energy function $\Delta U$ in the formation of a cluster of size i (image from [5])

A gas can be described as undersaturated, saturated or supersatured if the pressure of the gas is less than, equal to or higher than, respectively, the vapor pressure of the liquid phase at the same temperature. Since the undersaturated and saturated state are thermo-

dynamically stable, nucleation can only be observed in supersatured states [5]. In this case the phase transition towards the liquid phase is driven by a thermodynamical equilibrium. Although the creation of a new volume in the liquid phase provides free energy, due to the pressure difference between the supersaturated vapor and the liquid droplet, the resulting interface between the two phases costs energy [22]. Therefore, small droplets which provide only a small volume, but a relatively large interface, tend to collapse. Fig. 2.1 shows the basic scheme of the energy contribution of the so called volume and surface term. The number of particles where the energy $\Delta U$ reaches a maximum is called the critical size of a cluster. Clusters with particle numbers above the critical size have a high probability to remain stable and to grow further, as no more energy needs to be spent in expanding the droplet. For this reason mainly clusters with a particle number equal to or higher than the critical size are of interest for nucleation analysis.

The nucleation process is often characterized by its nucleation rate $J$ which describes the rate of clusters above the critical size which appear per unit volume and time. The general formula is [18]:

$$J = \frac{n}{t \cdot V} \tag{2.1}$$

where $n$ denotes the number of clusters above the critical size and $t$ and $V$ are the time and volume.

CNT provides an analytic estimate of $J$ under several assumptions, such as the restriction that clusters only change their size by capturing or loosing single particles, and considering only spherical clusters. For further information see [17, 29]. However, CNT is not reliable for every nucleation process. As Fladerer [11] indicated, especially in the case of Argon, the CNT estimate fails to predict the nucleation rate. Similar results were obtained by Yasuoka et al. [29]. For this reason, numerical simulation is important to the study of nucleation processes.

## 2.2. Molecular dynamics

In molecular dynamics simulation [14] the behavior of particles is simulated. The particles are represented by their type, velocity, position, mass, geometry and the forces which act on them. In order to be able to compute their movement the particle interactions have to be considered. Newton's second law:

$$F = m \cdot a \tag{2.2}$$

can be used to formulate a system of ordinary differential equations which is solved in a time-step method. One famous candidate is the Velocity-Störmer-Verlet method [26] which first calculates the new position $x_i$ from the values of the forces and velocities of the last time-step 2.3. In a second step the new velocities are calculated 2.4. The resulting algorithm looks as follows [14]:

$$x_i^{n+1} = x_i^n + \delta t v_i^n + \frac{F_i^n \cdot \delta t^2}{2m_i} \tag{2.3}$$

$$v_i^{n+1} = v_i^n + \frac{(F_i^n + F_i^{n+1})\delta t}{2m_i} \tag{2.4}$$

How the force $F_i^n$ is obtained is discussed in the next section.

### 2.2.1. Force calculation

One of the crucial points of the molecular dynamics simulation is the force calculation. The general idea is to calculate the force from a potential $U$ which can be calculated from the multibody potentials $U_n$ by [18]:

$$U = \sum_i U_1(r_i) + \sum_i \sum_{j>i} U_2(r_i, r_j) + \sum_i \sum_{j>i} \sum_{k>j>i} U_3(r_i, r_j, r_k) + ... \tag{2.5}$$

The force is then calculated by [18]:

$$F = -\nabla U \tag{2.6}$$

where $\nabla$ denotes the spatial gradient.

For the simulation with gas molecules, potentials with more than two bodies can be neglected. In this work the Lennard-Jones potential is used [14]:

$$U_{LJ}(r_{ij}) = \frac{1}{n-m} \left(\frac{n^n}{m^m}\right)^{\left(\frac{1}{n-m}\right)} \epsilon \left[\left(\frac{\sigma}{r_{ij}}\right)^n - \left(\frac{\sigma}{r_{ij}}\right)^m\right], m < n \tag{2.7}$$

Here $r_{ij}$ is the distance between the two particles $i$ and $j$ . To simulate the van der Waals force, i.e. the attraction between the molecules, a common choice is $m = 6$. For the ease of computation often $n = 12$ is assigned, which has no real physical justification [14] but has shown good results in simulating the repulsive forces between particles[18]. The resulting equation is:

$$U_{LJ}(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^6\right) \tag{2.8}$$

The remaining variables $\epsilon$ and $\sigma$ are used to characterize the properties of the used chemical species. If the Lennard-Jones potential between different materials needs to be computed, the $\epsilon$ and $\sigma$ values of the mixture are obtained using the Lorentz-Berthelot mixing rule [6]:

$$\sigma_{12} = \sigma_{21} = \frac{\sigma_1 + \sigma_2}{2} \tag{2.9}$$

$$\epsilon_{12} = \epsilon_{21} = \sqrt{\epsilon_1 \epsilon_2} \tag{2.10}$$

However, in homogeneous nucleation only one single chemical species is simulated and therefore the mixing rule does not need to be applied.

### 2.2.2. Linked cell algorithm

The force calculation which was presented in the last chapter has to be carried out for every particle pair in the domain. With a domain consisting of $N$ particles, there exist $\binom{N}{2}$ such pairs resulting in an overall complexity of the order of $O(N^2)$, which is unacceptable for a simulation. One method to reduce the order of complexity is to truncate the Lennard-Jones potential. The so called cutoff radius $r_c$ only considers the particles which contribute significantly to the force calculation. Since the high value of the exponent $n = 12$ in equation 2.8 produces very small forces for particles with a large distance $r_{ij}$, this approximation makes sense. As only a limited number of particles can reside in one cell, due to the finite volume of one molecule, only a constant number of particles within this cutoff radius have to be checked for the force calculation. This reduces the complexity of the force calculation to $O(N)$.

The problem now is the fast determination of the particles within this range without checking every particle in the domain, which would again result in a complexity of $O(N^2)$. One solution was introduced by Verlet [27], where for every molecule a list of neighbors is saved which are at a distance of at most $r_{max}$. Thus, only particles from the list have to be considered in the force calculation. After a fixed number of time-steps, the lists are updated to allow a dynamic setup. Hence, $r_{max}$ and the number of time-steps until the next update have to be chosen carefully. Otherwise particles at a distance below $r_c$ might appear during the simulation which are not contained in the neighboring list.

Although this method guarantees the linear complexity, the assignment of the particles to the neighboring list requires a lot of book-keeping. A simpler approach is to introduce a grid to the domain by dividing it into separate cells. This procedure directly leads to the linked cell algorithm[14].

The linked cell method uses the cells generated by the grid to limit the space which needs to be searched for neighboring particles within the cutoff radius. A common choice is to set the cell length equal to $r_c$. Consequently, only the particles within the same cell or an immediately adjacent cell have to be considered to calculate the Lennard-Jones potential on a single particle. As a result the overall complexity remains $O(N)$. Since particles move in space and therefore might leave one cell and enter another one, the assignment of particles to the cells has to be done in every time-step. A 2D example of the linked cell structure can be seen in Fig. 2.2.

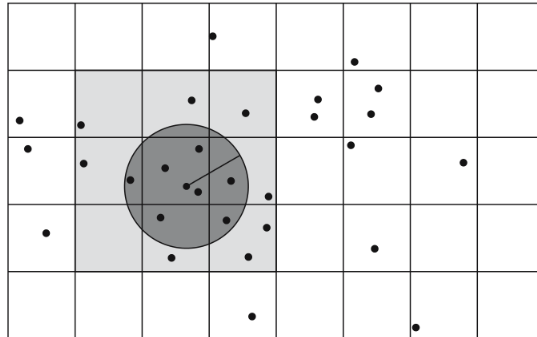


Figure 2.2.: The linked cell algorithm. Only the neighboring cells (gray) have to be traversed for the force calculation. (image from [14])

### 2.2.3. Periodic boundary condition

Another important factor of the numerical simulation is the treatment of the domain boundaries. As computational resources are limited, one can only simulate a limited number of cells. Therefore, boundary conditions have to define the treatment of the borders of the domain. In the simulation of gases, usually the periodic boundary condition is applied, which mimics an infinite volume. If a molecule leaves the domain on one side, it is moved to the boundary of the opposite side, e.g. from left to right. Fig. 2.3 shows this conceptional idea and the resulting new interpretation of the cutoff radius in the force calculation.

To implement this behavior in the numerical simulation, an additional layer of cells, referred to as the "halo layer", is attached to the border of the domain. In this layer the particles of the opposite border are copied to allow easy access during the force calculation, as they can now be within the cutoff radius (see Fig. 2.3). An example of a domain with halo cells can be seen in Fig. 2.4.

The halo cells have to be updated in every time-step, since particles may leave or enter the "border cells", i.e. the cells of the domain which are directly attached to the border, at any time during the computation.

### 2.2.4. Parallelization

The aim of molecular dynamics is to simulate as big scenarios as possible in order to predict the results of experiments or to study molecular phenomenona, as well as reducing statistical noise during the simulation. Unfortunately, the force calculation is a computationally intensive task and restricts the number of particles which can be simulated. For this reason, massive parallelization is applied to the numerical simulation to obtain bigger scenarios. Eckhardt [10] recently succeeded in simulating $4 \cdot 10^{12}$ particles by using 146016 cores of the SuperMUC [4] to simulate several steps of a molecular dynamics simu-

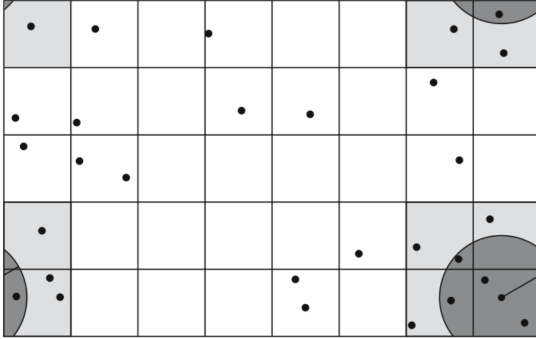


Figure 2.3.: Visualization of the cutoff radius with periodic boundaries (image from [14])

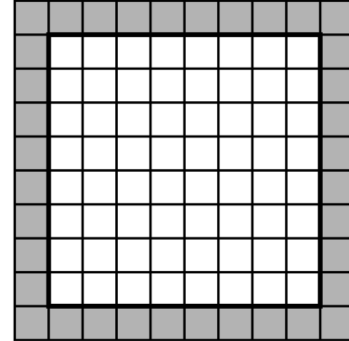


Figure 2.4.: Visualization of the domain with halo cells (gray) (image from [14])

lation. Such numbers of particles can only be reached by applying suitable parallelization techniques to the numerical methods.

The basic algorithm for parallelizing the simulation is the decomposition of the computational domain [14]. Since the Lennard-Jones potential is short ranged, the force calculation can be computed locally by only considering the particles within the cutoff radius. Consequently, the domain can be split up and assigned to different processors. A basic example can be seen in fig. 2.5.

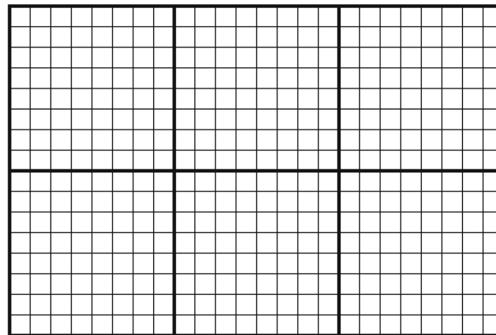


Figure 2.5.: Decomposition of the domain in case of six processors (image from [14])

This strict separation leads to the problem that particles within the cutoff radius at the borders of the single domains are located at different processors. To circumvent this problem every subdomain is expanded by a layer of halo cells, similar to the ones with periodic boundaries. In these halo cells the particles of the corresponding neighbor are copied. It should be noted that the periodic boundary concept needs still to be applied correctly.

Since particles can enter and leave halo cells at any time during the simulation, the halo cells and the local domains need to be updated every time-step which requires a suitable particle exchange. As the particles are now distributed over the different processors this particle exchange needs to be coordinated. The naive implementation for updating the halo cells would send the border particles, i.e. the particles within border cells, to all the 26 surrounding cells. A more efficient algorithm [14] uses only three communication steps for each cell. The main idea is to separate communication into three phases, one for each dimension, and to use the information which was received from other processors and send it together with the own border particles. As a result only three communication neighbors are needed. The concept can be seen in Fig. 2.6. For the exchange of particles which enter another subdomain, a similar algorithm is used. For further information we refer to [14].

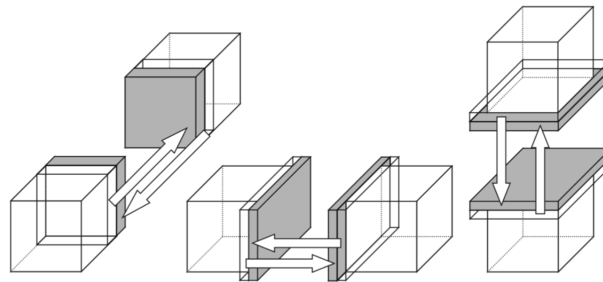


Figure 2.6.: Exchange of the particle information for the halo region. The relevant information gained in the first exchange can be send together with the border particles to the next processor in the second exchange. Analogously the information from the first and second exchange is reused in the third one. (image from [14])

## 2.3. Clustering criteria

With the methods of molecular dynamics a particle simulation can be computed in supersaturated scenarios where nucleation is expected. However, a clustering criterion has to be defined to identify clusters, i.e. a set of bounded particles.

Various techniques of cluster definitions have been studied in the work of Kible [18]. He found out that the classical data-mining clustering algorithms, such as k-Means [15], are not suited for the nucleation process. Therefore, Kible proposed the usage of the thermodynamic clustering criteria which where mainly influenced by the work of Hill [16]. Hill postulated a clustering criterion which assumes a cluster to be a set of pairwise bounded particles. This approach transfers the cluster detection into a simple graph problem: the detection of connected components. The nodes of the graph are defined as the particles

and the edges as bonds between them. For the detection of bonds, a clustering criterion is used which defines the properties of a bounded particle pair.

### 2.3.1. Energy criterion

The energy criterion was first introduced by Hill [16]. He studied the energetic properties of particles in clusters. He concluded that two particles can be considered to be bound in a cluster if their relative kinetic energy $e_{kin} = \frac{1}{2}m \cdot v_{rel}^2$ , using the relative velocity $v_{rel} = |v_i - v_j|$ [18] of the two particles i and j, is smaller than their negative potential energy $e_{pot}$, e.g. the Lennard-Jones potential. Consequently the following equation holds for a bounded particle pair [18]:

$$e_{kin} + e_{pot} <= 0 \tag{2.11}$$

If the Lennard-Jones potential is used, equation 2.11 can be written as [18]:

$$4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) + \frac{m}{2}(v_i - v_j)^2 <= 0 \tag{2.12}$$

This clear definition of a bounded pair enables a simple way of implementing the energy criterion in a numerical simulation. In addition, the Lennard-Jones potential is already computed in the force calculation. Hence, only the relative kinetic energy produces an overhead in the whole simulation.

Despite this easy integration within the force calculation, the energetic criterion has some weaknesses. The energetic analysis of a pairwise bond between particles is not always sufficient to reliably identify clusters. As pointed out in [20] a cluster of multiple particles (e.g. 3) cannot be detected if the pairwise potential energies do not fulfill equation 2.11. Thus, there might be a certain amount of clusters which appear unrecognized in the simulation. For this reason the geometric criterion is often implemented, which is discussed in the following.

### 2.3.2. Geometric criterion

In 1963 Stillinger [23] came up with the idea to analyze the formation of droplets during nucleation from a geometric point of view. Since liquid drops inside the vapor phase have a higher density than the gas molecules, the distances between the particles in the liquid phase should have smaller values than others. This idea leads to a very intuitive and easy-to-implement clustering criterion which is called the geometric criterion or "Stillinger" geometric criterion. Similar to the cutoff radius, which is used in the force calculation, a connectivity distance $r_g$ is defined which describes the maximum distance of two bounded particles. A connected component within the graph of bounded particles then forms a cluster. One key element of the geometric criterion is the determination of a suitable connectivity distance $r_g$. As Stillinger stated in [23], a too high value results in many false

positives, i.e. particles are assigned together in a cluster which do not form a cluster, and a too small value for $r_g$ does not detect all the existing clusters. A common choice for this connectivity distance is [17, 18]:

$$r_g = 1.5 \cdot \sigma \tag{2.13}$$

where $\sigma$ denotes the constant of the simulated chemical species which is also used in the Lennard-Jones potential.

One great advantage of the geometric criterion is the simple distance calculation which can be easily implemented within the simulation routine. Since in the normal molecular dynamic simulation the distance between every particle is already determined during the force calculation, the detection of bounded particles creates nearly no computational overhead.

Apart from the choice of the right $r_g$ the geometric criterion faces another problem. As the distance between two particles does not necessarily imply a bound between them, false positives are detected during the computation. For example if particles with a very high relative velocity are within the connectivity distance, they are falsely considered to form a cluster [20]. Thus, the results of the simulation become somewhat inaccurate.

One possibility to fix this problem was proposed by Pugnaloni and Vericat [20] who added a residence time $\tau$ to the method of the geometric criterion. They differentiate between physical clusters and chemical clusters. A physical cluster does not necessarily need fixed bonds between particles, i.e. particles can move within the cluster independently, and therefore the new criterion only considers the set of particles which form the cluster. A physical cluster at time $t$ exists if this set has been connected, over a path of particles with a distance less than $r_g$, during the whole time interval $[t - \tau, t]$. On the other hand, for a chemical cluster every single bonds needs to last for at least the time $\tau$ in order to belong to a cluster.

Another way to improve the accuracy of the geometric criterion is to consider particles as part of one cluster if the graph fulfills the biconnectivity criterion [9]. Consequently, particles which are temporally located near the cluster and which are only bounded over a single path are not assigned to the cluster. This method is supported by the fact that some chemical species form droplets of spherical shape, such as Argon [18], and therefore no particles in big droplets should only be connected over a single path to the cluster. This also holds if two big clusters are temporally connected via one particle, as this connection should not be considered to connect both clusters.

### 2.3.3. Hybrid criterion

The disadvantages of the geometric criterion were the reason for further research in the area of clustering criteria. One promising criterion is called the hybrid criterion [18, 17]. As the name suggests this criterion combines two different properties to characterize the bond between two particles, the geometric criterion and a new energy criterion. The new energy criterion differs from the one from Hill [16], since it does not look at single particle pairs

anymore but instead considers potentials from all of the surrounding particles. Hence, the potential energy from one particle to the surrounding particles is compared to the kinetic energy of the particle. A particle is considered to be part of the liquid phase if the following equation holds [18]:

$$\sum_j \frac{1}{2} u_{ij} + u_{kin} < 0 \tag{2.14}$$

This means if the attraction, due to the potential energy, exceeds the kinetic energy, the particle cannot move away from the other particles and is therefore bounded. The factor $\frac{1}{2}$ is used to simulate the effect that only one half of the potential energy is received by every particle for every interaction. It should be stated that $u_{kin}$, in this case, represents the kinetic energy using the absolute velocity and that only the part of the kinetic energy which is caused by the Brownian motion must be considered. For further details read [18].

The main method looks then as follows. The geometric criterion is evaluated during the force calculation of the simulation. The new energy criterion is then applied on the resulting graph. All the particles which are assigned to the liquid phase, according to equation 2.14, form a cluster in the graph which was obtained by the geometric criterion. In other words the new energy criterion acts like an additional filter sorting out particles from the graph which are not bound from an energetic point of view.

The advantage of the hybrid criterion is an improved accuracy in comparison to the geometric criterion. Furthermore it provides an energetic justification for the clusters. Although little artifacts, such as droplets consisting of single particles, were noticed by Kible, the new criterion showed better results than the energy and the geometric criterion, especially in the most relevant case of larger clusters.

## 2.4. Cluster identification

### 2.4.1. Sequential cluster identification

Based on the clustering criterion bonds between single particles can be detected. A clustering algorithm has to define how these bonds are processed and how they form a cluster. A standard technique is to define the cluster as a connected component of bounded particles. Several methods have been developed to identify connected components in nucleation scenarios. One approach was developed by Stoddard [25] and uses a list-like structure saved in an array to determine which particles belong to the same cluster. A pseudo code of the algorithm looks as follows [18]:

```
1 for (i = 1; i<=n; i++) L[i]=i //initialize
2 for (i = 1; i<n; i++){
3   if (L[i] != i) continue;    //i is already in a cluster
4   j=i;
5   do {
```

```
6         for (k=i+1; k<=n; k++){
7             if(L[k]==k  //not assigned to a cluster yet
8             && molecules j and k share a bond){
9                 swap (L[j] and L[k]);
10                }
11        }
12        j = L[j];
13    }while (j!=i)
14 }
```

The algorithm iterates over every particles and searches all particles to which they are bound. If a bond exists between particle j and k and k is not already assigned to a cluster, their ids are swapped. Therefore a linked list is constructed for every cluster.

Another way of identifying the clusters is to use the graph structure explained in section 2.3. In this graph a connected component can be determined by a depth-first search or a breadth-first search [9].

### 2.4.2. Parallel cluster identification

A problem of the parallel cluster identification are the bonds between domains of different processors. For local clusters (clusters that have no bond to particles outside the local domain) and the local parts of parallel clusters (clusters that are connected to other domains) one of the sequential cluster identification methods can be applied. To merge the local parts of the parallel clusters a communication step needs to be added to the cluster detection. For this purpose two main concepts have been developed over the last years: an iterative [18] and a hierarchical approach [18, 21].

The iterative approach communicates the cluster ids of the local parallel clusters to the neighboring processors. With this information the clusters ids can be updated by setting the same cluster ids on both processors. One deterministic method for obtaining the same cluster ids is to use the smaller cluster id for both processors. Since clusters can cross several processors and can have very complicated shapes, multiple communications steps may be required until every part of the cluster has the same id. The method stops if no cluster id changes during one communication step. This possibly very large number of communication steps can be a bottleneck of the iterative method. An example for a cluster which needs many communication steps to update every cluster id is shown in Fig. 2.7. In this case it would need eight steps to propagate the cluster id 1 through the whole cluster since in every step all local cluster ids are only propagated to directly connected cluster parts.

On the other hand the hierarchical approach uses subsets of the global domain and performs the cluster identification within these subdomains. A master processor then merges the parallel clusters in each subdomain. Therefore, all processors of one subdomain have to send all parallel cluster information to the master processors. However, clusters with

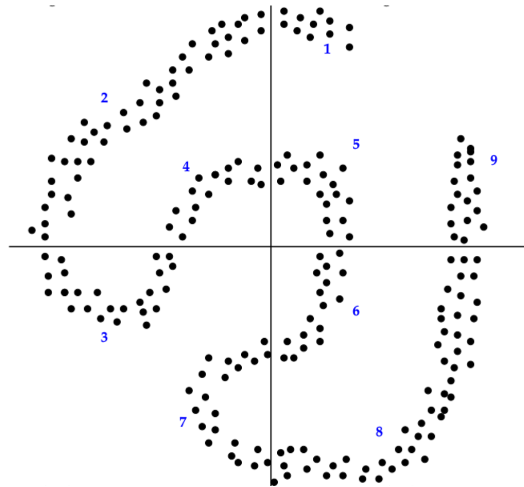



Figure 2.7.: Problematic cluster for the iterative method. Eight communication steps are needed to update the cluster ids correctly (image from [18])

bonds to other subdomains cannot be merged. For this reason this procedure is applied recursively on subdomains of increasing size till one processor gains all the necessary clustering information of the whole domain. A common choice is to double the length of the subdomains in every step, which leads to a logarithmic number of communication steps. An example of the hierarchical method with two communication steps can be seen in Fig. 2.8.

A special case of the hierarchical method uses only one communication step. Hence, all parallel clustering information is sent from every processor to one master processor which performs the merging of the parallel clusters. This method reduces the communication steps to one, resulting in a highly efficient cluster detection, but reduces the scalability of the method. This one-layered hierarchical approach will be implemented in this work and possible optimizations to provide better scalability are introduced.

One benefit of the hierarchical methods is that the problematic case of the iterative method (see Fig. 2.7) can be solved with no additional communication steps since cluster information can be updated globally.

There exist other methods based on a parallel connected component detection as described in [12], but they are too general for the estimation of the nucleation rate, as only the size of the clusters is important. In addition, further simplifications, such as the usage of unique particle pairs between parallel clusters (see 3.2.3), can be made for the merging of parallel clusters which are not generally applicable to a connected component search. Thus, the general methods are too expensive for calculating the nucleation rate.

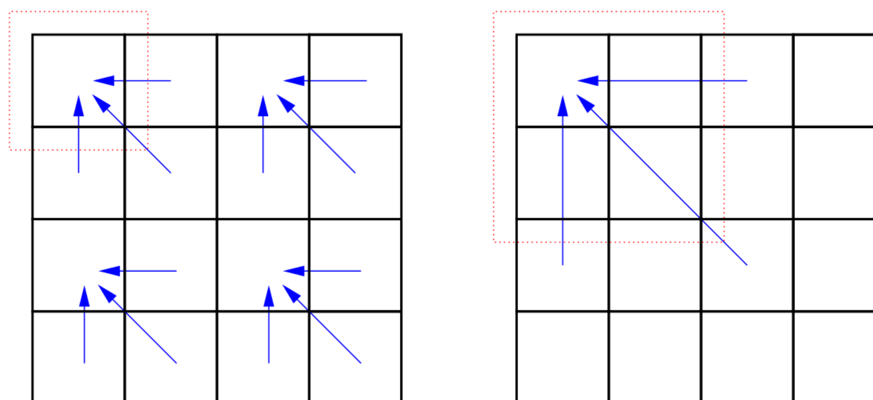


Figure 2.8.: Hierarchical method with two communication steps. In the first steps four processors form a subdomain. In the second step only one master processor is left which gains all the information of the whole domain from the previous subdomain master porcessors. (image from [18])

## 2.5. MarDyn

The molecular dynamics simulator used in this work is called MarDyn [8]. MarDyn was written in C++ and uses a modular structure in order to implement different algorithms for various applications scenarios within molecular simulations. For parallelization the Message Passing Interface MPI [13] was used along with the domain decomposition described in section 2.2.4. For further information read [8].

In this thesis the existing features of MarDyn are extended by a clustering detection. The concrete implementation will be described in chapter 3.

## 2.6. Relation to connected component labeling

The cluster detection problem in the numerical simulation is closely related to the connected component labelling (CCL) problem in computer vision [24]. In CCL unique labels are assigned to connected components within a regular grid. Knop and Rego [19] described a parallel method to solve the problem of CCL by applying a data reduction method. They use the hierarchical method described in section 2.4.2 and reduce the communication data with mapping tables. By merging these mapping tables, globally unique cluster labels are assigned to the parallel clusters. This reduction of data is similar to the coarse graining used for the second depth-first search of our parallel clustering algorithm (see chapter 3.2).

However, there are some important differences between the CCL problem and the clus-

ter detection in our simulation. At first, we only need statistical data of the clusters, such as the cluster size, and not the specific bonds of particles within the cluster. Therefore in our approach local clusters can be simply collapsed into single nodes which reduces the amount of data. Second, no globally unique cluster ids are needed in our implementation, which further reduces the complexity of the algorithm in comparison to the CCL problem. As a result, a dedicated master processor is able to compute the global cluster detection in our method since no cluster information from the global depth-first search has to be sent back to the local domains. Finally, the usage of a regular grid can not be directly transferred to the molecular dynamic simulation due to the fact that molecule positions have real values, while the pixel position of the regular grid have integer values. Even if the linked cell structure is used to mimic a regular grid, an important difference would be that multiple clusters with various parallel edges (edges that connect parallel cluster parts of different subdomains) might exist within one cell, in contrast to the CCL problem where every cell either contains a unique local cluster or nothing.

In conclusion, the CCL problem shows some similarities to our algorithm, but it is not directly applicable on nucleation scenarios. Therefore, to the best of our knowledge, the presented coarsening strategy is novel.

# 3. Implementation

In this work the geometric criterion was chosen as cluster criterion and a depth-first search is used for the cluster identification. The cluster detection is divided up in several steps. The first step is the detection of the single bonds between particles. Therefore the clustering criterion has to be integrated in the existing simulation environment. The next section will describe in detail the implementation of the geometric criterion in the sequential case.

## 3.1. Sequential cluster detection

### 3.1.1. Implementation of the geometric criterion

The geometric criterion 2.3.2 considers two particles to be bound if their distance is smaller than the connectivity distance. The calculation of the distance in 3D is quite expensive due to the square root.

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \tag{3.1}$$

One approach to reduce this complexity is to evaluate only the squared value of the distance, so that no square root needs to be applied. Consequently, the squared connectivity distance has to be compared to the squared distance:

$$r_{ij}^2 < 1.5^2 \cdot \sigma^2 \tag{3.2}$$

The implementation can further be optimized by integrating the geometric criterion within the distance comparison of the cutoff radius in the force calculation. As the squared distance $r_{ij}^2$ is already computed for the Lennard-Jones potential, this strategy provides very good performance. A schematic example, illustrating the force calculation of one molecule combined with the evaluation of the geometric criterion, can be seen next.

```
1 Molecule& m1 = currentCellNextMolecule();
2 for (//iterate over all Molecules within current cell
3      //and neighboring cells) {
4    Molecule& m2 = nextMolecule();
5    double squaredDistance = m2.distSquared(m1);
6    //check cutoff radius and calculate force
7    if (squaredDistance < cutoffRadiusSquared) {
8       calculateForce(m1,m2);
```

```
9       }
10      //check geometric criterion and add edges to graph
11      if (squaredDistance < connectivityDistanceSquared){
12          edges[m1.id()].push_back(m2.id());
13          edges[m2.id()].push_back(m1.id());
14      }
15  }
```

Since the adjacency matrix of the resulting graph will be a sparse matrix, i.e. only few fields will be set to 1, the edges are saved in an array of vectors called $edges$. The vector at position $edges[i]$ saves the molecule ids of all molecules connected to molecule $i$. It should be noted that the molecule ids in MarDyn are globally unique which guarantees that the edges can even be set correctly in a parallel cluster detection.

### 3.1.2. Depth-first cluster detection

The previous section described how the graph for the cluster detection can be constructed. The clusters can be seen as connected components within this graph. Therefore an algorithm which determines connected components has to be implemented. One easy approach is to use a depth-first search [9] which is shown next.

```
1   void ComponentCalculator::depthFirst(){
2       int id = 0;
3       int clusterSize;
4       //mark every molecule as unvisited
5       for(int i=0; i<numMolecules; i++){
6           visited[i] = -1;
7       }
8       //iterate over all molecules
9       for(int i=0; i<numMolecules; i++){
10          //start search if not already visited
11          if(visited[i] == -1){
12              clusterSize = search(i,id);
13              size.push_back(clusterSize);
14              id++;
15          }
16      }
17  }
18
19  int ComponentCalculator::search(int vertex, int id){
20      if(visited[vertex] != -1){
21          //already visited
22          return 0;
```

```
23     }
24     //mark as visited
25     visited[vertex] = id;
26     int clusterSize = 1;
27
28     //iterate over all nodes which are connected to current node
29     std::vector<int>::iterator it(edges[vertex].begin());
30     while(it != edges[vertex].end()){
31        //sum up number of particles
32        clusterSize +=search(*it,id);
33        it++;
34     }
35     return clusterSize;
36 }
```

In the method $depthFirst$ a depth-first search is iteratively started from every molecule id that has not been visited by a previous search. The array $visited$ is used to indicate if a molecule has already appeared in the searching process. The field $visited[i]$ is defined to be $-1$ if the molecule with id $i$ is unvisited. Otherwise it contains the cluster id of the cluster the molecule belongs to. The $search$ method first checks if the current vertex was already visited. This is important to avoid loops during the search. If the node is unvisited, a depth-first search is started at every node which is connected to the current vertex.

Furthermore the basic depth-first search was extended by a counter which determines the cluster size, i.e. the number of particles in a connected component. This feature is achieved with the recursively summing up of the $clusterSize$ variable. Every time a node is switched from unvisited to visited a value of 1 is assigned to the $clusterSize$ variable. Hence the resulting return value of the search method is the number of particles contained in the cluster. This number is then saved in a vector called $size$ containing the number of particles belonging to cluster $i$ at position $i$.

In this thesis every connected component is considered to be a cluster. As mentioned in section 2.3.2 it is possible to improve the method by checking for the biconnectivity criterion. Check [18] for further information on the implementation.

## 3.2. Parallel cluster detection

We developed an application-driven parallel version of our sequential method which will be introduced in the next sections. We point out that only statistics of clusters of different sizes are needed in our clustering algorithm, but not the specific connections between molecules. The method represents a coarse-grained version of the one-layered hierarchical cluster identification method described in 2.4.2 and combines a local depth-first search with a global depth-first search on a coarsened graph. This new strategy of coarsening the graph for the second depth-first search allows us to reduce the data which needs to be

transmitted to the master processor. In addition, only one communication step is needed resulting in an improved performance.

### 3.2.1. General concept

The basic concept of our cluster detection within the domain decomposition is shown in Fig. 3.1.

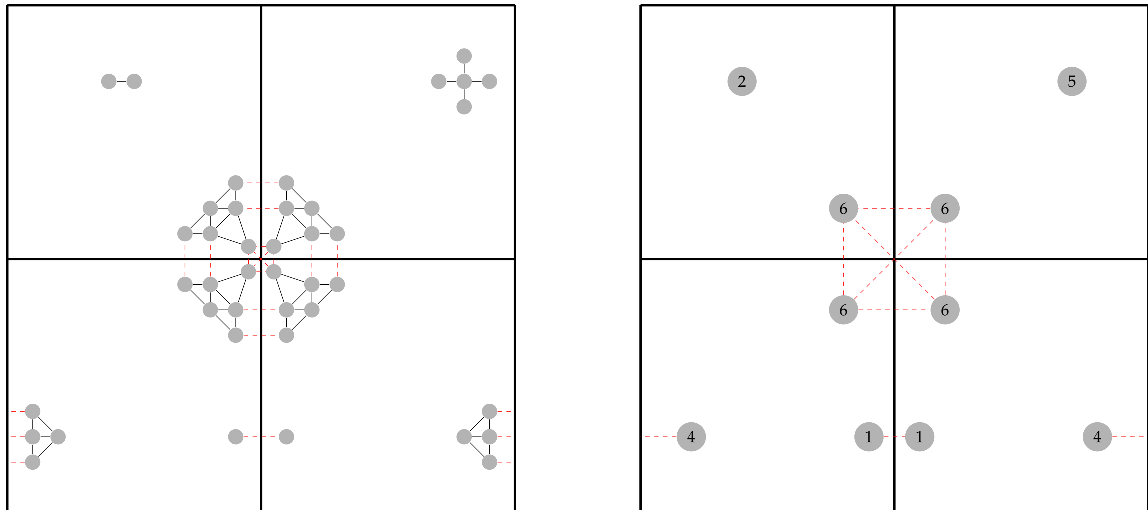



Figure 3.1.: Scheme of the cluster detection within the domain decomposition for four threads where edges that leave the local subdomains are drawn as red dashed lines. The local clusters are evaluated locally as in the sequential algorithm, but the local parts of parallel clusters are collapsed into single nodes and unique edges are set between them. The initial situation can be seen in the left figure and the coarsened graph with the collapsed clusters can be seen on the right figure.

The problem at hand is how to proceed with a cluster that is crossing the domain borders. Our algorithm shown in Fig. 3.1 treats such a parallel cluster like a part of a second more coarser graph. Since the local part of the cluster can be determined by a simple depth-first search as in the standard algorithm, it is possible to calculate the local number of particles. After this step the local cluster parts are collapsed into single nodes with an attached node weight equal to the local cluster size.

The problem which appears at this point is how to set deterministic edges between these cluster nodes. As the border layer of nodes is known to the neighbor processor, due to the halo layers, every processor can determine the parallel edges of a cluster. However, sending all parallel edges to the master would decrease the communication performance.

For this reason only one unique edge between two local clusters is exchanged to reduce the data for the communication step.

The key of the method is now how to determine a unique edge on the basis of these parallel edges, which can be potentially thousands. The idea of our algorithm is to choose the minimum molecule id of all the molecules from the border which have a connection to the halo and the minimum of all the molecule ids from the halo which are connected to the border area. This method has to be applied separately for every halo region from any surrounding processor. The result is a set of, up to 26, unique particle pairs representing the bonds to the surrounding processors which are connected to the cluster. We point out that the unique particle pair does not need to have a real bond since the minimum calculation is done separately for the boundary and the halo molecules. Hence, the particles contained in parallel edges are split into two groups for each of the 26 directions: the border particles and the halo particles. For every direction the minimal particle ids of both groups form the unique particle pair. The basic steps of this algorithm are shown in Fig. 3.2.

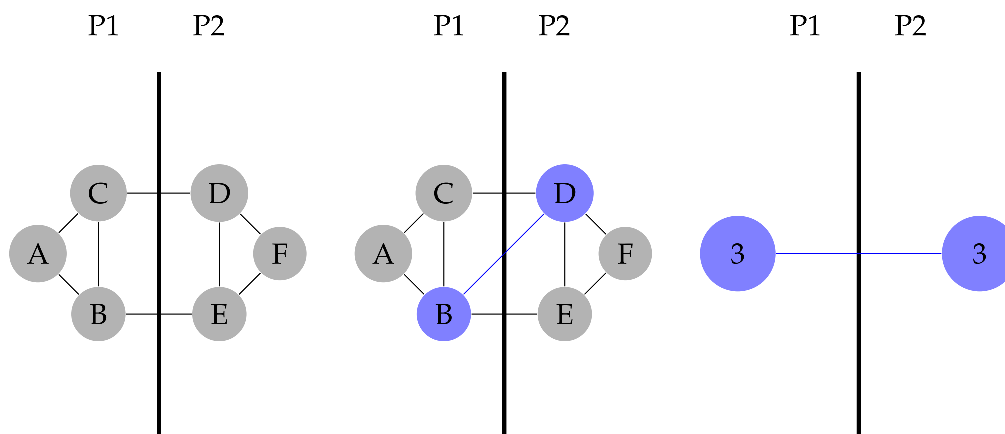



Figure 3.2.: Parallel cluster detection between domains of two different processors. Left: Initial situation. Middle: Unique particle pairs are identified. Right: Local clusters are collapsed into single nodes with node weight equal to local cluster size.

The result is a coarse graph representation of the global graph where the collapsed clusters are the nodes and the unique particle pairs represent the edges. Since the information is only known locally on every processor, a communication step is required. The relatively small size of the coarse graph, in relation to all bonds of one domain, leads to the idea of using a master processor which receives all the global cluster information. Therefore, for each parallel cluster part the local cluster size is sent together with the, up to 26, unique particle pairs to the master processor. Based on this information the edges between the parallel clusters are formed with the unique id pairs and a second depth-first search is performed. Finally, the cluster sizes of the parallel clusters are obtained by summing up the

weights of the parallel cluster parts during the global depth-first search.

There are two ways to define a master processor. One possibility is to use one of the processors which also performs the numerical simulation and the first depth-first search, causing a sequential section of the whole algorithm. The other method uses a dedicated processor which only performs the evaluation of the parallel clusters. An advantage of the second method is that the molecular dynamic simulation can proceed during the global cluster identification, provided that no cluster information is needed by the molecular dynamics simulation.

The next sections describe the changes to the sequential implementation by applying this concept.

### 3.2.2. Evaluation of the geometric criterion in halo regions

The comparison of the particle distance to the connectivity distance stays the same in the parallel evaluation of the geometric criterion. However, the location of edges which enter the halo or which are within the halo needs to be saved for the determination of the parallel edges in the depth-first search. Since the parallel algorithm differentiates between the halo regions coming from different neighbors, an important part of the new implementation is the fast determination of the specific halo region. In the new implementation every cell within the domain but not in the border area gets the location value 0. The border cells receive the value 1 and the different halo areas have the location values 2-27. To make the evaluation fast this assignment is already precomputed in the cell objects during the construction of the linked cell container. Consequently, the assignment of the location to a particle can be done very fast in the force calculation. The basic implementation for iterating over all molecule pairs of 2 cells looks like this:

```
1  //get locations
2  unsigned short locationCell1 = cell1.getLocation();
3  unsigned short locationCell2 = cell2.getLocation();
4  //check if one of the cells is in the halo
5  bool halo = (locationCell1 > 1 || locationCell2 > 1);
6  // loop over all particles in the cell
7  for (//iterate over all particles in cell1) {
8     Molecule& m1 = nextMolecule(cell1);
9     for (//iterate over all particles in cell2) {
10       Molecule& m2 = nextMolecule(cell2);
11       //calculate squared distance between molecules
12       double squaredDistance = molecule2.dist2(molecule1);
13       if (squaredDistance < cutoffRadiusSquared) {
14          calculateForce(m1,m2);
15       }
16       //checking geometric criterion and adding edges to graph
```

```
17        if (squaredDistance < connectivityDistanceSquared){
18            //adding bonds as edges to the graph
19            if(!halo){ //determine if it is a normal edge
20                _edges[molecule1.id()].push_back(molecule2.id());
21                _edges[molecule2.id()].push_back(molecule1.id());
22            }
23            else{ //edge into or within the halo region
24                //locations are saved in addition to the molecule ids
25                _haloEdges[m1.id()].push_back(std::make_pair(m2.id(),
26                    std::make_pair(locationCell1,locationCell2)));
27                _haloEdges[m2.id()].push_back(std::make_pair(m1.id(),
28                    std::make_pair(locationCell2,locationCell1)));
29            }
30
31        }
32    }
33
34 }
```

The difference in comparison to the sequential implementation is that the location is used to evaluate if bonds are within the halo area or enter it, also referred as a halo edge. If the bond is a halo edge, it is stored separately in the $haloEdges$ array together with the location values of both cells. The pair of location values is always saved in the same way at the vector position $haloEdges[i]$: the first location value refers to the location of molecule $i$ and the second one the location of the molecule which is connected to $i$. We point out that the molecule-pair traversal of MarDyn is not performed within halo cells. For the purpose of the cluster detection, however, we needed to add the traversal of the halo cells introducing a small overhead. The evaluation of edges within the halos is necessary to detect connections of parallel cluster parts over the halo layer, which reduces the number of parallel clusters parts and, therefore, the communication data. Furthermore, the number of cases in which the problem explained in section 4.5.1 appears, is reduced significantly. The effects of this overhead will be discussed in chapter 4.

### 3.2.3. Local depth-first search

The local depth-first search has to look for connected components in the new graph considering both sets, the normal edges and the halo edges. Furthermore the unique pairs of particle ids have to be found for the parallel communication exchange. This is done with the help of the location values of the $haloEdges$ array mentioned in the last section. A specific array called $parallelEdges$ saves the unique particle pairs for all, potentially, 26 directions per cluster. The concept is to save the unique particle pair crossing the domain border to the neighbor $i$ at position $parallelEdges[i]$. The first value of each pair is the min-

imum of the particle ids within the border belonging to a halo edge, whereas the second value refers to the minimum of the particle ids within the halo belonging to a halo edge. These minimal ids are updated throughout the depth-first search. Another challenge is the correct determination of the particle numbers in the local parts of clusters. Molecules which only exist in the halo cells must not be counted in the local cluster number. A problem is that some particles might appear in halo regions as well as in the border regions with small numbers of processors due to the periodic boundaries. In this special case they have to be counted, as the cluster is still on the same processor and therefore the depth-first search explores the whole cluster. Thus, it has to be assured that particles only contained in the halos are not counted. The resulting depth-first search can be found in the appendix A.2.

### 3.2.4. Parallel cluster communication

To compute the second depth-first search on the coarser graph, the parallel clusters have to be sent to the master together with the connecting particle pairs. This can be done with three MPI commands. First, the number of parallel clusters each process has detected, is sent to the master with one $MPI_Gather$. According to this information the master generates an offset array to correctly sort the incoming parallel clusters, i.e. the unique particle pairs and the cluster sizes of the local parts, into the receive buffer of the second communication. Here the slightly different command $MPI_Gatherv$ is needed which allows variable input sizes of the different processors. Additionally, the number of already detected local clusters of appropriate sizes has to be communicated to the master in an $MPI_Allreduce$ command.

An improvement of this method can be achieved if the first and third step are combined. Hence, in the first $MPI_Gather$ the number of local clusters of appropriate size is sent in addition to the number of parallel clusters. Then only two MPI communication steps are needed. The complete implementation of the exchange is illustrated in the appendix A.1.

### 3.2.5. Global depth-first search

One challange of the second depth-first search is the unusual representation of the edges. The unique particle pairs are therefore first transformed into a common edge format, as it was used in the first depth-first search. For this purpose the routine $createGraph$ was implemented to create a graph with the parallel clusters as nodes, identified by a special id, and edges between those nodes. One simple method is to assign ids in the order of the parallel clusters received in the $MPI_Gatherv$. The edges are then obtained by setting up a 2D matrix of size $N^2$, where $N$ denotes the number of particles in the simulation and by using a sequential scanning process to find the corresponding particle pairs. The idea is to simply mark the matrix position $m_{ij}$ and $m_{ji}$ with the cluster id of the current cluster if no marking exists for this field. If an existing value is detected at this position an edge is created between the current cluster id and the one found at the memory location of $m_{ij}$.

Finally a graph representation like in the sequential case is obtained. For a fast determination of the cluster sizes of the local cluster parts an additional array $parallelClusterSizes$ is used which contains the local cluster size of the parallel cluster $i$ at position $parallelClusterSizes[i]$.

The resulting depth-first search is just a slight variation of the sequential one explained in section 3.1.2 where only the search procedure changes. Here the cluster size of a node which gets marked is set the cluster size of the corresponding parallel cluster, which is stored in $parallelClusterSizes$, instead of setting it to 1. The resulting new $searchParallel$ method was implemented as follows:

```
1  unsigned long ConnectedComponentCalculator::searchParallel(
2          unsigned long vertex, unsigned long id){
3      if(visited[vertex] != -1){
4          //already visited
5          return 0;
6      }
7      visited[vertex] = id;
8      //set cluster size
9      unsigned long clusterSize = parallelClusterSizes[vertex];
10     std::vector<unsigned long>::iterator it(
11         parallelEdges[vertex].begin());
12     while(it != parallelEdges[vertex].end()){
13         clusterSize +=searchParallel(*it,id);
14         it++;
15     }
16     return number;
17 }
```

### 3.2.6. Optimization

#### Memory optimization

The $createGraph$ method introduces a compexity of $O(N^2)$ for memory allocation, because of the N x N matrix. This makes the algorithm unusable for larger scenarios. For 5000 particles the memory used for the matrix is already around 100MB. Therefore, a sparse matrix was implemented for the detection of matching particle pairs. Equal to the edges in the depth-first search, an array of vectors called $matchArray$ is used. For every particle pair $ij$, $i$ is inserted in the vector at array position $matchArray[j]$ and vice versa. To find the corresponding cluster id for every edge, the cluster id is always saved right after the molecule id of the particle pair in $matchArray$. A match for a pair $ij$ exists if the corresponding vector $i$ already contains an entry $j$. If a match is found, the corresponding cluster id is used to form the cluster edge and the pair is set invalid in the sparse matrix. This searching in the vector does not affect the runtime, as there might be at most 26 edges for every

molecule and therefore only up to 26 entries in the vector have to be compared to $j$. This holds because every molecule belongs to one unique cluster and the number of edges to neighboring processors is limited by 26 for every cluster. Therefore the runtime is just affected by a constant and the overall complexity remains $O(c)$, where c denotes the number of parallel clusters parts. In the same way the memory complexity is reduced to $O(N)$. In order to delete the vectors at $matchArray[i]$ efficiently for the next iteration, the first entry of every vector contains the number of unresolved particle pairs for the molecule $i$. If this number drops to 0, the vector is deleted to indicate that no unresolved pair is contained in the vector. Hence, there is no need to run over the whole array at the end of the method. The resulting $createGraph$ method can be found in the appendix A.3.

### Runtime optimization

Another point for optimization is the depth-first search. In the original version a depth-first search is started at all particle ids contained in the global domain, which could lead to a bad scaling. Thus, an additional vector called $boundedParticles$ was integrated in the calculation which saves the particles which form bonds. The insertion of the molecules into the $boundedParticles$ can be directly implemented in the force calculation as shown below.

```
1  //checking geometric criterion and adding edges to graph
2  if (dd < _edgeCutoffSquared){
3      //adding molecules to the vector of boundedParticles
4      //if they are not already contained
5      if(_edges[m1.id()].empty() && _haloEdges[m1.id()].empty()){
6          _boundedParticles.push_back(m1.id()-1);
7      }
8      if(_edges[m2.id()].empty() && _haloEdges[m2.id()].empty()){
9          _boundedParticles.push_back(m2.id()-1);
10     }
11     //adding bonds as edges to the graph
12     if(!boundaryToHalo){ //determine if it is a normal edge
13         _edges[m1.id()].push_back(m2.id());
14         _edges[m2.id()].push_back(m1.id());
15     }
16     else{ //edge from halo to boundary
17         _haloEdges[m1.id()].push_back(std::make_pair(
18             m2.id(), std::make_pair(locationCell1,locationCell2)));
19         _haloEdges[m2.id()].push_back(std::make_pair(
20             m1.id(), std::make_pair(locationCell2,locationCell1)));
21     }
22
```

```
23 }
```

Consequently, in the depth-first search only the particles which have edges and, therefore, reside in the local domain are considered. This optimization provides a better scalability. Especially in the time before nucleation appears runtime is significantly reduced. The resulting implementation of the local depth-first search is shown in the appendix A.2.

### 3.2.7. Validation

The validation of the parallel algorithm was one of the crucial parts of this thesis. However, the validation against the serial version was complicated due to the fact that over a long period of time-steps the numerical values of the parallel version slightly differ from the ones of the serial version. This behavior could be noticed after about 1000 time-steps and caused slightly changed positions of the molecules. The reason for this is the changed summation order in the force calculation, as well as some round-offs within the global property calculations, such as the temperature for applying the thermostat. For this reason a direct comparison of simulation results over long periods of time-steps, which was used for validation of the optimization steps, were not possible, as the clustering results differed after very long time periods. Although these differences in the simulation appeared, it could be noticed that the first few time-steps of the computation returned the same result with any number of processors.

This observation was used to create a different validation method. The basic idea is to compare the results of short numbers of time-steps at different points of the simulation. Therefore several restart files were created between 200000 time-steps and 300000 time-steps of the simulation, where nucleation already occurs. At these starting points 100 time-steps were performed and the results were validated against each other.

In order to get reliable validation results, various numbers of cores in the parallel program, such as 1,2,3,4,8,16,32 and 64, were compared to the sequential program. The results showed no deviations from the sequential ones and the parallel algorithm can, consequently, be expected to be correct.

# 4. Results

## 4.1. Simulation setup

The introduced method was applied to several test-cases where nucleation appears with different numbers of particles. The following parameters have to be defined for the simulation of argon (one Lennard-Jones center) and ethane (two Lennard-Jones centers) (taken from [18]):

| Parameter | Description |
|---|---|
| $\rho$ | pressure |
| $T$ | temperature |
| $\delta t$ | time-step |
| $r_c$ | Lennard-Jones cutoff radius |
| $L$ | distance between the Lennard-Jones centers |
| $\epsilon/k_B$ | energy parameter of the gas molecule |
| $m$ | mass of the molecule |
| $\sigma$ | length parameter of the molecule |

For the computation the following values were assigned to the simulation parameters [18] which simulate physical valid nucleation scenarios:

| Molecule | $\rho$[mol/l] | $T$[K] | $\delta t$[fs] | $r_c$[Å] | $L$[Å] | $\epsilon/k_B$[K] | $m$[u] | $\sigma$[Å] |
|---|---|---|---|---|---|---|---|---|
| argon | 0.97 | 80 | 10.8 | 15.3 | - | 119.80 | 39.9 | 3.405 |
| ethane | 3.0 | 220 | 4.0 | 14.0 | 2.345 | 135.57 | 15.0 | 3.500 |

The scalability tests were performed on the SandyBridge partition of the MAC cluster [3] featuring 28 nodes of dual socket Intel SandyBridge-EP Xeon E5-2670 processors with 16 cores. Furthermore the simulation program MarDyn [8] was used with the standard domain decomposition and therefore no load balancing is applied. For time measurements the different processors were synchronized with $MPI_Barrier$. Hence, the times of the processor with largest workload are obtained. In the following the main simulation results will be presented and an evaluation of the introduced parallel algorithm will be performed.

## 4.2. Visual verification

The result of the nucleation process can be verified visually due to the usage of the geometric criterion. For visualization the tool ParaView [2] was used which shows snapshots of the particle configuration within the domain for every time-step. In Fig. 4.1 the common steps of the computation can be seen.

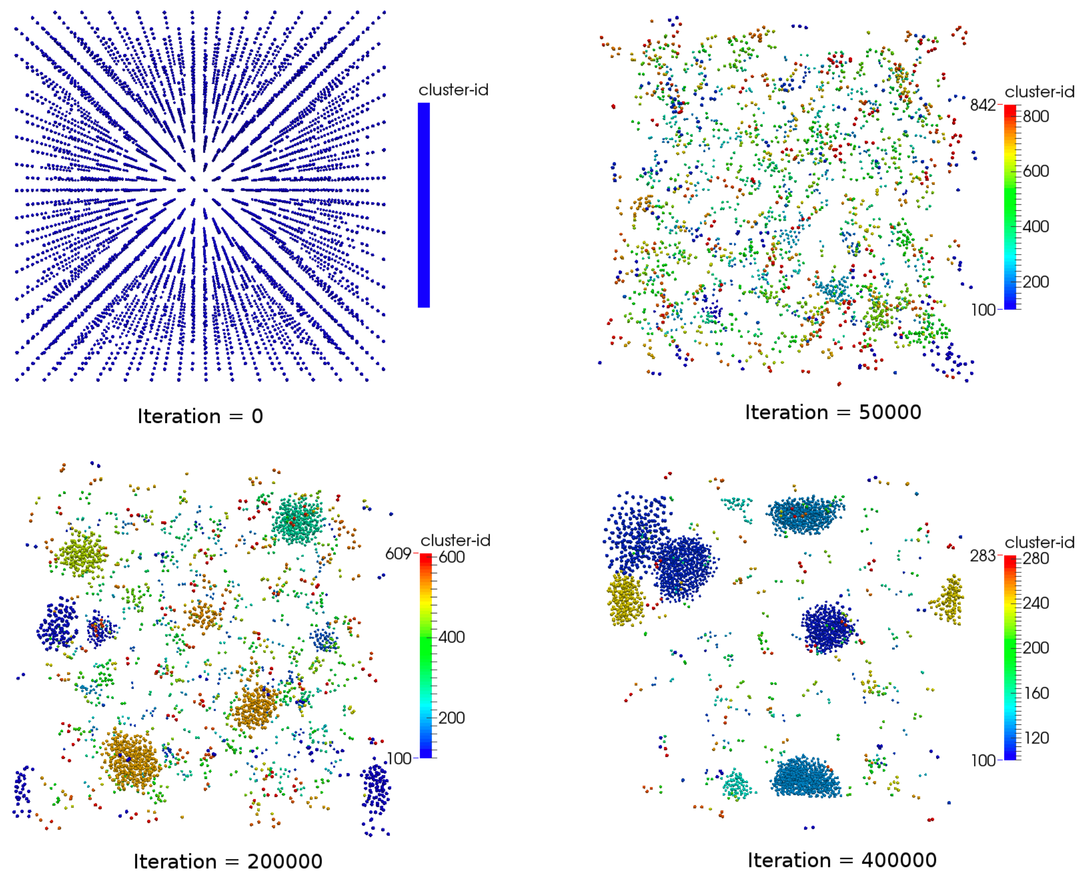



Figure 4.1.: Different clustering stages during the computation with 5000 particles. Clusters of size 1 (cluster id $< 100$) are only showed at iteration = 0. Top-left: Initial configuration on a nearly regular grid with no clusters. Top-right: Many, mostly unstable, clusters are formed. Bottom-left: Full nucleation reached with many clusters above the critical size of 20. Bottom-right: Final situation where most clusters have merged.

At first the molecules are arranged in a mostly regular grid, then few little clusters are

built which tend to collapse. After this step some clusters continue to grow but still most of them vanish. At about 200000 time-steps full nucleation is reached where many stable clusters have been built above the critical size. Finally clusters are merging and therefore the number of clusters is continuously decreasing. This effect of merging droplets is also called coagulation [18]. Fig. 4.2 shows an example of the coagulation process.

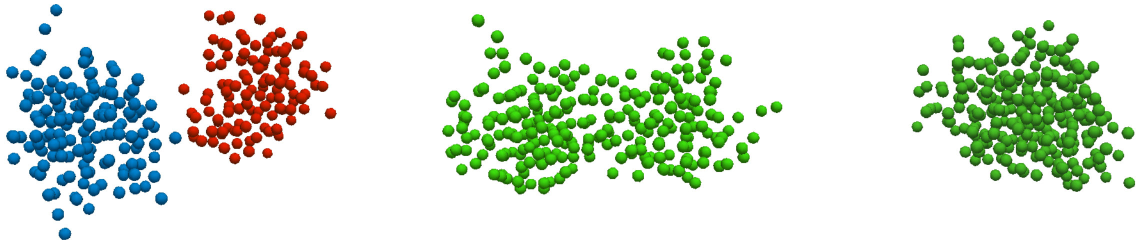


Figure 4.2.: Coagulation causes the merging of two clusters. Three merging steps (from left to right).

A similar observation can be made with ethane. Here the nucleation is much quicker than in the argon case and already after 50000 time-steps huge clusters are formed. Fig. 4.3 shows some of the stages of the nucleation with ethane. It can be clearly seen that ethane, in contrast to argon, builds non-spherical clusters.

Overall no errors of the implemented cluster detection were observed and the general nucleation process fits to the theoretical concept.

Another way of analyzing the results visually is to study the building of clusters over short time-periods. As this approach allows the examination of the rate of successfully detected clusters, it is a good measure for the effectiveness of a clustering criterion. Fig. 4.4 shows two clustering situations which are about one picosecond apart.

It can be clearly seen that some particles leave clusters and some clusters completely vanish. This observation is a result of the geometric criterion and not a bug of the implementation, since particles which are temporally close to each other are considered to share a bond. In addition small clusters tend to collapse which increases this effect. However, the overall rate of correctly identified clusters is high.

## 4.3. Physical verification

Besides the visual verification a physical one is important to determine the physical correctness of the whole simulation. One possibility is to compute the nucleation rate, as

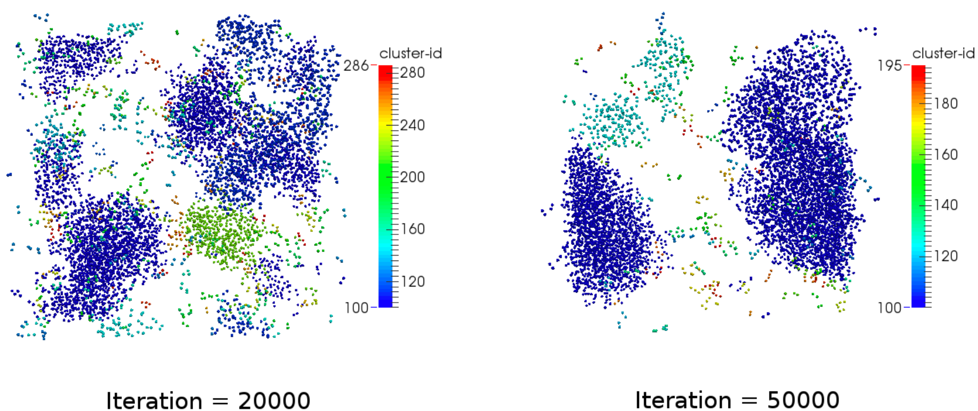



Figure 4.3.: Clusters of size > 1 in the nucleation steps of ethane. At first many small clusters are build (left) which quickly merge (right).

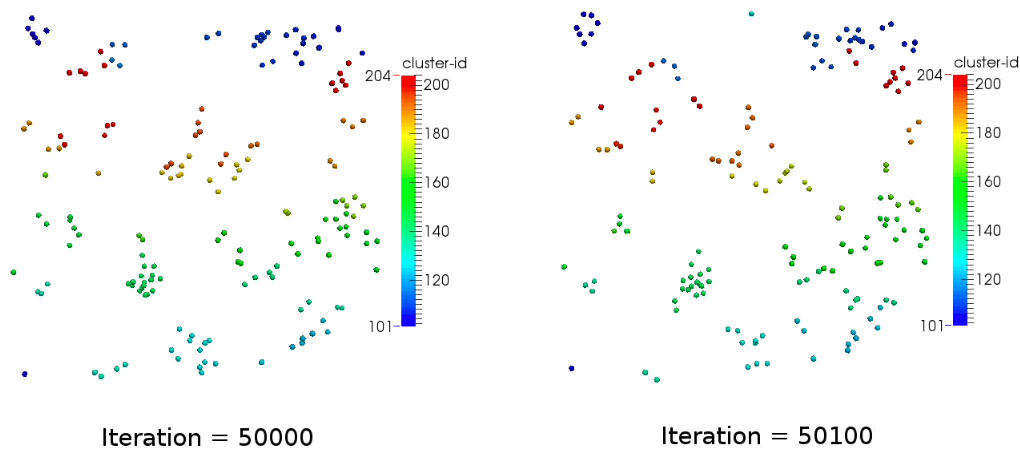



Figure 4.4.: Clusters of size > 1 at time-steps 50000 and 50100.

described in 2.1, and to compare it to other simulations or experiments.

Before the calculation of the nucleation rate the number of clusters above the critical size are compared to previous works. The nucleation rate can then be calculated by plotting the

number of clusters above the critical size and calculating a linear fit f during the nucleation phase [18].

Fig. 4.5 shows the numbers of clusters above size of 20, which is approximately the critical size for argon [18, 28], and above size 40. The results in general look very similar to the ones from Kible [18]. The only major deviation is the missing second increase in the cluster numbers in our results. However, as stated by Kible this was a rather unusual and unexpected behavior. Therefore the curve fits even better to the theoretical model.

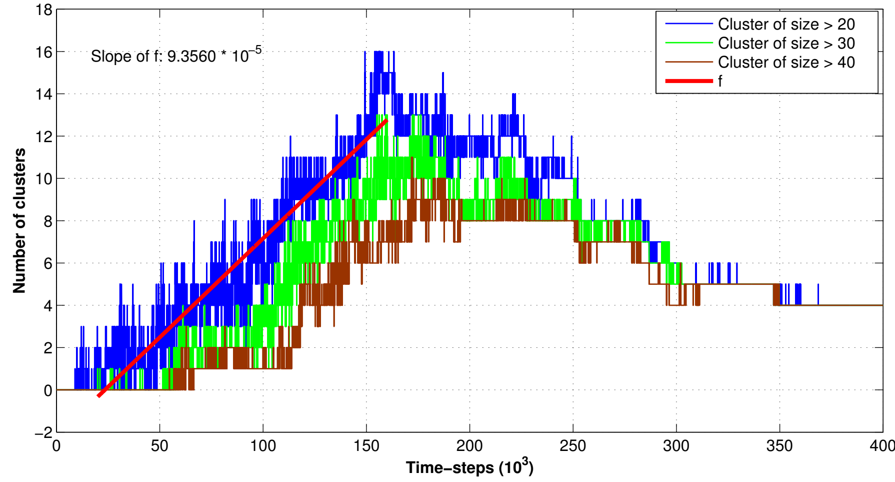



Figure 4.5.: Number of argon clusters of sizes larger than 20, 30 and 40 for 5000 particles. The critical size is 20.

In addition the results for 256000 particles are shown in 4.6 which is in good agreement to the results of Kible for 100000 particles.

For the calculation of the nucleation rate for Fig. 4.5 a linear fit f is generated using the least squares method. It is important to calculate the linear fit only for the nucleation phase where the number of clusters increases, here from about 20000 to 160000 time-steps. The slope of the curve is then used to calculate the nucleation rate.

As stated in section 2.1 the nucleation rate is defined according to the following equation:

$$J = \frac{n}{t * V} \tag{4.1}$$

In this case the domain is a cube of length $204.56$ Å and the time-step $\delta t$ is equal to $10,8$ fs. The slop of the linear fit is $9.36 \cdot 10^{-5} \frac{Cluster}{\delta t}$. Therefore, the nucleation rate can be calculated as follows:

$$J = 9.36 \cdot 10^{-5} \cdot \frac{1}{(204.56 \cdot 10^{-10} m)^3 \cdot 10.8 \cdot 10^{-15} s)} = 1,01 \cdot 10^{33} \frac{1}{m^3 \cdot s} \tag{4.2}$$

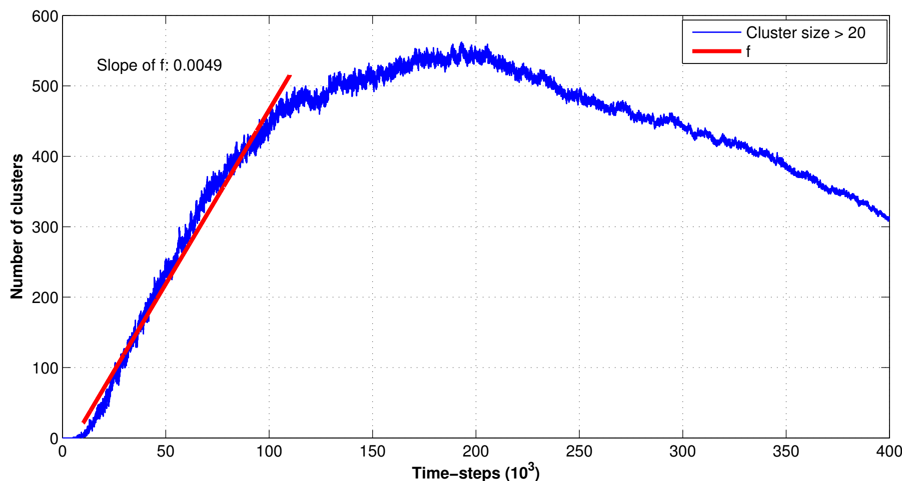



Figure 4.6.: Number of argon clusters of sizes larger than the critical size 20 for 256000 particles

For the 256000 particle experiment the nucleation rate between 10000 and 110000 time-steps can be calculated similarly. With the slope of the linear fit $4.94 \cdot 10^{-3}$, a domain length of $759, 58$ Å and a time-step of $10, 8$ fs the nucleation rate $J = 1.04 \cdot 10^{33} \frac{1}{m^3 \cdot s}$ is obtained. These results fit very well to the nucleation rates which where calculated by Kible [18] ($7.64 \cdot 10^{32} \frac{1}{m^3 \cdot s}$) and Walter [28] ($7.73 \cdot 10^{32} \frac{1}{m^3 \cdot s}$).

## 4.4. Scalability

### 4.4.1. Strong scaling with argon

Parallel algorithms are often measured by the overhead they produce in comparison to the sequential algorithm. A common criterion is the scalability which examines the runtime which can be reached with different numbers of processors. The standard approach for evaluating performance is called the strong scalability analysis. Here the system size, i.e. the number of molecules in our case, is held constant and the runtime with different numbers of processors is compared. In an ideal case the runtime should halve if the number of processors is doubled. Fig. 4.7 shows the results at the beginning of the simulation and at 200000 time-steps of MarDyn with and without the cluster detection for argon. It should be noted that we used a MPI simulation with one core as a reference and not the sequential program. However, this should not significantly affect the runtime of the cluster detection since the runtime of the communication step can be neglected for one core (see Fig. 4.8) and in this work we mainly focus on the analysis of the cluster detection.

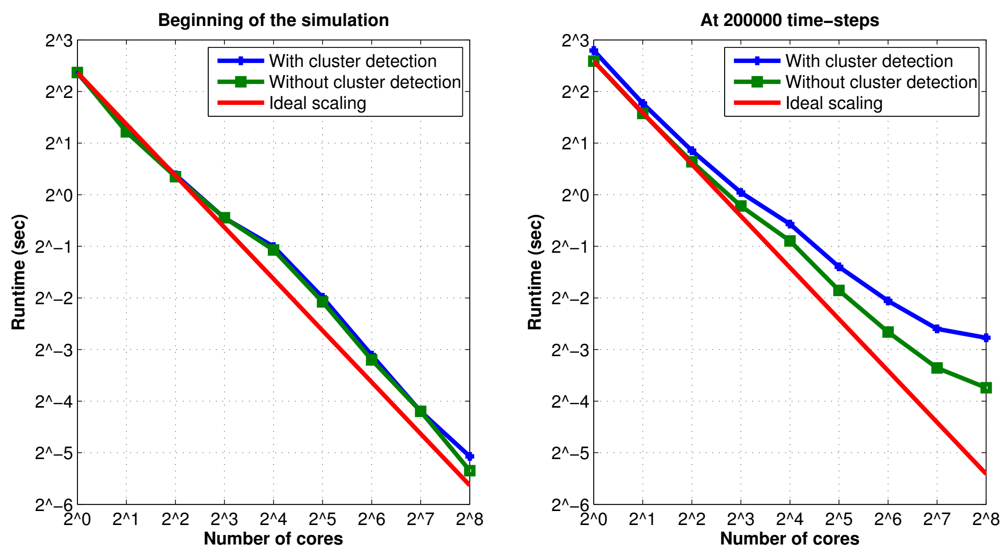



Figure 4.7.: Strong scaling with and without cluster detection using the geometric criterion at the beginning of the simulation (left) and at 200000 time-steps (right) with nucleation for 256000 argon molecules.

At the beginning of the simulation the scaling is nearly equal to the original scaling of MarDyn. This is an effect of the runtime optimization which was discussed in section 3.2.6. Only with 256 cores a slight increase in the runtime can be noticed. This could be caused by the additional traversals of the cluster detection method, since edges within the halos are searched as well and therefore more distances have to be calculated.

At 200000 time-steps the scaling is nearly equal to the scaling of MarDyn with low numbers of processors. This indicates that the depth-first search is scaling well with the domain decomposition. However, for large numbers of processors the clustering method produces an additional overhead which affects the scalability. This can be seen by the increasing gap between the runtimes.

This effect can be further analyzed with the runtimes of the three basic components of the cluster detection: the local and global depth-first search and the communication step. Fig. 4.8 shows the results for argon at 200000 time-steps.

The reason for this additional overhead is caused by the sequential global depth-first search and the communication time of the parallel algorithm. As in our implementation no dedicated master processor was used, the scalability of the whole cluster detection is affected. Furthermore the computation time of the sequential part increases with increasing number of processors, since more parallel clusters appear at the same molecular configuration. These assumptions are supported by Fig. 4.8 where the sequential part of the

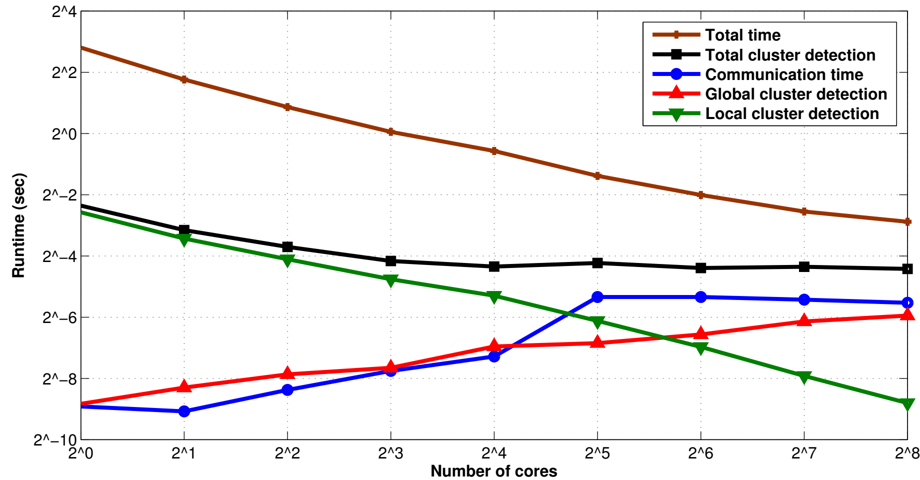



Figure 4.8.: Times for the components of the cluster detection in the strong scaling analysis with 256000 argon molecules compared to the overall runtime of the computation

algorithm and the communication time increases with the number of cores. In addition it can be seen that the scaling of the local cluster detection is similar to the scaling without the cluster detection.

Another observation of the strong scaling analysis is that MarDyn without cluster detection does not provide a good scaling performance due to the lack of a load balancing strategy. Nevertheless, in comparison to the results of Buchholtz [7] a better speedup, i.e. the factor by which the runtime is reduced in comparison to the sequential task, of about 62 is obtained for 128 processors, whereas Buchholtz only observed a speedup of about 20. This is caused by the fact that Buchholtz used artificial nucleation-like scenarios which can not be directly compared to a real nucleation scenario, as without load balancing the scalability depends heavily on the used scenario. Another aspect might be that we compared the runtimes with multiple cores to the runtime of an MPI run with one thread instead of the sequential programm.

In addition to the strong scalability, the computational overhead produced by the implementation is an important factor for the clustering algorithm. For this purpose the runtimes of MarDyn are compared with and without the cluster detection. In Fig. 4.9 the overhead for argon at the beginning of the simulation without nucleation and at 200000 time-steps with nucleation can be seen.

The overhead at the beginning of the simulation is very small due to the runtime optimizations. At 200000 time-steps the cluster detection gets more complex due to the bonds in the nucleation process. This can also be seen in Fig. 4.9. As a result the overhead is big-

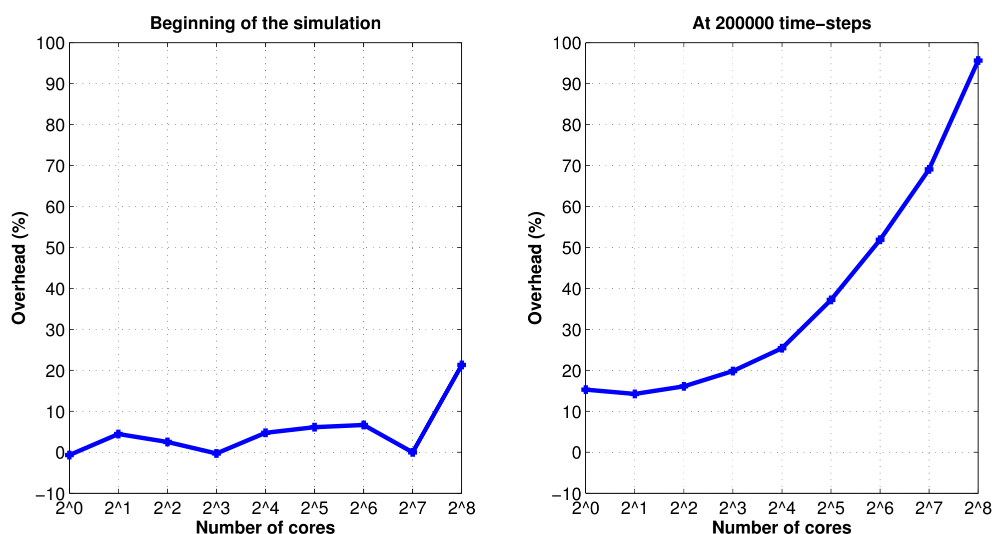



Figure 4.9.: Relative overhead without nucleation at the beginning of the simulation (left) and at 200000 time-steps with nucleation (right) for 256000 argon molecules

ger than at the beginning of the simulation. Nevertheless this overhead is mainly caused by the fact that the master processor in our work also computes the molecular simulation, i.e. the global cluster detection is a blocking operation, and the overhead is still beyond 100 % with 256 cores. Furthermore the additional distance calculations with the clustering detection leads to further overhead, especially with nucleation.

In comparison to the overhead observed by Kible [18] (see Fig. 4.10) the overhead of our implementation is much smaller, especially for low numbers of processors. Nevertheless, for very large processor counts Kible's overhead seems to decrease or at least to stay constant, whereas our overhead continuously increases. However, the increasing overhead can be circumvented by the use of a dedicated master processor.

### 4.4.2. Weak scaling of argon

Another way to examine performance is to use the weak scalability analysis. This method holds the system size per processor constant, i.e. the molecule number is increased by the same factor as the number of processor is increased. As a result the runtime should be constant in an ideal case. The result of the weak scalability method can be seen in Fig. 4.11 and Fig. 4.12.

As in the strong scalability analysis, a deviation of the runtimes can be seen with higher numbers of processors. With the results of Fig. 4.12 this additional overhead is again caused by the increasing runtime of the global cluster detection and the communication

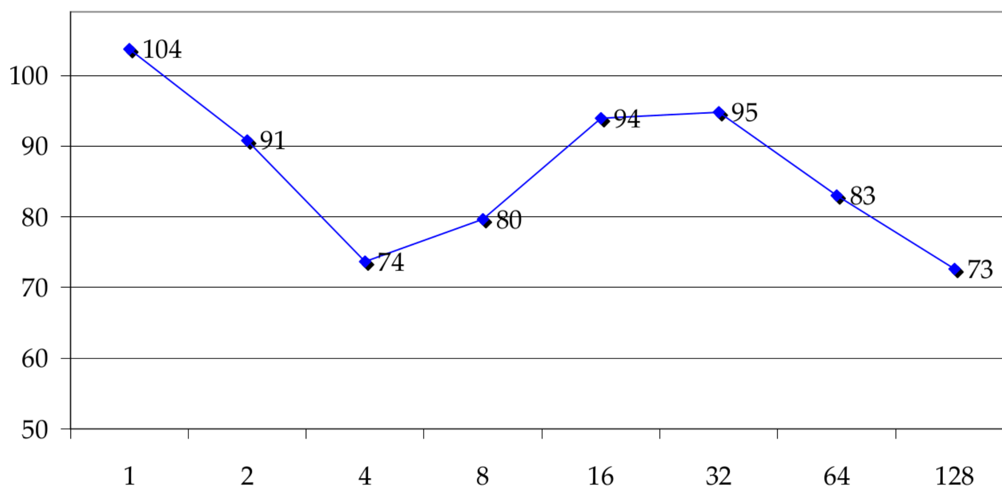



Figure 4.10.: Relative overhead of the hybrid cluster detection with nucleation for different numbers of processors observed by Kible (image from [18])

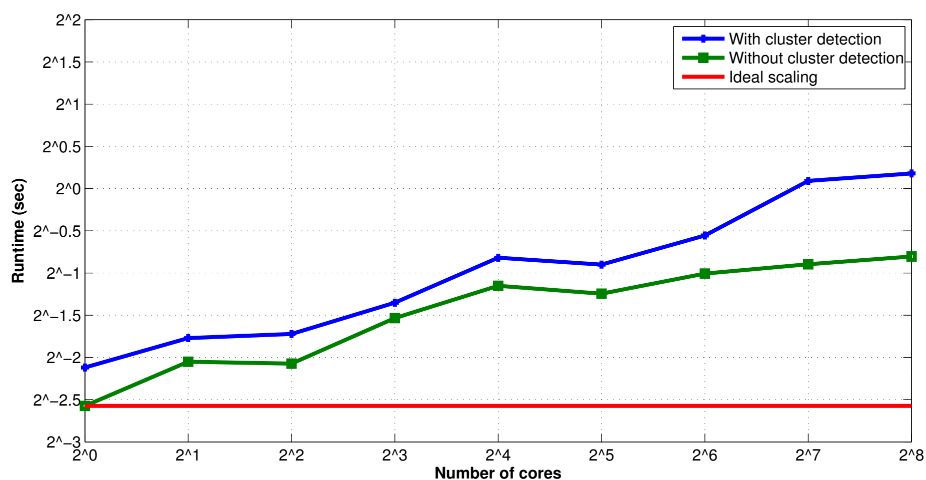



Figure 4.11.: Weak scaling with and without cluster detection using the geometric criterion at 200000 time-steps and 1000 argon molecules per processor

time. Moreover the local cluster detection scales similar to the simulation time without the cluster detection.

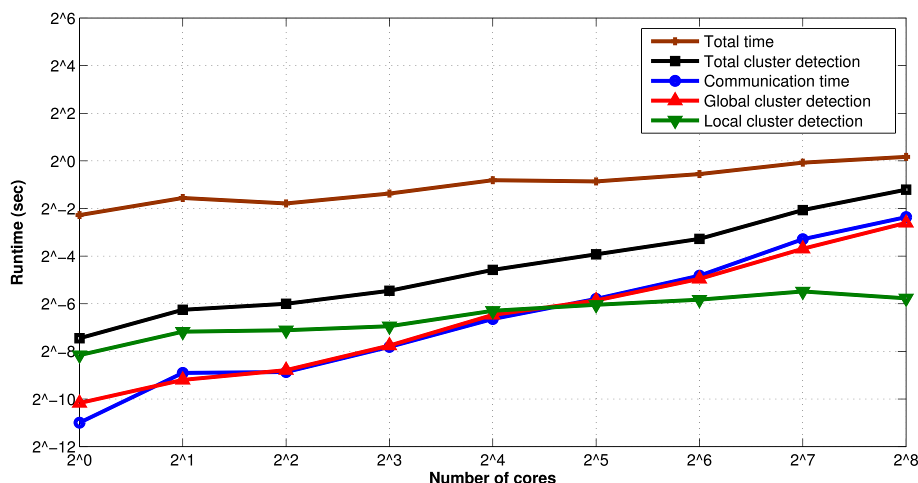



Figure 4.12.: Times for the components of the cluster detection in the weak scaling analysis with 1000 argon molecules per processor compared to the overall runtime of the computation

### 4.4.3. Performance analysis with ethane

Since the force calculation with argon is faster than for molecules with multiple Lennard-Jones centers, the additional traversal of the halo cells and the total time of the cluster detection are contributing more to the overall time. Therefore, an improved scaling behavior should be observed for more complex calculations. For this reason a testcase for ethane, which consists of two Lennard-Jones centers, was simulated as well for the strong and the weak scalability at time-steps where nucleation appears. Fig. 4.13 shows the results of the strong scaling and Fig. 4.15 the results for the weak scaling analysis with ethane. The times for the single components of the cluster detection can be seen in Fig. 4.14 and Fig. 4.16. The results show that for ethane the local cluster detection dominates the rest of the cluster detection even with 256 cores. This could be the result of the few very huge clusters which are formed quickly in the ethane simulation. Therefore only a small amount of clusters needs to be exchanged and the local cluster detection dominates the rest of the detection. Another observation is that the whole cluster detection is cheaper compared to the overall computation time for argon. This is the result of the more complex force calculation for molecules with multiple Lennard-Jones centeres. However, for larger numbers of processors the bad scaling of the global search and the communication time will likely become a problem.

Similar to the scalability results the overhead for ethane is smaller and increases slower with the number of cores in comparison to the overhead for argon. Fig. 4.17 shows the

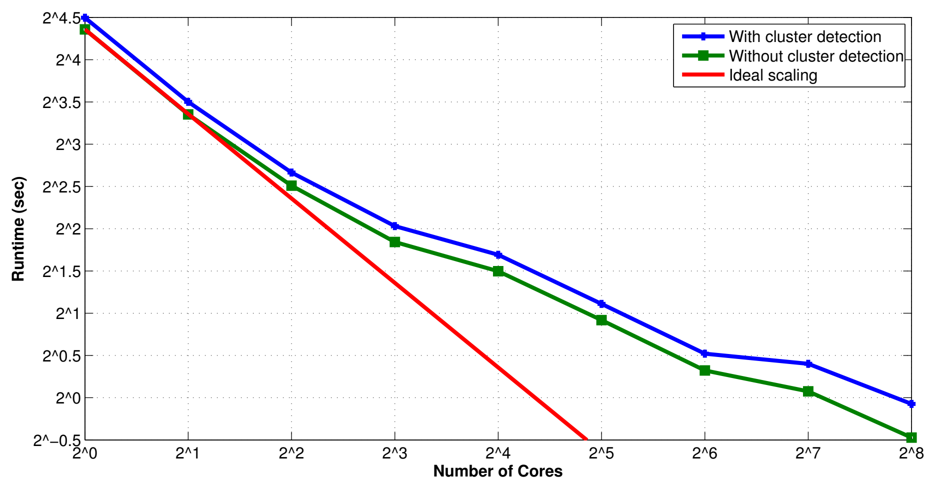



Figure 4.13.: Strong scaling with and without cluster detection using the geometric criterion at 200000 time-steps for 256000 ethane molecules

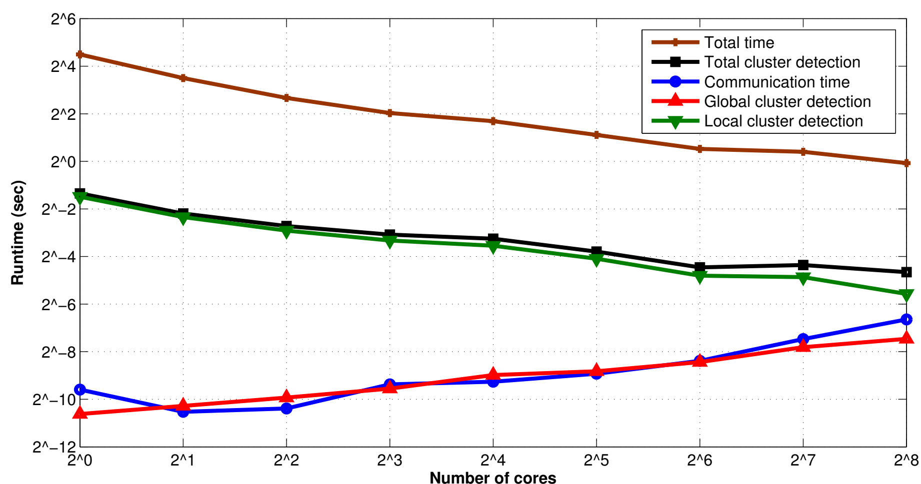



Figure 4.14.: Times for the components of the cluster detection in the strong scaling analysis with 256000 ethane molecules compared to the overall runtime of the computation

relevant changes for the simulation with nucleation at 200000 time-steps.

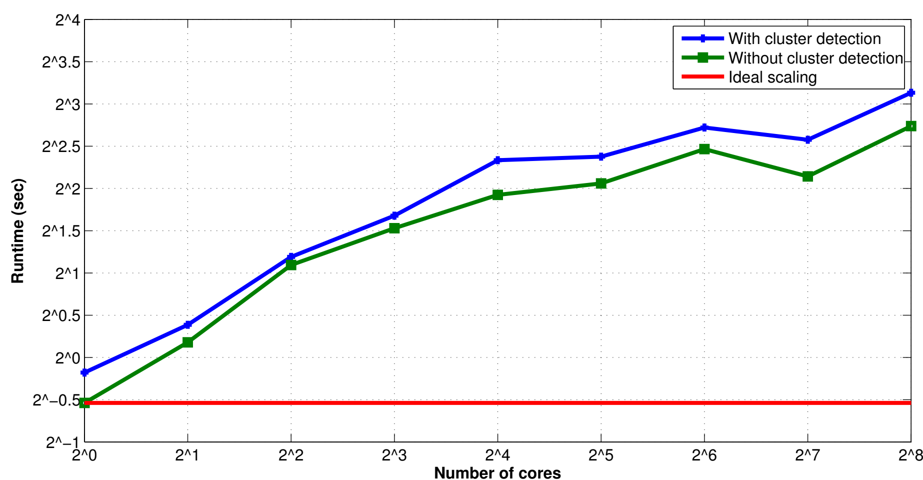



Figure 4.15.: Weak scaling with and without cluster detection using the geometric criterion at 200000 time-steps and 1000 molecules per processor for ethane

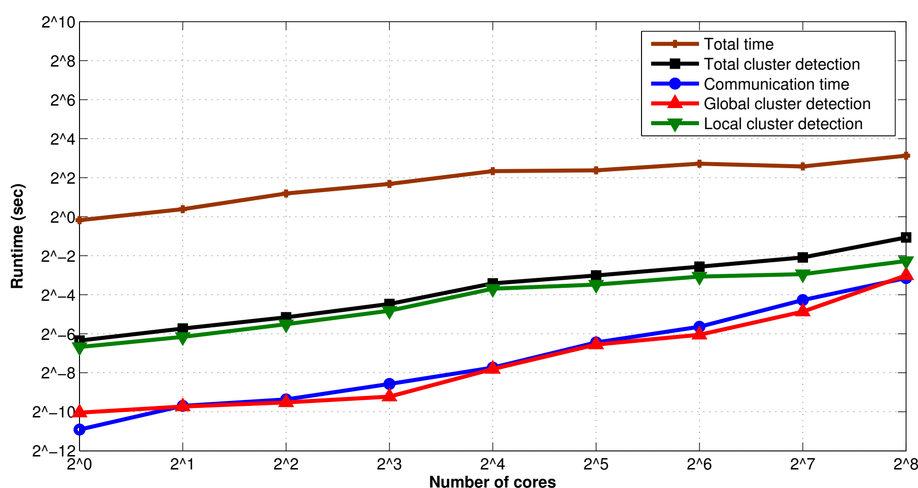



Figure 4.16.: Times for the components of the cluster detection in the weak scaling analysis with 1000 ethane molecules per processor compared to the overall runtime of the computation

Overall the overhead analysis shows promising results, especially for ethane.

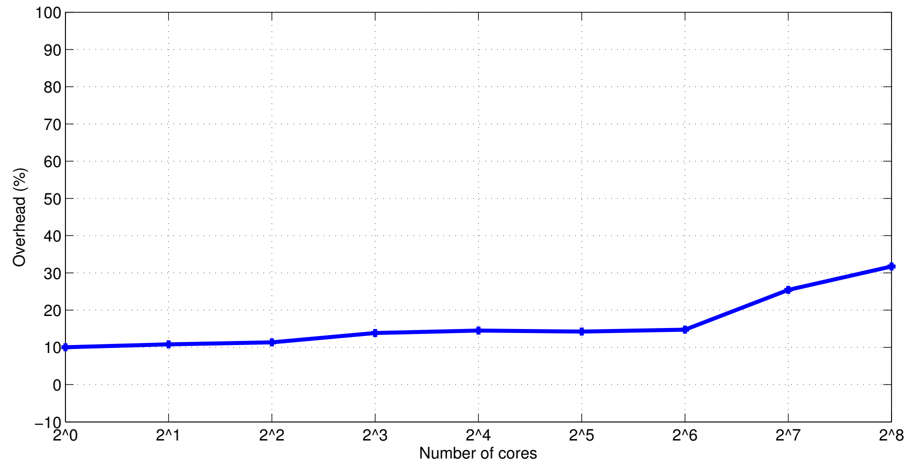



Figure 4.17.: Relative overhead with nucleation at 200000 time-steps for ethane

## 4.5. Evaluation of the clustering algorithm

### 4.5.1. Problems

In this section a conceptual problem of the parallelization algorithm is outlined. Since the whole parallel algorithm relies on the unique particle pairs, both processors have to be able to calculate the same pairs in every scenario. However, in one very rare case the design of the algorithm fails to identify the same particle pairs on every processor. One of this problematic cluster, also referred as ”banana cluster”, is visualized in Fig. 4.18.

The problem arises if one processor (here P2) can not determine that two parallel clusters which are not connected within the local domain are connected to the same cluster on the neighbouring processor. Since the neighbouring processor (here P1) correctly identifies the whole cluster, it only uses one unique particle pair (BE), whereas P2 chooses two unique pairs (BE and CD). The reason for the problem is that P2 is not able to detect the connection of B and C over A, as only the border area is saved in the halo region. As a consequence, the $createGraph$ method cannot find the right matches and the cluster detection returns wrong results for such ”banana clusters”. In the implementation this problem produces unresolved particle pairs within the $createGraph$ method (see section 3.2.5) and as a consequence some parts of a cluster cannot be merged during the global depth-first search. Although this problem exists, it is not significant for most scenarios. The problem was not detected in small scenarios, but in simulations with at least 128000 particles. Furthermore, the problem only arises in very few time-steps and mostly in coagulation processes (see Fig. 4.19). In this few situations the overall number of clusters is not significantly biased, as the complete number of clusters outweighs the number of problematic cases.

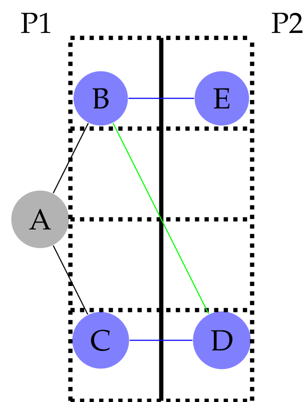



Figure 4.18.: Problematic case of the parallel algorithm for two processors: P1 and P2. P1 chooses the green bond to connect the clusters, whereas P2 chooses the two blue bonds. The dotted lines indicate the linked cells of the border layers.

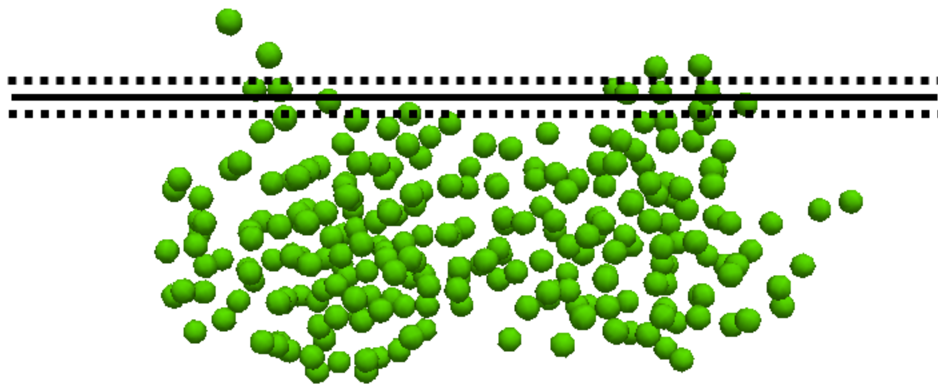


Figure 4.19.: An example for a ”banana cluster” during a coagulation process where the method fails to determine correct bonds. The domain borders are marked by the straight line and the first layer of linked cells by a dashed line.

Nevertheless for very huge scenarios, e.g. trillions of particles as mentioned in the introduction, and for molecules which form non-spherical clusters, this problem might occur more often. For this reason a solution is outlined which can be integrated in the parallel algorithm. The main idea is to build the particle pairs for subsets of the cluster in the halo and boundary layers, which are known to both processors, in order to verify that the neighboring processor can collapse down to the same number of parallel edges. One possibility to achieve this is to save all the edges from the boundary to the halo during the first depth-first search. Next, an additional depth-first search is started from all these edges, traversing only edges within the border and halo regions, to determine whether a path connecting the cluster components is fully contained in the halo and boundary layer. The results are different connected components which can be reliably detected from both processors. Hence, for every subset a unique pair is chosen which solves the problematic case. In the example shown earlier the graph for searching the new particle pairs is given in Fig. 4.20.

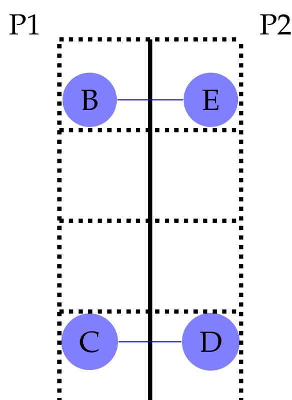



Figure 4.20.: Restricted search space within the border and halo area for the determination of the unique particle pairs. Both processors determine the same particle pairs connected by the blue bonds.

The problematic node A does not appear anymore in the process of assigning the particle pairs and therefore both processors choose the same bonds.

This solution is currently not implemented in MarDyn, but this problem will be addressed in future works. It should be noted that all performance measurements for argon were applied for simulations where this problematic case did not appear to not falsify the results. In the case of ethane, the problematic case only occurred in case of 128000 and 256000 particles.

### 4.5.2. Advantages

Despite the current problems of the method, it has several advantages. The concept of the two combined depth-first searches results in a method which is easy to understand and easy to implement. In addition, the number of communication steps is fixed and does not increase with the problem size as the method described by Kible [18]. This reduces the overall runtime and the parallel overhead.

Another advantage is the use of a depth-first search which enables the possibility to perform a local biconnectivity detection by counting the depth (see [18] and references therein). As a result a more reliable cluster detection can be implemented with little computational overhead. For parallel clusters a biconnectivity detection will be researched in the future.

Furthermore there is no need for unique cluster ids in this implementation and cluster ids do not have to be exchanged between processors as in Kible's work [18], which reduces the runtime. If such unique global cluster ids are needed in future works, this feature can be integrated in the algorithm by sending the local cluster ids in the communication step. After this step the master processor only needs to merge the local clusters and create unique cluster ids for the whole domain. These ids could then be sent back to the different processors.

Finally, the clustering algorithm is not limited to the coarse grained depth-first search. Based on the coarsening of the graph, all existing methods could be implemented to do the cluster detection. For example the iterative method could operate on the coarse graph construction. As a result the method is highly flexible and can be extended by further cluster detection algorithms.

# 5. Summary and outlook

## 5.1. Summary

A parallel algorithm for the cluster detection using a new coarsening strategy was introduced and evaluated. The performance analysis showed that the results of the methods agree with the theory and the simulation results of other works. Moreover the scalability results showed the potential of the method. Although the scaling of the local search was equivalent to the scaling without cluster detection, the communication time and the global depth-first search introduce a bottleneck for larger scenarios and the simulation with many cores. However, for more complex simulation with multiple Lennard-Jones centers, which are of most interest for industrial application, this effect only occurs for large numbers of cores. Further optimizations, as described in the next section, should reduce the effect of the global depth-first search on the scalability.

Another current weakness of the model is caused by unresolved particle pairs during the preparation of the second depth-first search which only happens in rare cases with so called "banana clusters". Even though this conflicts with the theoretical correctness, the overall results are only influenced by a very small factor. Nevertheless a solution for the problem was outlined which solves this issue.

## 5.2. Outlook

The new coarsening strategy enables the possibility for many further optimizations which can be implemented in future works. One of the most important factors is the integration of a load balancing approach, as without load balancing the scaling of the complete simulation is limited by the current implementation without cluster detection. Buchholtz [7] has done some research on load balancing within MarDyn and showed that the use of KD-trees is an efficient load balancing strategy for nucleation scenarios. The KD-Decomposition is already implemented in MarDyn. However, the cluster detection needs to be adapted to the KD-Decomposition since the number of neighbors may vary with KD-trees, in contrast to the regular Domain Decomposition with exactly 26 neighbors. Another important task is the implementation of the solution for the problematic case caused by unresolved particle pairs (see section 4.5.1).

Regarding the scalability of the cluster detection many optimizations could be applied to the existing method. For example a new data structure for the communication can be implemented with less memory consumption which would reduce the bottleneck produced

by the communication. Moreover, the global cluster detection could be done by a dedicated master processor which could also do further more complex analysis of the whole nucleation process. For very large numbers of processors, however, further master processors may be needed to not affect the scalability. Therefore, the method could be optimized by using a multi-layered hierarchical approach instead of the one-layered approach with only one master processor, as explained in section 2.4.2, which would reduce the sequential part of the algorithm. However, this approach would take more than one communication step and further performance analysis need to evaluate if the method provides efficient results within MarDyn.

# Appendix

# A. Source Code

This chapter contains excerpts from the implemented source code.

## A.1. Parallel cluster communication

```
1  unsigned long ParallelClusterCommunication::doClusterCommunication(
2        std::vector<std::pair<unsigned long,
3        std::pair<unsigned long,unsigned long> * > >& parallelCluster,
4        unsigned long numberOfClusters){
5
6     //get the number of MPI threads
7     int numberOfThreads;
8     MPI_Comm_size(MPI_COMM_WORLD, &numberOfThreads);
9     //initialize the two receive buffers
10    unsigned long *rbuf1 = new unsigned long[2*numberOfThreads];
11    //assign NULL -> only valide receive buffer at master
12    unsigned long *rbuf2 =NULL;
13    //initialize the length Buffer for the Gatherv
14    int *lengthBuffer = new int[numberOfThreads];
15
16    //get number of local parallel clusters
17    const unsigned long k = parallelCluster.size();
18    unsigned long clusters[2];
19    //number of parallel clusters
20    clusters[0]=k;
21    //number of local clusters above critical size
22    clusters[1]=numberOfClusters;
23    MPI_Gather(clusters, 2, MPI_UNSIGNED_LONG, rbuf1,
24          2,MPI_UNSIGNED_LONG,0, MPI_COMM_WORLD);
25    //get the rank of the current MPI process
26    int rank;
27    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28    //initialize the offset array for the Gatherv
29    int *offset = NULL;
30    //position for the receiving data of the Gatherv
```

```
31    int position = 0;
32    //number of all local clusters above the
33    //critical size of all MPI processes
34    numberOfClusters=0;
35    if(rank == 0){ //master
36       //calculate offset array
37       offset = new int[numberOfThreads];
38       for(int i=0; i< numberOfThreads;i++){
39          //set offset to current position
40          offset[i] = position;
41          //increase position by the number
42          //of parallel clusters of process i;
43          //size of a cluster is 53 unsigned long
44          position += (int) rbuf1[2*i] * 53;
45          //set length buffer according to
46          //the parallel clusters of process i
47          lengthBuffer[i] = (int) rbuf1[2*i]*53;
48          //add number of local clusters of process i
49          numberOfClusters += rbuf1[2*i+1];
50       }
51       //array size = last position value
52
53       rbuf2 = new unsigned long[position];
54    }
55    //collect parallel clusters in send buffer
56    unsigned long *sendBuf = new unsigned long[k*53];
57    for(unsigned long i=0; i<k;i++){
58       sendBuf[i*53] = parallelCluster[i].first;
59       for(int j=0; j<26;j++){
60          sendBuf[i*53+2*(j)+1]= parallelCluster[i].second[j].first;
61          sendBuf[i*53+2*(j+1)]= parallelCluster[i].second[j].second;
62       }
63    }
64    //sendData
65    MPI_Gatherv(sendBuf,k*53,MPI_UNSIGNED_LONG,rbuf2,lengthBuffer,
66             offset,MPI_UNSIGNED_LONG,0,MPI_COMM_WORLD);
67
68    if(rank==0){ //master
69       //write data into parallelCluster
70       for(int i=k; i<position/53;i++){
71          //read clusterSize of current parallel cluster
72          unsigned long size = rbuf2[i*53];
```

```
73          //read parallel edges
74          std::pair<unsigned long,unsigned long>  *edges =
75             new std::pair<unsigned long, unsigned long>[26];
76          for(int j=0; j<26; j++){
77             std::pair<unsigned long,unsigned long> edge =
78                std::make_pair(rbuf2[i*53+2*(j)+1],
79                rbuf2[i*53+2*(j+1)]);
80             edges[j]=edge;
81          }
82          //insert parallel cluster into parallel cluster vector
83          parallelCluster.push_back(std::make_pair(size,edges));
84       }
85       //free arrays
86       delete[] rbuf2;
87       delete[] offset;
88    }
89    //free arrays
90    delete[] rbuf1;
91    delete[] sendBuf;
92    delete[] lengthBuffer;
93    //return the total number of all local clusters
94    //above the critical size from all MPI processes
95    return numberOfClusters;
96 }
```

## A.2. Local depth-first search for the parallel algorithm

```
1  unsigned long ComponentCalculator::depthFirst(
2        std::vector<std::pair<unsigned long, std::pair<
3        unsigned long,unsigned long> *> >& parallelCluster){
4     //initial id 100 to better separate it later
5     //visually from the id 0 from clusters of size 0
6     int id = 100;
7     //current cluster size
8     unsigned long clusterSize;
9     //number of clusters above critical size
10    unsigned long clusters=0;
11    //parallel indicates if parallel cluster was found;
12    //initially parallel is set to true so
13    //that initial parallelEdes array is allocated
14    parallel = true;
15    //indicates if parallelEdges array is initialized
```

```
16  bool createdParallelEdges = false;
17  //current vertex id
18  unsigned long vertex;
19  //traverse all interacting Nodes
20  for(unsigned long i=0; i<boundedParticles.size(); i++){
21     //select next vertex
22     vertex = boundedParticles[i];
23     if(visited[vertex] == -1){ //traverse only unvisited nodes
24        if(parallel){
25           //initialize new parallelEdges array
26           parallelEdges =
27              new std::pair<unsigned long,unsigned long> [26];
28           for(int j=0; j<26;j++){
29              //default values ULONG_MAX
30              parallelEdges[j]=std::make_pair(ULONG_MAX,ULONG_MAX);
31           }
32           //reset parallel and set createdParallelEdges
33           parallel = false;
34           createdParallelEdges = true;
35        }
36        //depth first search from current vertex
37        clusterSize = search(vertex,id);
38        if(!parallel){ //no parallel cluster
39           //push the size and increase
40           //clusters if above critical size
41           size.push_back(clusterSize);
42           if(clusterSize >= criticalSize){
43              clusters++;
44           }
45        }
46        else{ //parallel cluster
47           //insert into parallel clusters
48           parallelCluster.push_back(
49              std::make_pair(clusterSize,parallelEdges));
50           //reset createdParallelEdges
51           createdParallelEdges = false;
52        }
53        //increment cluster id
54        id++;
55     }
56  }
57
```

```
58     if(createdParallelEdges){ //delete unused parallelEdges array
59        delete[] parallelEdges;
60     }
61     return clusters;
62
63 }
64
65
66 unsigned long ComponentCalculator::search(unsigned long vertex,
67          unsigned long id){
68     if(visited[vertex] != -1){ //skip visited nodes
69        return 0;
70     }
71     //boolean that checks if vertex is in halo area
72     bool inHalo = false;
73     //boolean that checks if vertex not in halo area
74     bool notInHalo = false;
75     //set vertex visited and save cluster id
76     visited[vertex] = id;
77     //set counter to one because current edge was not visited
78     //needs to be decreased again if this is a halo cell
79     unsigned long clusterSize = 1;
80
81     //search through edges which enter
82     //halo or are in the halo area
83     std::vector<std::pair<unsigned long,
84           std::pair<unsigned short, unsigned short> > >::iterator
85           itHaloBoundary(haloEdges[vertex].begin());
86
87     while(itHaloBoundary != haloEdges[vertex].end()){
88        //get locations of the molecules: halo values start at 2
89        unsigned short currentLocation =
90           ((*itHaloBoundary).second).first;
91        unsigned short nextLocation =
92           ((*itHaloBoundary).second).second;
93        unsigned long nextVertex = (*itHaloBoundary).first;
94        if(currentLocation < 2){ //parallel edge to halo
95           if(nextLocation < 2){
96              //cannot happen since it is an parallel edge
97              std::cout<< "Error";
98              throw;
99           }
```

```
100            //not in Halo since location < 2
101            notInHalo = true;
102            //parallel since next location > 2
103            parallel = true;
104            //insert edge in parallel edges and set location values
105            if(parallelEdges[nextLocation-2].first > vertex){
106               parallelEdges[nextLocation-2].first = vertex;
107            }
108            if(parallelEdges[nextLocation-2].second > nextVertex){
109               parallelEdges[nextLocation-2].second = nextVertex;
110            }
111         }
112         else{ //current vertex is in halo
113            inHalo = true;
114            if(nextLocation < 2){ //incoming edge from halo
115               parallel = true;
116               //insert edge in parallel edges and set location values
117               if(parallelEdges[currentLocation-2].first > nextVertex){
118                  parallelEdges[currentLocation-2].first = nextVertex;
119               }
120               if(parallelEdges[currentLocation-2].second > vertex){
121                  parallelEdges[currentLocation-2].second = vertex;
122               }
123            }
124         //if completly in halo do not
125         //adjust anything in parallelEdges
126
127         }
128         clusterSize +=search(nextVertex,id);
129         itHaloBoundary++;
130      }
131
132      //search through normal edges
133      std::vector<unsigned long>::iterator it(edges[vertex].begin());
134      while(it != edges[vertex].end()){
135         clusterSize +=search(*it,id);
136         it++;
137         //current vertex not in halo if normal edge exists
138         notInHalo = true;
139      }
140      //decrease counter again if vertex
141      //appeared only in halo region
```

```
142     if(inHalo && !notInHalo) {
143        clusterSize--;
144     }
145     //return cluster size
146     return clusterSize;
147 }
```

## A.3. Create graph

```
1 void ComponentCalculator::createGraph(unsigned long numParticles,
2        std::vector<std::pair<unsigned long,
3        std::pair<unsigned long,unsigned long> * > >& parallelCluster,
4        std::vector<unsigned long>  *parallelEdges){
5     //iterator for the parallel clusters
6     std::vector<std::pair<unsigned long,
7        std::pair<unsigned long,unsigned long> * > >::iterator
8        it(parallelCluster.begin());
9     //cluster id an loop variable
10    unsigned long i = 0;
11    //number of unresolved matches
12    unsigned long unresolvedMatches=0;
13    //sizes of the clusters
14    //clusterSize[i] = size of the i-th cluster in parallelClusters
15    parallelClusterSizes.resize(parallelCluster.size());
16    while(it != parallelCluster.end()){
17       //assign cluster size
18       parallelClusterSizes[i] = (*it).first;
19       //get the vector of parallel edges
20       std::pair<unsigned long,unsigned long> * edges = (*it).second;
21       for(int j=0; j<26;j++){
22          if(edges[j].first != ULONG_MAX){ //only check existing edges
23             //indicates if current bond is already contained
24             bool match=false;
25             //position of match
26             unsigned long position = 0;
27             //search for existing match
28             for(unsigned long k=1;
29                   k<matchArray[edges[j].first].size();k+=2){
30                if(matchArray[edges[j].first][k] == edges[j].second
31                   && matchArray[edges[j].first][k+1] != ULONG_MAX){
32                   //match found
33                   match = true;
```

```
34                      position = k+1;
35                      break;
36                  }
37              }
38              if(!match){
39                  //add particle pair to the sparse matrix
40                  matchArray[edges[j].first].push_back(edges[j].second);
41                  //add cluster id
42                  matchArray[edges[j].first].push_back(i);
43                  //increase the unresolved matches
44                  //counter at position 0
45                  matchArray[edges[j].first][0]++;
46                  //add particle pair to the sparse matrix
47                  matchArray[edges[j].second].push_back(edges[j].first);
48                  //add cluster id
49                  matchArray[edges[j].second].push_back(i);
50                  //increase the unresolved matches
51                  //counter at position 0
52                  matchArray[edges[j].second][0]++;
53                  //increase total counter of unresolved matches
54                  unresolvedMatches++;
55              }
56              else{ //match found
57                  //save new edge between the
58                  //two parallel clusters
59                  //in parallel edges matchArray
60                  parallelEdges[matchArray[edges[j].first]
61                      [position]].push_back(i);
62                  parallelEdges[i].push_back(
63                      matchArray[edges[j].first][position]);
64
65                  //make found bonds invalid
66                  matchArray[edges[j].first][position] = ULONG_MAX;
67
68                  //search for second entry and make it invalid
69                  for(unsigned long k=1;
70                          k<matchArray[edges[j].second].size();k+=2){
71                      if(matchArray[edges[j].second][k]==edges[j].first){
72                          position = k+1;
73                          break;
74                      }
75                  }
```

```
76                  matchArray[edges[j].second][position] = ULONG_MAX;
77                  //decrease the unresolved matches
78                  //value of the vertex;
79                  //delete vector in matchArray
80                  //if no unresolved match left
81                  matchArray[edges[j].first][0]--;
82                  if(matchArray[edges[j].first][0] == 0){
83                      matchArray[edges[j].first].clear();
84                      matchArray[edges[j].first].push_back(0);
85                  }
86                  matchArray[edges[j].second][0]--;
87                  if(matchArray[edges[j].second][0] == 0){
88                      matchArray[edges[j].second].clear();
89                      matchArray[edges[j].second].push_back(0);
90                  }
91                  //decrease the total unresolved matches
92                  unresolvedMatches--;
93              }
94          }
95      }
96      it++;
97      i++;
98  }
99  if (unresolvedMatches != 0){  //error case
100     //delete all edges
101     for(unsigned long i=0; i<numMolecules; i++){
102         if(array[i][0] != 0){   //delete remaining pairs
103             std::cout << "\n";*/
104             array[i].clear();
105             array[i].push_back(0);
106         }
107     }
108
109     //throw exception or ignore
110 }
111 }
```

# Bibliography

[1] Boost library - documentation. http://www.boost.org/doc/. (accessed on 30.08.2014).

[2] Paraview - documentation. http://www.paraview.org/documentation/. (accessed on 30.08.2014).

[3] Munich centre of advanced computing. http://www.mac.tum.de/wiki/index.php, February 2014. (accessed on 01.09.2014).

[4] Supermuc petascale system. http://www.lrz.de/services/compute/supermuc/, August 2014. (accessed on 01.09.2014).

[5] F. F. Abraham. *Chapter 1 - The Nature of the Nucleation Process*. Academic Press, 1974.

[6] P. Allen and D.J. Tildesley. *Computer simulation of liquids*. Oxford science publications. Clarendon Press, 1987.

[7] M. Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*. Verlag Dr. Hut, 2010.

[8] M. Buchholz, H.-J. Bungartz, and J. Vrabec. Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. *Journal of Computational Science*, 2(2):124 – 129, 2011. Simulation Software for Supercomputers.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[10] W. Eckhardt, A. Heinecke, R. Bader, M. Brehm, N. Hammer, H. Huber, H.-G. Kleinhenz, J. Vrabec, H. Hasse, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, A. Bode, and H.-J. Bungartz. *591 TFLOPS Multi-trillion Particles Simulation on SuperMUC*, volume 7905 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[11] A. Fladerer. *Keimbildung und Tröpfchenwachstum in übersättigtem Argon-Dampf - Konstruktion einer kryogenen Nukleationspulskammer*. PhD thesis, University of Cologne, 2002.

[12] L. K. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1800 of *Lecture Notes in Computer Science*, pages 505–511. Springer Berlin Heidelberg, 2000.

[13] Message Passing Interface Forum. Mpi: A message-passing interface standard version 3.0, 2012.

[14] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics*. Springer Verlag, 2007.

[15] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[16] T. L. Hill. Molecular clusters in imperfect gases. *The Journal of Chemical Physics*, 23(4):617–622, 1955.

[17] M. Horsch, J. Vrabec, M. Bernreuther, S. Grottel, G. Reina, A. Wix, K. Schaber, and H. Hasse. Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. *The Journal of Chemical Physics*, 128(16):–, 2008.

[18] R. Kible. Clusteringalgorithmen zur keimdetektion in gasen. Master's thesis, University of Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2006.

[19] F. Knop and V. Rego. Parallel labeling of three-dimensional clusters on networks of workstations. *Journal of Parallel and Distributed Computing*, 49(2):182 – 203, 1998.

[20] L. A. Pugnaloni and F. Vericat. New criteria for cluster identification in continuum systems. *The Journal of Chemical Physics*, 116(3):1097–1108, 2002.

[21] J. Schluttig. A parallel cluster algorithm for monte carlo simulations applied to model dna systems. Leipzig University.

[22] R. P. Sear. Nucleation: theory and applications to protein solutions and colloidal suspensions. *Journal of Physics: Condensed Matter*, 19(3):033101, 2007.

[23] F. H. Stillinger. Rigorous basis of the frenkel-band theory of association equilibrium. *The Journal of Chemical Physics*, 38(7):1486–1494, 1963.

[24] G. Stockman and L. G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[25] S. D. Stoddard. Identifying clusters in computer experiments on systems of particles. *Journal of Computational Physics*, 27(2):291 – 293, 1978.

[26] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, 1982.

[27] L. Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[28] J. Walter. Molekulare simulation und visualisierung der keimbildung in mischungen. Master's thesis, University of Stuttgart, June 2006.

[29] K. Yasuoka and M. Matsumoto. Molecular dynamics of homogeneous nucleation in the vapor phase. i. lennard-jones fluid. *The Journal of Chemical Physics*, 109(19):8451–8462, 1998.