Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Efficient Verlet-List Implementation for KNL Processors

Daniel Langer

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Efficient Verlet-List Implementation for KNL Processors

Effiziente Verlet-Listen Implementierung für KNL Prozessoren

Author: Daniel Langer
Supervisor: Univ.-Prof. Dr. rer. nat. Hans-Joachim Bungartz
Advisor: Steffen Seckler M.Sc. (hons)
Submission date: June 15th, 2018

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

Munich, June 15th, 2018 .....

Daniel Langer

Abstract

The increasing power of computer systems over the last decades has influenced many areas of our civilisation. It has changed our daily life but also had a broad impact on science and research. Here, computers are, for example, used to simulate natural and physical processes. The scientific field of Molecular Dynamics (MD) is about the simulation of molecular particles in a spatial system where the individual particles interact through intermolecular forces to each other. As the number of particles usually is extraordinary high, efficient numerical approaches and algorithms are necessary. One of these approaches uses so-called Verlet lists. The goal of this Bachelor's thesis was to extend an existing MD Simulation with a Verlet list implementation and test it on Intel Xeon Phi Processors.

Zusammenfasung

Die Leistungsfähigkeit von Computern hat in den letzten Jahrzehnten exponentiell zugenommen und damit viele Bereiche stark verändert, sowohl im täglichen Leben als auch in Wissenschaft und Forschung. Dort werden Computer unter anderem verwendet um natürliche physikalische Prozesse zu simulieren. Ein Forschungsfeld in diesem Bereich ist die Molekulardynamik (MD), die sich mit der Simulation von molekularen Partikeln in einem räumlichen System beschäftigt. wobei die einzelnen Partikel über atomare Wechselwirkungen gegenseitig Kräfte aufeinander ausüben. Da die Anzahl der Partikel üblicherweise sehr hoch ist, sind effiziente numerische Herangehensweisen notwendig. Einer dieser Ansätze sind die sog. Verlet-Listen. Im Rahmen dieser Bachelorarbeit wurde eine bestehende MD Simulation um eine Variante mit Verlet-Listen erweitert und auf Intel Xeon Phi Prozessoren getestet.

Contents

1. Introduction

The increasing power of computer systems over the last decades has led to rising importance and influence of computer simulations in many areas of science and research. Here, often one is interested in the progress of physical processes. Examples, therefore, are the flow of water through a channel or the behaviour and distribution of smoke in a room. In Civil Engineering, one is interested in the stresses on the individual components of a building and their resulting deformations whereas, in medicine, it could be lifesaving to predict the pressure that strains the ventricles of a human heart due to blood flow. One possible approach is to construct a physical replication of the real scenario with much smaller scale and to simulate the process on that model. Hence in many cases, this is expensive either in time as in money, and often, as in the medical example from above, it is quite impossible. Another disadvantage is that in most cases it is only possible to determine physical quantities like force, velocity, temperature or pressure on specific points by measuring instead of their distribution along the whole scenario. Computer simulations are a new and different approach to deal with the problems mentioned. Instead of building a physical model, one tries to develop a model on the computer and to simulate the process there. From that, several challenges arise. First of all, all laws of nature that are relevant, as well as their correlations, have to be known and formulated by mathematical equations. Second, these equations have to be solved. In most cases, this is not possible in an analytical way, so numerical methods are used to approximate the analytical solution. To achieve useful results, it is necessary that the model is sufficiently subtle and describes the real scenario accurately enough. As raising fineness and accuracy of the model usually lead to a raising computational effort of at least polynomial order, it is necessary either to use efficient numerical algorithms and data structures but also to make a compromise between accuracy and computational effort. A good model is precisely as subtle that the simulation yields useful results in a reasonable time. The scientific field of Molecular Dynamics (MD) is about the numerical simulation of molecular particles in a domain system. This system can, for example, be a spatial room, but also a 2-dimensional area is possible. The mentioned example of smoke in a room would be a possible application for MD (see also Fig.1.1). As the number of particles usually is exceptionally high, MD is challenging, even for today's supercomputers. Most approaches to deal with molecular dynamics use domain decomposition methods, i.e. they divide the domain into smaller subdomains and assign each particle to the subdomain corresponding to the position of the particle in the domain. The calculations for the individual subdomains then mostly can be done independently of each other and therefore in parallel. That makes MD well suited for high-performance computing (HPC) on today's many-core processors. The Xeon Phi processor (also named Knights Landing, KNL), which Intel released in June 2016 is a many-core processor with 64 up to 72 cores, designed for high-performance computing on supercomputers and therefore well appropriated for MD simulations. In September 2017 the Leibnitz-Supercomputing-Centre (LRZ) in Garching near Munich installed a Linux cluster

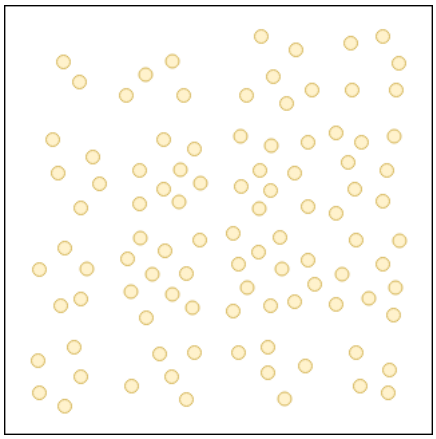


Figure 1.1.: Moving particles

named CoolMuc3, consisting of 148 Intel Xeon Phi processors. In the scope of this Bachelor's thesis an approach for MD simulations, using Verlet lists in combination with linked cells was implemented and tested on this cluster. As a basis for the implementation, the program AutoPas was used, which is currently developed at the Technical University of Munich. The aim was to measure the performance of this approach. In chapter 2, molecular dynamics will be introduced, including linked cells and Verlet lists. Chapter 3 will give a quick overview of the Intel Xeon Phi processor and the Linux cluster CoolMuc3. It will also be shown, what performance gains are generally possible with this processor. For that, another MD simulation called Lammps was tested on CoolMuc3 and compared to other results. In chapter 4, the developed implementation of the Verlet lists approach will be described, and how it was integrated into AutoPas, the performance results will be presented in chapter 5. The last chapter will give an outlook for future works, that could be done using the Verlet lists approach.

2. Molecular Dynamics

The simulation of molecular dynamics (MD) is a so-called N-Body problem. That means it deals with the problem of the physical interaction of a certain amount of objects in a domain system and their physical influence to each other. In MD simulations these objects often are called either particles or molecules, their number typically is exceptionally high while the individual particles itself are rather small, mostly they are simple chemical molecules. The domain system can, for example, be a spatial room or a two-dimensional area. One main characteristic, which distinguishes MD from other numerical simulations like fluid dynamics or the Finite-Element-Method, is that each particle is associated with a position $\vec{r}(t)$ and a velocity $\vec{v}(t)$ which are bound to it during the whole simulation. The aim is now to simulate the movement of the particles i.e. to compute their positions $\vec{r}(t)$ and velocities $\vec{v}(t)$, additionally aims could for example be the resulting distribution of pressure or temperature in the domain. The computation is done by numerical integration over time. There exist many classical methods for numerical integration, ranging from explicit and implicit Euler methods up to the so-called Leapfrog method which is frequently used in MD because of its numerical stability [1]. With the Leapfrog method the positions and velocities of the molecules at each timestep are calculated as

$$\begin{aligned} \vec{v}(t + \frac{\Delta t}{2}) &= \vec{v}(t - \frac{\Delta t}{2}) + \Delta t \cdot \vec{a}(t) \\ \vec{r}(t + \Delta t) &= \vec{r}(t) + \Delta t \cdot \vec{v}(t + \frac{\Delta t}{2}) \end{aligned} \tag{2.1}$$

To calculate the acceleration $\vec{a}(t)$ of the molecules, it is necessary for each molecule to determine the total force that acts on it due to the influence of all other molecules. For every two molecules there exists a physical force between them, which can either be attractive or repulsive depending on the distance between them. If the molecules are close to each other, the repulsive part, resulting from overlapping electron orbitals, prevails. If the distance is rather far, the attractive part, resulting mainly from Van-der-Waals forces overweights. There are many approximations to describe the resulting force of these repulsive and attractive parts, the Lennard-Jones-(12-6) potential is one of them, which is often used in molecular dynamics. The Lennard-Jones-(12,6) potential approximates the potential between two particles and is given as

$$U_{LJ}(r) = 4\epsilon \cdot \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.2}$$

where r denotes the actual distance between the two particles. σ and ϵ are parameters, which depend on the types of the two concrete molecules. σ denotes the distance, at which the potential becomes zero and ϵ the minimal potential in Joule. The Lennard-Jones potential for $\sigma = 0.8$ and $\epsilon = 1.5$ is exemplary shown in Fig. 2.1. According to [1], from this potential the force between the molecules can be derived as

$$F(r) = -\frac{dU_{LJ}}{dr} = \frac{24\epsilon}{r} \cdot \left[2 \cdot \left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.3}$$

Therefore, to calculate the total resulting forces that invoke on each molecule at a timestep, a simple way would be to calculate the Lennard-Jones potentials for all molecule-molecule pairs and sum up the resulting forces for each molecule. Because of Newton's 3rd law ('Actio = Reactio'), the force that a particle p_1 invokes on another particle p_2 is equal and directed into

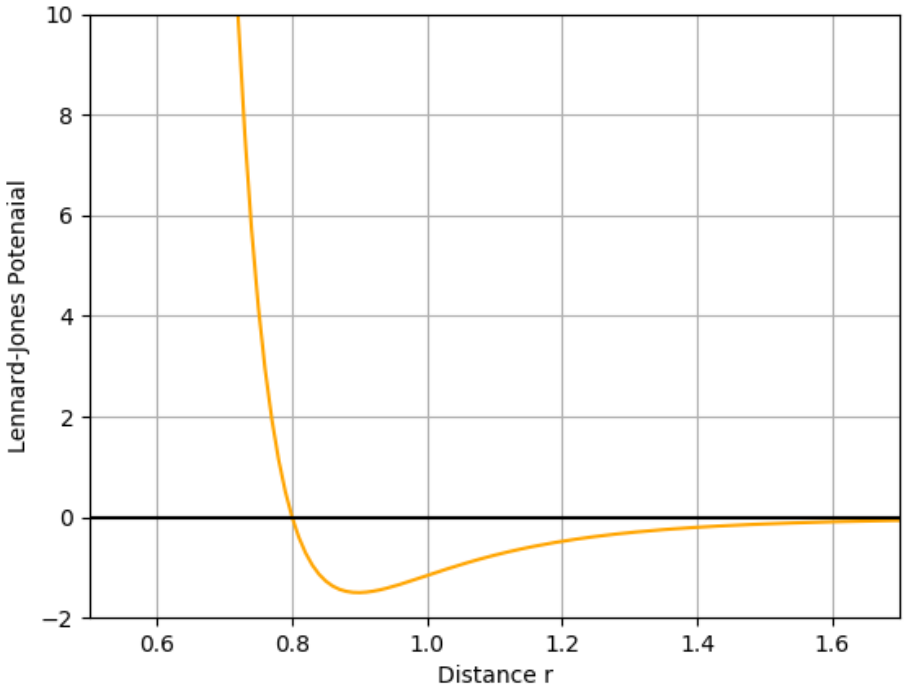


Figure 2.1.: Lennard-Jones potential for $\sigma = 0.8$ and $\epsilon = 1.5$

the opposite direction to the force, that p_2 invokes on p_1. So for each particle-particle pair, the force between them has just to be calculated once, not twice. Nevertheless, this results in a computational effort of $\mathcal{O}(N^2)$ where N is the total amount of molecules in the system and is therefore not very practical. Fortunately, there are better approaches based on the observation that the Lennard-Jones potential and its deviation decrease rapidly with increasing distance (see Fig. 2.1). So two particles whose distance is rather large have a negligible influence on each other. Hence it is useful to define a cutoff radius r_c. Then for every molecule, all other molecules whose distance is greater than the cutoff radius are ignored. If the particles are almost equally distributed in the spatial system and the cutoff radius r_c is chosen inversely proportional to the total amount of particles N, this results in an improved computational effort of $\mathcal{O}(N)$. Two approaches for MD, which make use of a cutoff radius are linked cells and Verlet lists. In the following, both will be briefly described.

2.1. Linked Cells

As already mentioned, if a cutoff radius r_c is used, then for a specific particle p, all other particles, whose distance to p is greater than r_c are considered to have a negligible influence on p and are therefore ignored. The basic idea behind the linked cells method is to divide the domain into cubic resp. square cells along a Cartesian grid with edge lengths greater or equal to r_c (usually the edge length is chosen equal to r_c). Then, for every particle p in a particular cell c, only other particles in cell c itself or other cells neighbouring c can be within the cutoff radius r_c. This is exemplarily shown in Fig 2.2 where the domain is a two-dimensional grid consisting of 16 cells. To determine all particles that are within the cutoff radius of the red particle and thus have a non-negligible influence on it, only the violet particles in the nine grey-coloured cells have to be considered and checked. The green particles in the remaining

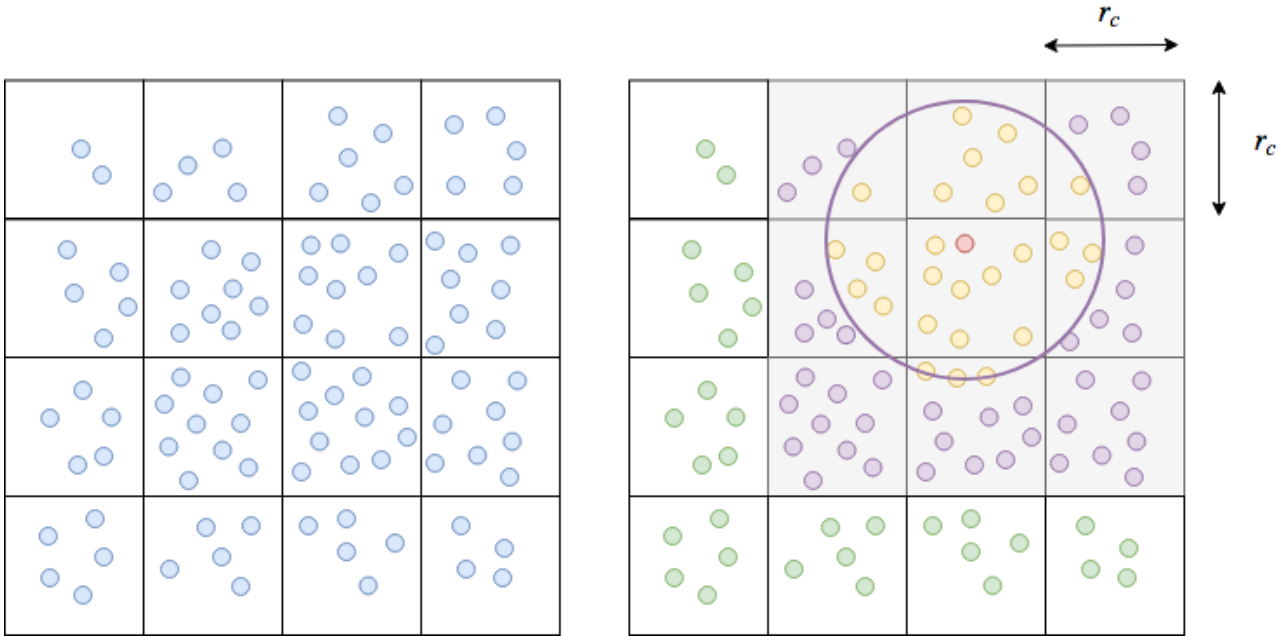


Figure 2.2.: Decomposition into linked cells

cells can be ignored. As already mentioned, this results in an improved complexity of $\mathcal{O}(N)$ in contrast to $\mathcal{O}(N^2)$ for an all-particle-pairs check. Another advantage of this locality of dependency is that it allows to divide the domain into subdomains which can be processed mostly independent to each other. This offers the possibility to process the particles of different subdomains in parallel. That makes the linked cells method well appropriated for today's multi-core resp. many-core processor architectures. In Fig. 2.3 on the left, a possible domain decomposition of a two-dimensional area is shown. The domain consists of 144 linked cells, divided into nine subdomains, each containing 16 cells. These nine subdomains now could be assigned to nine processor cores and be processed in parallel. However, there are restrictions, which is shown based on the dark-green marked cell of the red domain in Fig. 2.3. Three of the eight neighbouring cells that this cell has to check for influencing particles do belong to a different domain (violet) and are thus running on another processor. Another problem is that particles can migrate from one cell to another at a timestep and thus it is possible that particles migrate from one cell in one domain to another cell in a different domain. Hence, each domain has to keep a copy of its directly neighbouring cells from neighbouring domains and of all particles in there, as shown in Fig. 2.3 on the right. These cells are called halo cells, the particles within are often referred to as ghost atoms. Hence, after each timestep, the processors have to communicate with each other and exchange the data of their halo cells and also exchange migrated particles. For the domain decomposition, it is not necessary to distribute the linked cells into square subdomains like in Fig. 2.3. The distribution into subdomains rather should be done in a way that two main properties are fulfilled. The first one is to minimise the inter-process communication between the processors. The second one is to distribute the load equally on the processors so that all processors take approximately the same time to calculate their timesteps. That is important because, at each timestep, the processors have to wait until all other processors have finished their step as well, to exchange the data of their halo cells, so the slowest processor is decisive. The field that deals with finding suitable distributions of the cells onto the subdomains is called Load Balancing. Space-Filling curves and k-d-trees are examples for load balancing approaches. For further information on load balancing, see, for example [4].

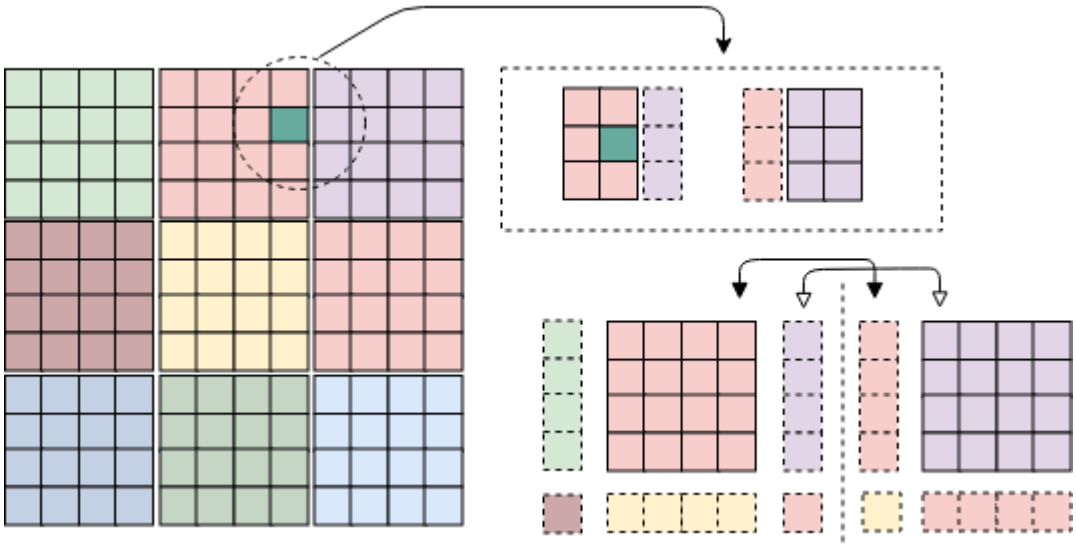


Figure 2.3.: Decomposition of a two-dimensional domain (left) and halo cells (right)

2.2. Verlet Lists

Although with linked cells a massive performance gain can be reached, there are still improvements possible. If one looks again at Fig. 2.2 and the considered red particle, one can see that from all checked particles (violet), only a small part (yellow) is indeed within the cutoff radius. For a two-dimensional area, assuming almost equally distributed particles, nine cells have to be checked, which results in an area of $A_{cells} = 9 \cdot r_c^2$, whereas the area for particles within the cutoff radius is $A_{r_c} = r_c^2 \cdot \pi$. So, only $^{A_{r_c}}/_{A_{cells}} \approx 35\%$ of the checked particles are statistically within the cutoff radius, whereas $\approx 65\%$ are checked unnecessarily. In a three-dimensional domain, where 27 cells have to be checked, the ratio is $^{V_{r_c}}/_{V_{cells}} = \frac{4/3 \cdot \pi \cdot r_c^3}{27 \cdot r_c^3} \approx 15\%$, so statistically 85% of the checked particles are checked unnecessarily. Another approach is the use of Verlet lists (also named neighbour lists). For each particle p, a Verlet list is used, which stores all particles (usually just the unique ID-numbers of the particles) that are within a certain distance r_v to p. If one chose $r_v = r_c$, the lists would store, for each particle, all other particles, that are within the cutoff radius, what would be exactly what is needed. The problem would be, that at each timestep, particles can leave the cutoff domain as well as new particles can enter it. Therefore, the lists had to be recomputed at each timestep which would not result in a performance gain.
A better way is to set $r_v = r_c + d_{skin}$, as shown in Fig. 2.4. Then, for every particle, all other particles within the cutoff radius plus additional particles within a skin layer are stored in its Verlet list. Hence, at the moment a Verlet list was built, it stores all particles that are within the cutoff radius plus all

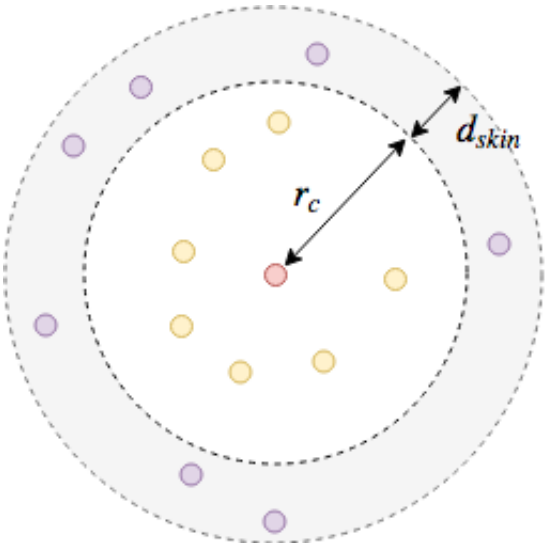


Figure 2.4.: Cutoff domain and skin layer for Verlet lists

particles that are outside of it but can enter it within a certain time $\Delta t_v = {}^{d_{skin}}/_{v_{max}}$, where v_{max} denotes the maximum velocity of all particles. As well, the list will contain every particle, which was at the beginning of the list-build within the cutoff domain and left it during Δt_v. Hence, the Verlet lists only need to be rebuild after Δt_v but not after each timestep. Therefore, for the computation of the total resulting force for a particle p , one just needs to check, for all particles in its Verlet list, if they are indeed within the cutoff domain or just within the skin domain, all other particles that are not in the list can be ignored during all timesteps within Δt_v. So the basic procedure, using Verlet lists is shown in pseudo-code in Listing 2.1.

```
while(t < t_end)
{
  for(int i = 0; i < N; i++) // iterate over all particles in the domain resp.
      subdomain
  {
      lst = vlists[i]; // assign the Verlet list of particle p_i to lst

      for (Particle p : lst) // iterate over all particles in lst
      {
          if (dist(p[i], p) < cutoff_radius) // check if particle p is within the
              cutoff radius of p_i
          {
            UpdateForces(); // Update the forces for p_i and p and do further
                calculations possibly needed
          }

      }
  }

  t = t + Δt;
  if (CheckRebuildVerletLists()) // check, if the Verlet lists have to be rebuild ..
      RebuildVerletLists();  // .. if yes, then rebuild them
}
```

Listing 2.1: Basic scheme with Verlet lists

3. The Intel Xeon Phi Processor

Today, modern graphics cards are fitted with powerful processors itself, additionally to the general CPU. These so-called GPU's [1] have the aim to unburden the CPU. They are specialised and optimised for graphical operations. As these often consists in applying one simple operation (for example an addition) to a high amount of data (for example, the pixels of an image), GPU's are optimized for SIMD[2]-Instructions in a high parallel way. Therefore, GPU's are typically many-core processors that consist of many simple cores. As an example, the Fermi architecture, developed by NVIDIA, consists of 16 streaming multiprocessors (SM) with 32 cores each, so it has a total amount of 512 cores. It reaches a peak performance of 1.5 TFLOPS[3] [9]. These architectures are therefore very interesting for scientific simulations like molecular dynamics. NVIDIA also recognised this potential, and in June 2007, a programming interface named CUDA was released, which offers the possibility to run the GPU of a graphics card as a coprocessor to the CPU and hence use it for general-purpose programming.

3.1. Knights Corner and Knights Landing

Programming on GPU's like the Fermi architecture has the disadvantage, that these architectures have their own ISA [4] and thus are incompatible to the Intel x86-64 architecture, so it takes much effort to make a program runnable with such a GPU as a coprocessor. In 2012, Intel released the first generation of a new many-core processor, called Intel Xeon Phi (codename Knights Corner). It is not a standalone processor but a coprocessor and needs to be used in combination with a central processor like Intel Xeon. The advantage of this processor is that its ISA is the x86-64 architecture. Thus it is compatible to other CPUs like the Intel Xeon processor. The amount of cores per processor ranges from 57 up to 61, and every core is, due to hyper-threading, able to run up to four hardware threads, so up to 244 hardware threads are possible per processor. The Knights Corner processors do not support the SIMD instruction set extensions MMX, SSE and AVX, instead they offer their own SIMD instruction set extension called Intel-IMCI [5] which consists of 32 512-bit large vector registers [8]. In 2016, the second generation, called Knights Landing (KNL) was released by Intel. In contrast to its predecessor, this is a full standalone processor. Variants are available with 64, 68 and 72 cores, so up to 288 hardware threads are available per processor. MMX, SSE, AVX as well as the AVX-2 SIMD instruction set extensions are supported. Additionally, they support the new AVX-512 vector instruction set, which offers a register file of 32 SIMD registers, each 512 bit wide. Each processor is suited with an on-package high bandwidth MCDRAM[6] memory with a size of 16 GB, which can be used either as third level cache or as memory. In Fig.3.1, a scheme of a Xeon Phi processor with 72 cores is shown, the cores are distributed into 36 tiles, two cores in each tile. Every two cores in a tile have to share their L2-cache. With 1.3 GHz, the clock cycle of the Xeon Phi is relatively low, compared to other

[1] **G**raphics-**P**rocessing-**U**nit
[2] SIMD=**S**ingle-**I**nstruction-**M**ultiple-**D**ata
[3] FLOP = **F**loating **P**oint **O**peration **P**er **S**econd, 1 TFLOP = 10^{12} FLOPS
[4] **I**nstruction-**S**et-**A**rchitecture
[5] Intel-**I**nitial **M**any **C**ore **I**nstructions
[6] Multi-Channel DRAM

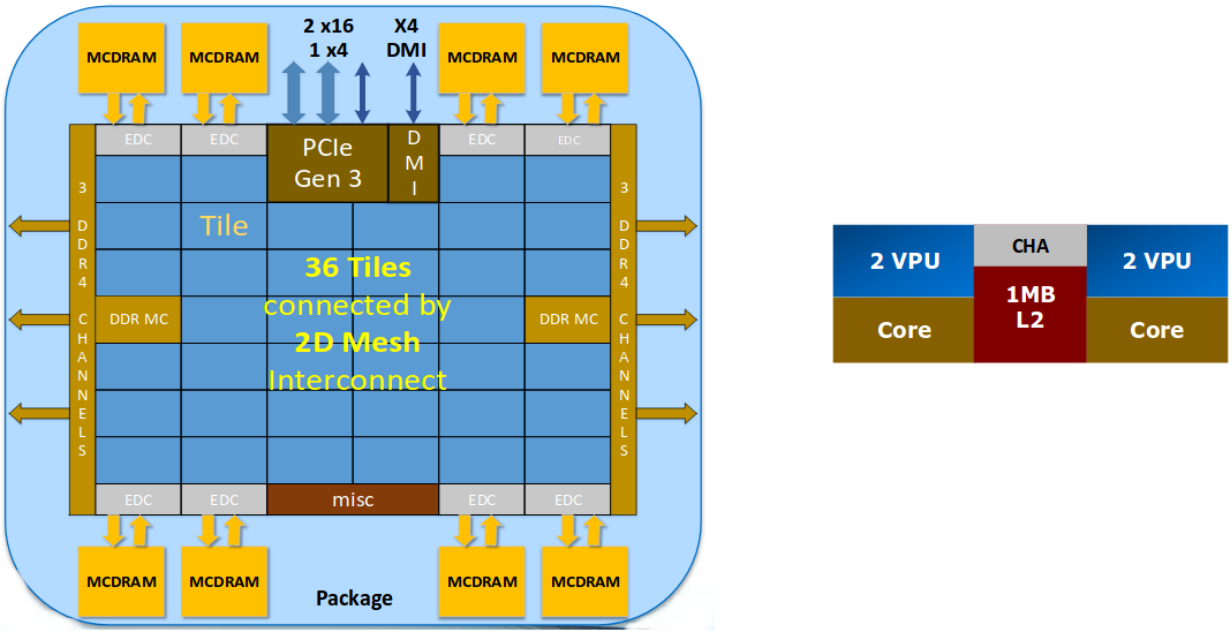


Figure 3.1.: The Intel Xeon Phi processor (left), one single tile (right), pictures from [5]

processors. Hence, for high-performance computing, it is especially important to use the AVX-512 vector instructions efficiently.

3.2. CoolMuc3

In September 2017, the Leibnitz-Supercomputing-Centre (LRZ) in Garching near Munich put a new Linux cluster named CoolMuc3 into operation. The cluster consists of 148 Intel Xeon Phi 7210-F processors. Each processor has 64 cores. Additionally to its 16 GB MCDRAM, each processor is suited with 96 GB DDR4RAM. The processors are connected via the Intel Omnipath technology, which offers a bandwidth of 25 GB/s. The total system reaches a theoretical peak performance of 394 TFLOPS [10].

3.3. Performance of Lammps on KNL processors

Lammps[7] is an open-source software project that performs MD simulations. It is programmed in C++ and maintained and distributed by Sandia National Laboratories, a US Department of Energy. The software is highly flexible and configurable concerning potential energy models. Due to its expandability, there exist many packages to optimise the simulation for different architectures, including the Knights Landing processor. Lammps can run on single-, multi- and many-core systems. It uses a domain decomposition method as described in chapter 2. The spatial domain is divided into subdomains and assigned to the available cores and processors, which communicate with each other via MPI[8], a message-exchange interface standard. A further internal parallelisation of each MPI process by shared memory threads is also possible due to the OpenMP standard. Lammps uses Verlet lists as described in Section 2.2. As molecular dynamics offer a certain kind of spatial as well as temporal locality, Lammps

[7] **L**arge-scale **A**tomic/**M**olecular **M**assively **P**arallel **S**imulator
[8] **M**essage **P**assing **I**nterface

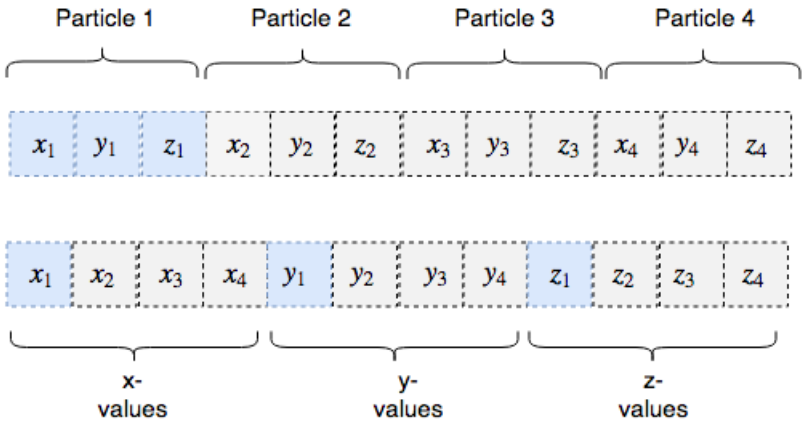


Figure 3.2.: AoS layout (top line) vs. SoA layout (bottom line)

tries to exploit that fact. For example, the Verlet lists of two particles that are close together will have many particles in common, so it is likely that a particle if it was accessed once, it will be reaccessed shortly after. As well, if a particle is accessed (because it is in a Verlet list of some particle), then particles that are close by that particle might also be accessed shortly after because they are likely to be in the list as well. To exploit this fact, at certain intervals, Lammps sorts and reorders the particles in memory, so that particles that are close to each other in the domain, are nearby in memory as well. The sorting is done by a bucket sort algorithm. The particles are binned to the individual cells of the domain. Further optimisations, which are specially intended for KNL processors are (based on [2], page 449 - 459)

1. 64 Byte Alignment:
   Data in Lammps is aligned to 64 Byte boundaries in memory which corresponds to the cache line size of the KNL processors. This shall prevent false sharing. False sharing occurs when data of two different threads belong to the same cache line. If the data of one thread changes frequently, the data of the other thread also has to be updated to ensure cache coherence, even if it does not change at all. This slows down the second thread.

2. AoS instead of SoA:
   Data layout i.e. the question in what way the data of the particles is stored in memory is very important. There are two main types of layout: Array of Structures (AoS) or Structure of Arrays (SoA). With an AoS layout, the particle data i.e. the x, y and z position plus additional the type of each particle will be stored consecutive in memory while with an SoA layout, the individual properties of the particles will be stored together. This is shown in Fig. 3.2 (for simplicity, just the x, y and z values of the particle's position are shown). Because in MD, if a particle is accessed, mostly more than one property is needed (for example to calculate the distance between two particles), the AoS layout is better suited and thus taken in Lammps. As mentioned, additional to the three position values, the type of the particle is also stored in the AoS layout. At first, this is, because the particle type is also needed for the potential calculation, secondly, it ensures that the complete data for one particle will be stored in just one cache line, and not distributed among several cache lines.

3. Vectorization:
   As already mentioned, with the KNL processors it is important to exploit the AVX-512 vector instruction set efficiently. Important is here to not rely on the compiler's skills to

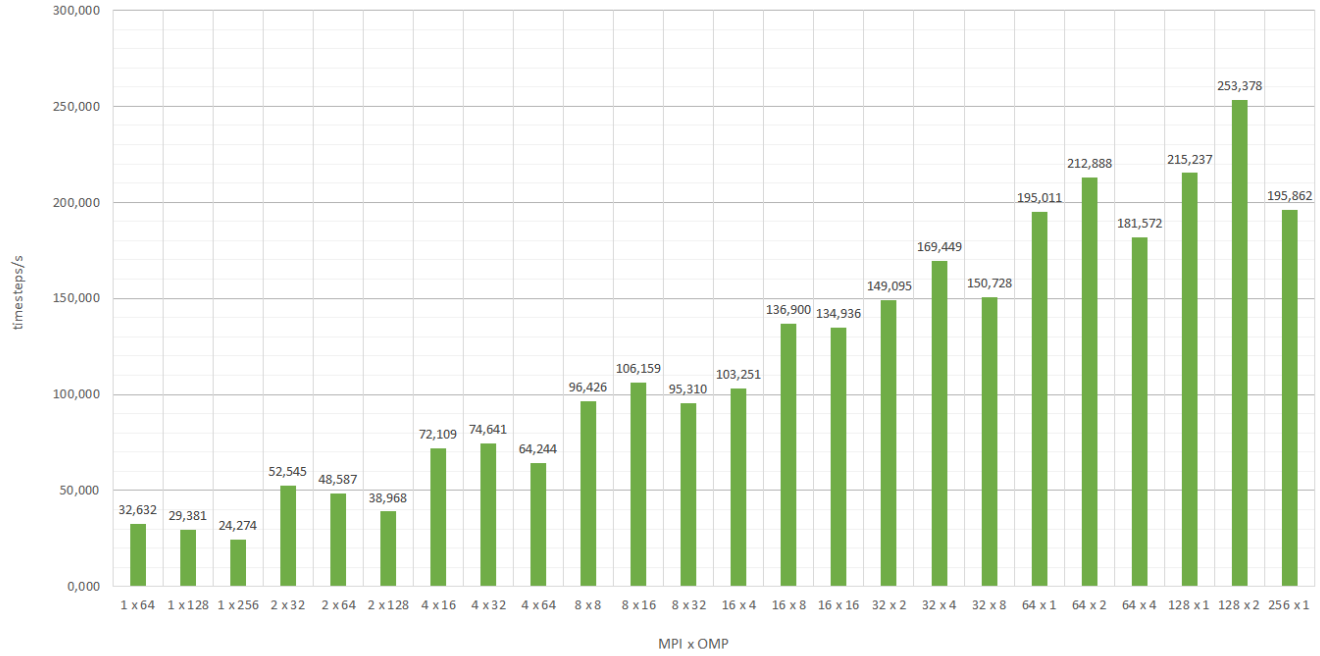


Figure 3.3.: Results of Lammps on one KNL node of CoolMuc3 with different combinations of MPI processes x OMP threads per MPI process

auto-vectorize loops, because MD offers several possibilities to vectorize, which even modern compilers are unable to detect. For example, in the Verlet list of a particle, no other particle will occur twice, so no data dependency is possible here, so these loops can be vectorized [2]. As well it is possible to inform the compiler that the data is aligned to 64 Byte boundaries, which can also lead to optimized code [2]. To explicit vectorize loops in C++, the available SIMD-PRAGMA directives can be used.

For this thesis, the performance of Lammps with several combinations of MPI processes and OMP threads per MPI process was tested. The test suite contained 512K particles with a cutoff radius of 2.5. As the intermolecular potential, the Lennard-Jones potential as described in chapter 2 was applied. The version of Lammps, optimised for KNL processors was used and the test was run on one single node of CoolMuc3. The combinations of MPI processes and OMP threads always exploit the full available 256 hardware threads. The results are shown in Fig. 3.3, the performance was measured in timesteps per second. One can see that the performance increases significantly with increasing amount of MPI processes (i.e. distributed memory hardware threads which work on different subdomains), which is exactly as expected. The amount of OMP threads, however, has a minor influence on the performance. Furthermore, to compare the performance of Lammps on KNL processors to other processors, additional tests were performed.

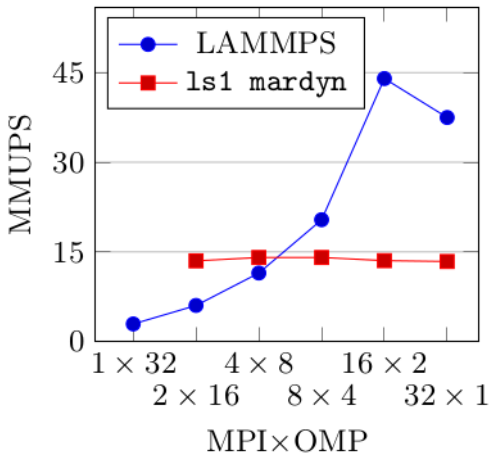


Figure 3.4.: Performance of Lammps on a Sandy-Bridge Intel Xeon processor, taken from [3]

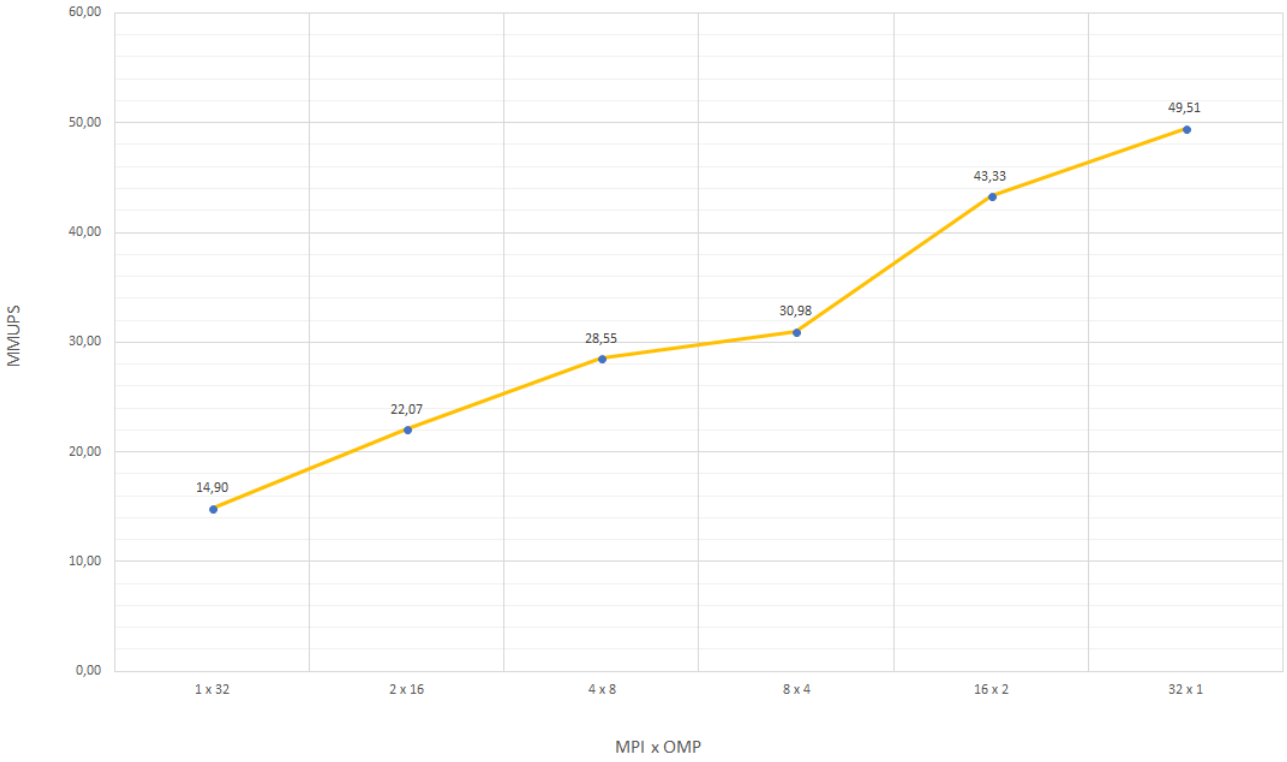


Figure 3.5.: 32 hardware threads used

In [3], a test is described, where Lammps was run on one node of the SuperMuc Phase 1 cluster of the LRZ in Garching, which means two 8-core Intel Sandy-Bridge-EP Xeon E5-2680 processors. The results are shown in Fig. 3.4. Again, the test suite contained 512K particles, and the Lennard-Jones potential was used with a cutoff of 2.5. The test was run with 32 hardware threads in different combinations of MPI processes and OMP threads. This time the performance was measured in MMUPS[9]. One can see, that the maximum performance of about 45 MMUPS is reached at a combination of 16 MPI processes and two OMP threads per MPI process. For this thesis, the same test was run twice on one node of CoolMuc3. At the first test, also 32 hardware threads were used in different combinations, the second test used the full available 256 hardware threads. The results of the first test are shown in Fig. 3.5. One can see, that the performance is about equal to the results on the Intel Xeon processors. However, a massive performance increase can be reached, if the full 256 hardware threads are used, as can be seen in Fig. 3.6. The best performance of 132.35 MMUPS was reached with 128 MPI processes and two OMP threads per process and is almost about three times as the maximum performance on the Intel Xeon processors.

[9] **M**illion **M**olecule **U**pdates **p**er **S**econd

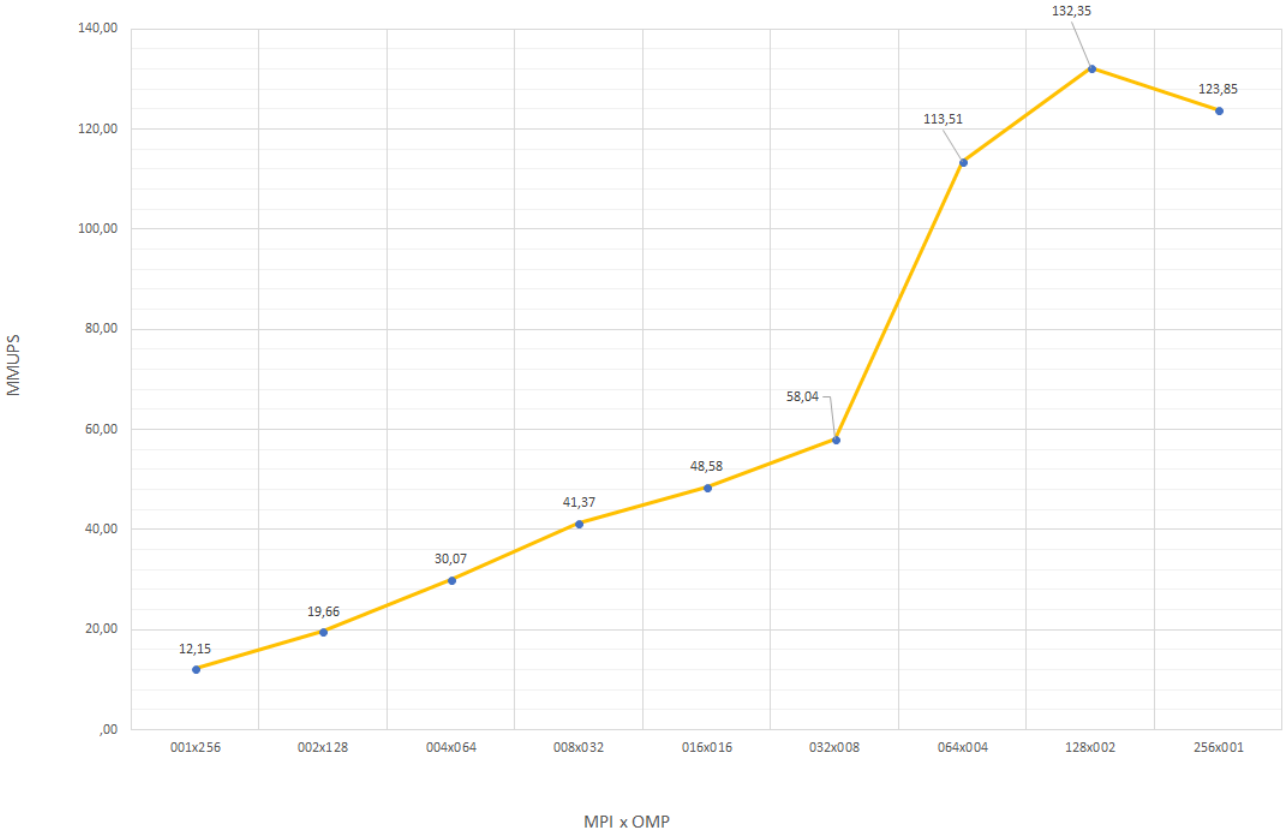


Figure 3.6.: 256 hardware threads used

4. Verlet Lists Implementation in AutoPas

AutoPas is a software, that is currently developed at the Technical University of Munich in context of the TalPas project and shall become an auto-tuned library for particle simulations with optimised node-level performance. It is written in C++, and it already offers a linked cells domain decomposition, which was used as the basis for this thesis and extended. In this chapter, the developed implementation of Verlet lists and its integration into AutoPas will be described. Therefore at first, the existing classes of AutoPas will be described, including the class hierarchy. Then, the newly developed classes will be introduced.
Annotation: The following UML class diagrams and explanations do not contain all classes of the AutoPas project. As well, in the classes, just the methods and attributes that are relevant for this thesis are shown and explained.

4.1. Existing Classes of AutoPas

The classes, in which the particles resp. molecules are implemented are shown in Fig. 4.1. Base class is the class `particle`. It stores three-dimensional arrays of type `array<double, 3>` for the position, velocity and the invoking force of the particle, and offers getter and setter methods for these values. Additionally, two methods, `addF` and `subF`, are provided to add/sub a force to/from the force array. Finally, every particle gets an ID of type `unsigned long` and the associated get/set methods. From this base class, the class `MoleculeLJ` is derived, which extends the `particle` class with two attributes of type `double`, `EPSILON` and `SIGMA`, which describe the Lennard-Jones potential as described in chapter 2. The `printableMolecule` class, which is derived from `MoleculeLJ`, extends this class with a `print` method to output the data of the particle on console.
The container classes are shown in Fig. 4.2. To store the particles and to represent one cell, the class `FullParticleCell` is used. It contains a vector of type `vector<TParticle>` to store the particles of the cell, where `TParticle` is a template parameter for the concrete type of particle that is used. The abstract class `ParticleContainer` works as a container for the cells and therefore represents the whole domain. The cells are stored in a vector of type `vector<TParticleCell>` with `TParticleCell` as a template parameter. It also contains values to store the dimensions of the domain (`_boxMin` and `_boxMax`) as well as a `double` value to store the cutoff radius. The abstract method `addParticle` has to be overwritten in subclasses. The class `LinkedCells` is the concrete type of container to implement the linked cells method and is thus derived from `ParticleContainer`. It uses an object of type `CellBlock3D`, which maps the one-dimensional vector of cells to the three-dimensional domain and offers associated methods, like returning the cell index that belongs to a spatial point within the domain. With the help of this object, it implements the methods `addParticle` from `ParticleContainer` and `iteratePairwiseAoS2`. The last method iterates once over all particle-particle pairs in the whole domain who are either in the same cell or neighbouring cells and thus enables the functionality to implement the linked cells method as described in section 2.1. As an argument, it takes an object of type `functor<TParticle>` (see Fig. 4.3). This functor has a method called `AoSFunctor` which takes two arguments of type `TParticle`. This method is applied to all of the particle-particle pairs during the iteration. In this way, for example, a functor of type `LJFunctor` (Fig. 4.3) can be used to calculate the forces for all

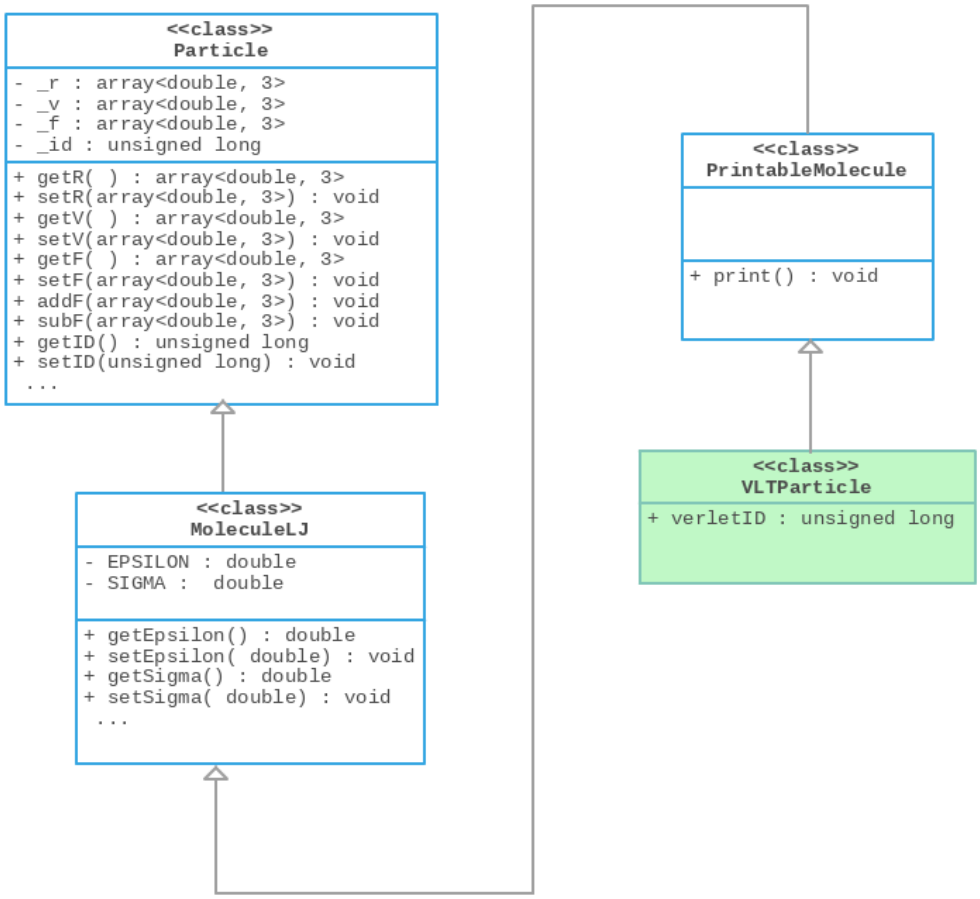


Figure 4.1.: Uml class diagram of the Particle classes

particles due to the Lennard-Jones potential.

4.2. Verlet Lists Implemenation

For the Verlet list implementation, at first, a new particle type had to be created. It was named `VLTParticle` and is derived from the `PrintableMolecule` class. It extends this class by an additional attribute of type `unsigned long`, called `verletID`. This attribute gives every particle a unique ID which is used for the build-up of the Verlet lists. The source code for the `VLTParticle` class is shown in Appendix A.1.
Furthermore, a new functor class called `VerletFunctor` was created, which is used to build the Verlet lists. It is derived from the `Functor` class and thus implements the method `AoSFunctor` to iterate over particle pairs. The complete source code of the class is shown in Appendix A.3

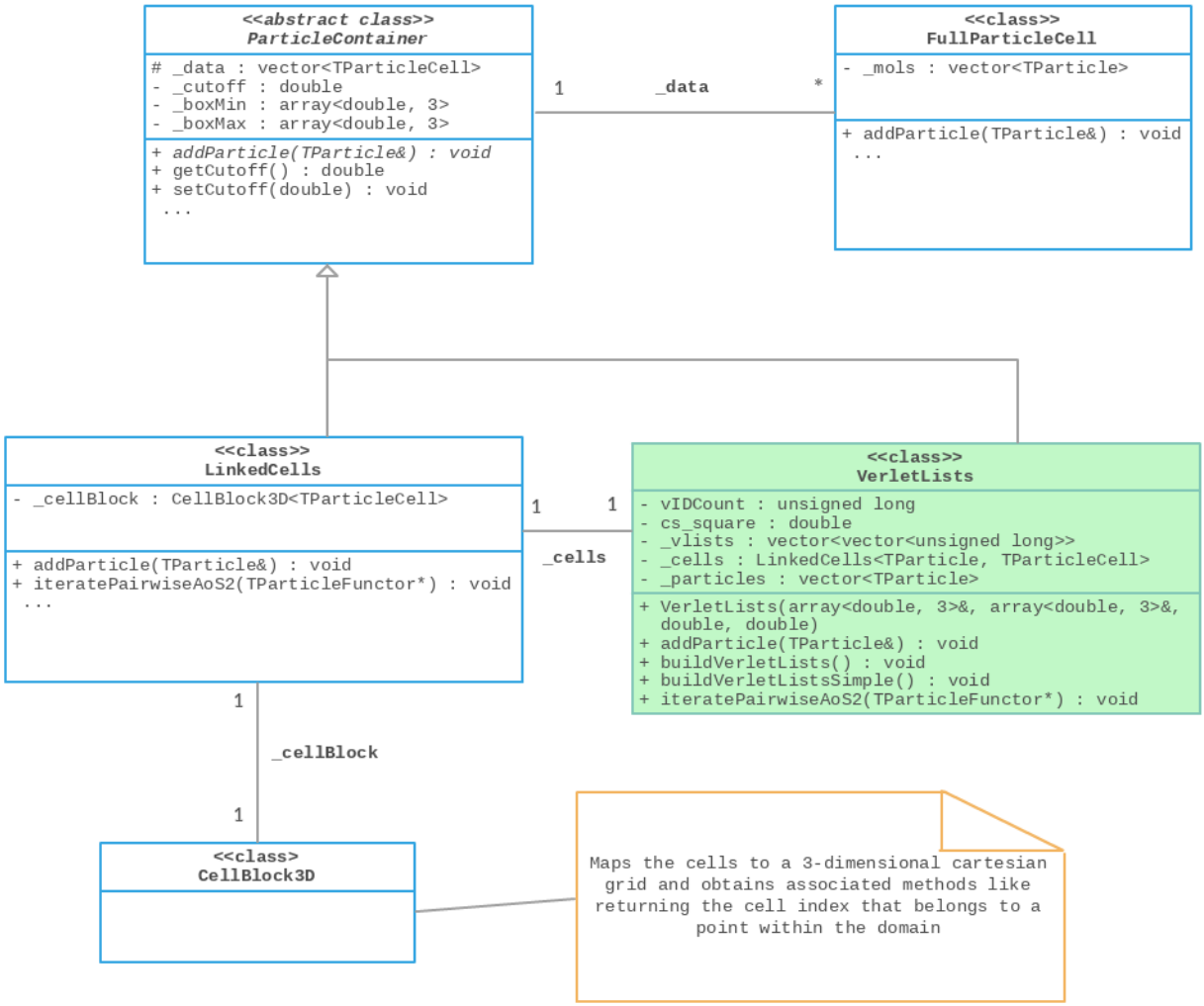


Figure 4.2.: Uml class diagram of the container classes

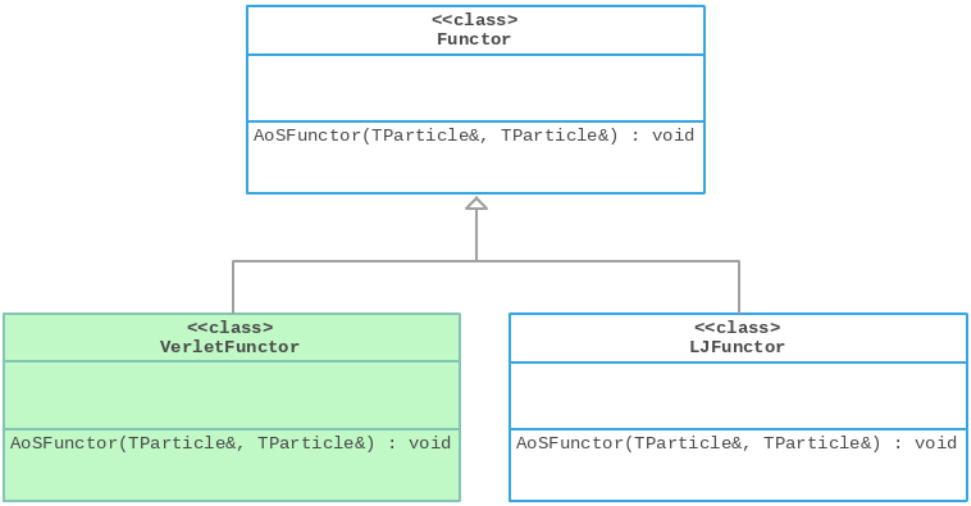


Figure 4.3.: Uml class diagram of the functor classes

The main object of the extension is the `VerletLists` class, where the mechanism of Verlet lists is implemented. It is directly derived from `ParticleContainer` and has the following additional attributes:

- `vIDCount ...`
  ... is of type `unsigned long`, stores the number of particles in the domain and is the value that the next added particle will get assigned as `verletID`.
- `cs_square ...`
  ... is of type `double` and stores the value $r_v^2 = (r_c + d_{skin})^2$
- `_vlists`
  This is the vector, that stores the Verlet lists. Each list is of type `vector<unsigned long>`, so `_vlists` is of type `vector<vector<unsigned long>>`. The particles are stored in the Verlet lists by their unique `verletID`.
- `_cells ...`
  ... is an object of type `LinkedCells<TParticle, TParticleCell>` and will be used to efficiently build the Verlet lists.
- `_particles ..`
  is a vector that stores all particles of the domain. It is therefore of type `vector<TParticle>`.

In the following, the implemented methods of `VerletLists` will be explained

4.2.1. Constructor

The constructor, inherited from `ParticleContainer` is extended with a fourth argument, which is a value of type `double` that denotes the thickness of the skin layer d_{skin}. The base class as well as the `LinkedCells` object `_cells` are initialized with a cutoff radius of $r_v = r_c + d_{skin}$, so that the cells in the `_cells` object have the size of the Verlet radius r_v. Also, the attribute `cs_square` is initialized with the value $(r_c + d_{skin})^2$.

```
template<class Particle, class ParticleCell>
VerletLists<Particle, ParticleCell>::VerletLists(const std::array<double, 3> &
    boxMin, const std::array<double, 3> &boxMax, double cutoff, double skin)
        : ParticleContainer<Particle, ParticleCell>(boxMin, boxMax, cutoff +
            skin), _cells(boxMin, boxMax, cutoff + skin) {

    cs_square = (cutoff + skin) * (cutoff + skin);
}
```

4.2.2. addParticle

This method is inherited from the base class `ParticleContainer`, its only parameter is a reference to a particle that shall be added to the domain.

```
    void addParticle(Particle &p) override {

        p.verletID = vIDcount++;
        _cells.addParticle(p);
        _particles.push_back(p);
        _vlists.push_back(std::vector<unsigned long>());
    }
```

At first, the particle gets its unique `VerletID`, then it is added to the `LinkedCells` object as well as to the global particle vector. At the end, a new empty Verlet list for this particle is added to the Verlet list vector.

4.2.3. buildVerletLists

This is the method, that builds resp. updates the Verlet lists.

```
    void buildVerletLists() {

        for(auto &vlist: _vlists)
            vlist.clear();

        VerletFunctor<Particle> vfunctor(&_vlists, cs_square);
        _cells.iteratePairwiseAoS2(&vfunctor);
    }
```

At first, all Verlet lists are cleared. Next, a functor object of type `VerletFunctor` is created and initialized. Then, the `LinkedCells` object `_cells` is used to iterate over all particle-particle pairs, that are either in the same cell or in neighboured cells and to apply the `VerletFunctor` to them. The code of the Functor method which is applied to each particle-particle pair of the iteration is shown below.

```
    inline void AoSFunctor(Particle &p1, Particle &p2) override {

        std::array<double, 3> dist = arrayMath::sub(p1.getR(), p2.getR());
        double distsquare = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] *
            dist[2];

        if (distsquare < cs_square)
            (*vlists)[p1.verletID].push_back(p2.verletID);
    }
```

First, the distance between the two particles p_1 and p_2 is calculated and squared. Then, it is checked, if this squared distance is less than $(r_c + d_{skin})^2$ i.e. if p_2 is within the Verlet radius r_v of p_1 (and vice versa). If it is, then the `VerletID` of p_2 is added to the Verlet list of p_1. Due to Newton's third law, it is not necessary to also add p_1 to the Verlet list of p_2.

4.2.4. buildVerletListsSimple

This is an alternative method to build the Verlet lists, but in a less efficient way without linked cells.

```
        void buildVerletListsSimple() {

            std::array<double, 3> dist;
            double distsq;

            for (unsigned long i = 0; i < _particles.size(); i++) {
                _vlists[_particles[i].verletID].clear();

                for (unsigned long j = i + 1; j < _particles.size(); j++) {

                    dist = arrayMath::sub(_particles[i].getR(), _particles[j].getR()
                        );
                    distsq = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] * dist
                        [2];

                    if (distsq < cs_square)
                        _vlists[_particles[i].verletID].push_back(_particles[j].
                            verletID);
                }
            }
        }
```

Within a double nested for loop it is iterated over completely all particle-particle pairs of the whole domain. Like in the method before, the distance between the pairs is computed and if the particles are within the Verlet radius of each other, the second particle is added to the Verlet list of the first particle.
Annotation: Because in both methods, the order in which the particle pairs are iterated differs, the individual Verlet lists that both methods create will be different. But overall, both created sets of Verlet lists will contain the same particle pairs.

4.2.5. iteratePairwiseAoS2

This method is similar to the equivalent method of the `LinkedCells` class.

```
        template <class ParticleFunctor> void iteratePairwiseAoS2(ParticleFunctor *f
            ) {

            Particle p1, p2;
            std::array<double, 3> dist;
            double distsq;

            for (unsigned long i = 0; i < _particles.size(); i++ )
            {
                p1 = _particles[i];

                for (auto j: _vlists[i])
                {
                    p2 = _particles[j];
                    dist = arrayMath::sub(p1.getR(), p2.getR());
                    distsq = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] * dist
                        [2];

                    if (distsq < cs_square)
                        f->AoSFunctor(p1,p2);
                }
            }
```

```
        }
```

It iterates over all particle-particle pairs of the Verlet lists, checks if their distance is less than the cutoff-radius r_c and, if so, applies the passed `Functor` object to the pair. The distance check is mandatory as the Verlet lists contain all particles whose distance is less than $r_c + d_{skin}$, so it has to be checked if the distance is indeed less than r_c.

5. Performance Tests

For the tests, the target build `md-main` of the AutoPas project was used and modified. First, it was modified to accept six command line arguments instead of three. Additional to the type of simulation (direct sum , linked cells or Verlet lists), the number of particles and the number of iterations to be done, also the cutoff radius r_c, the skin distance d_{skin} and the number of timesteps until to rebuild the Verlet lists has to be passed on command line. A method called `fillVLTContainerWithMolecules` was written, that fills an object of type `VerletLists` with particles of type `VLTParticle`. That was necessary because, in contrast to the already available method `fillContainerwithMolecules`, after the particles were added to the container, the Verlet lists have initially to be built. The following four tests were done:

1. Varying the number of particles
2. Varying the cutoff radius
3. Varying the skin distance
4. Varying the iterations between rebuilding the Verlet lists

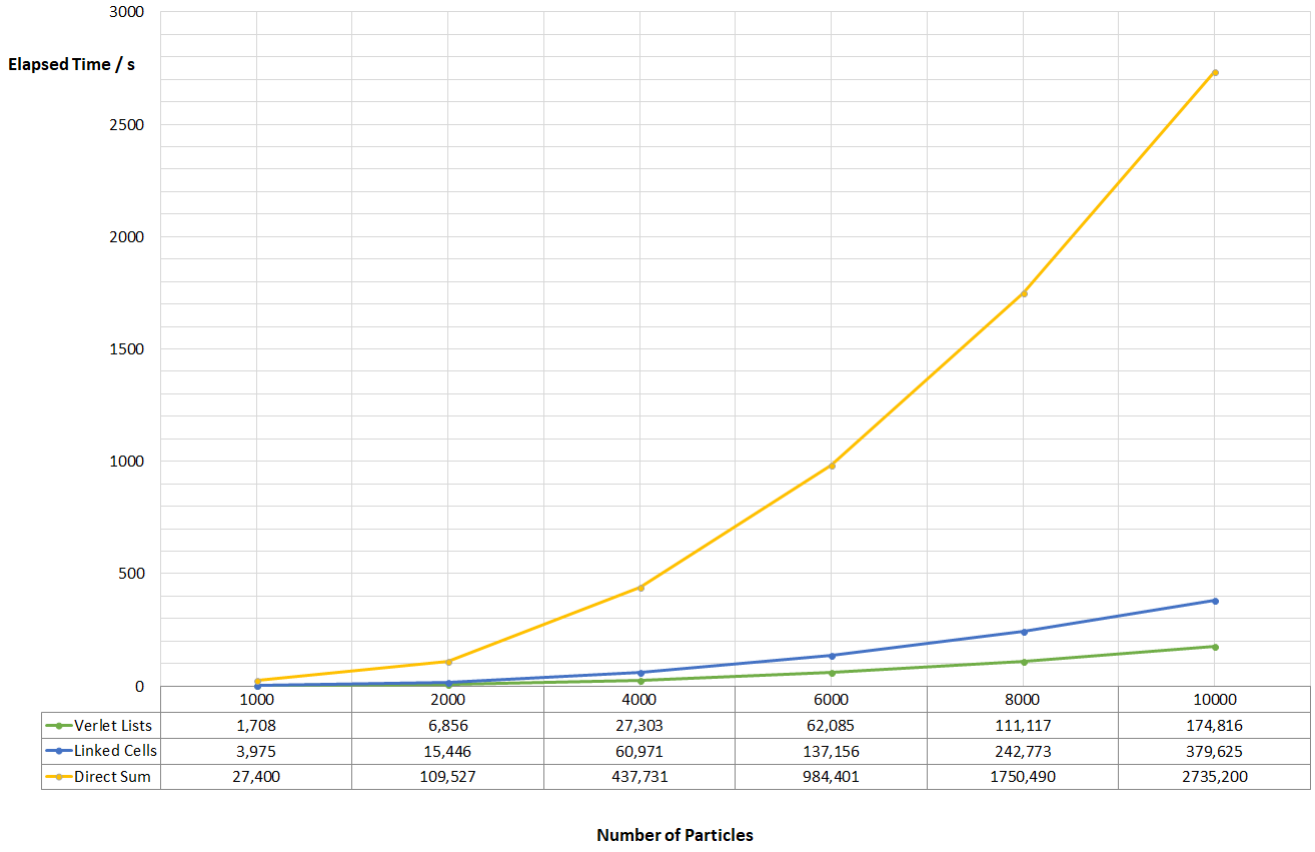
Additional to the Verlet lists simulations, the tests were also run with the linked cells method as well as with a direct sum method, which does a simple all-particle-particle-pair iteration. All tests were run in a spatial domain of size 5.0 x 5.0 x 5.0. The performance was measured by the elapsed time in seconds that the tests required for the simulation.

5.1. Number Of Particles

This test was run with a cutoff radius of 1.0 and a skin distance of 0.2, 50 timesteps were run. The rebuild interval, i.e. the number of timesteps until the Verlet lists are rebuild was set to 10. The number of particles starts at 1000 and increases up to 10000. The results for all three methods are shown in Fig. 5.1. The duration of the direct sum method increases significantly in a non-linear way with increasing amount of particles while the duration of the linked cells and the Verlet lists method increases distinctly less. The Verlet list approach performs significantly better than the linked cells method.

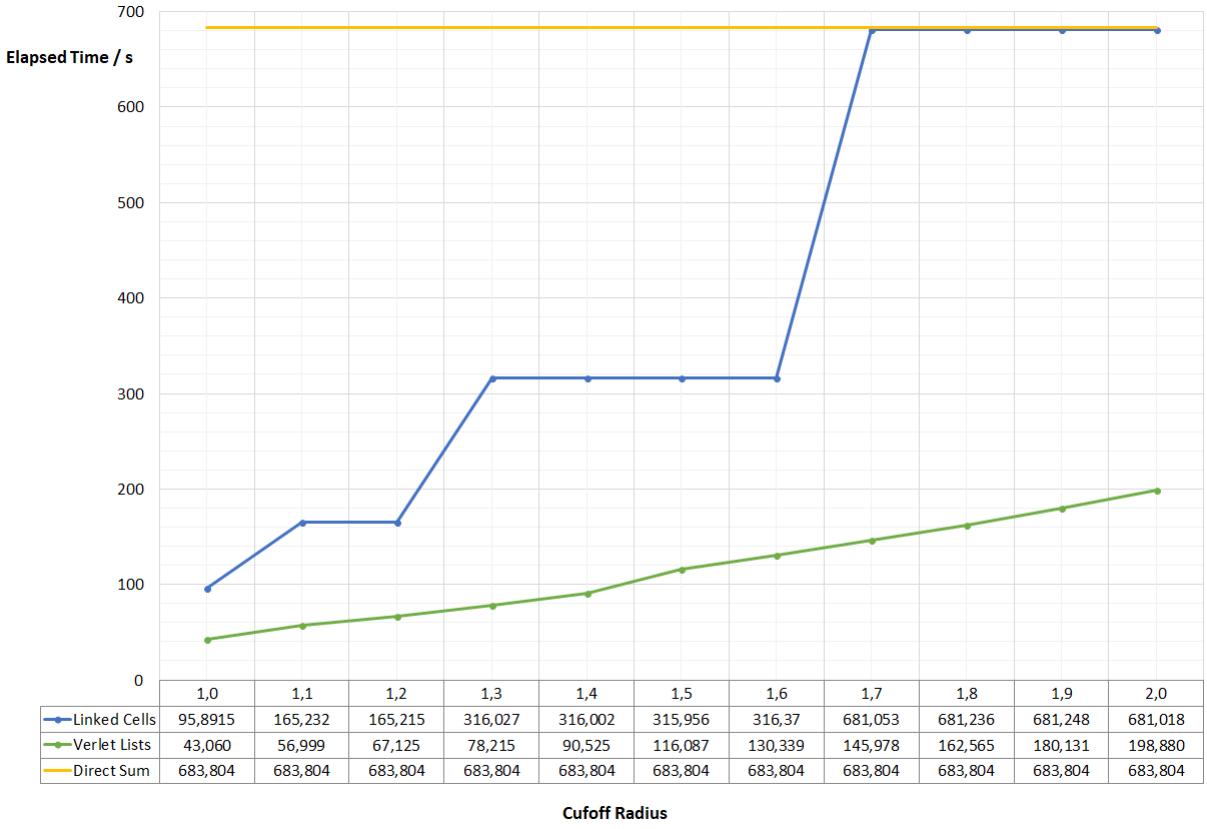
5.2. Cutoff Radius

This test was run with 5000 particles, a skin distance of 0.2 and a rebuild interval of 10. Again, 50 timesteps were run. The cutoff radius varies between 1.0 and 2.0. The results are shown in Fig. 5.2. Because the direct sum method is independent of the cutoff radius, it stays constant at an elapsed time of about 683 seconds. Both, the runtime of the linked cells and the Verlet lists method increase with increasing cutoff radius. The elapsed time of the linked cells method stays constant for several timesteps (for example between 1.1 and 1.2) and then performs a massive runtime increase, for example of about 100% between 1.2 and 1.3 until it reaches at a cutoff radius of 1.7 its maximum of about 681 seconds. From there, it performs similar to the direct sum method. The runtime of the Verlet list method instead increases in a more smooth way. Again, the Verlet list method performs significantly faster than the linked cells method.



	1000	2000	4000	6000	8000	10000
Verlet Lists	1,708	6,856	27,303	62,085	111,117	174,816
Linked Cells	3,975	15,446	60,971	137,156	242,773	379,625
Direct Sum	27,400	109,527	437,731	984,401	1750,490	2735,200

Figure 5.1.: Varying the Number of Particles



	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	2,0
Linked Cells	95,8915	165,232	165,215	316,027	316,002	315,956	316,37	681,053	681,236	681,248	681,018
Verlet Lists	43,060	56,999	67,125	78,215	90,525	116,087	130,339	145,978	162,565	180,131	198,880
Direct Sum	683,804	683,804	683,804	683,804	683,804	683,804	683,804	683,804	683,804	683,804	683,804

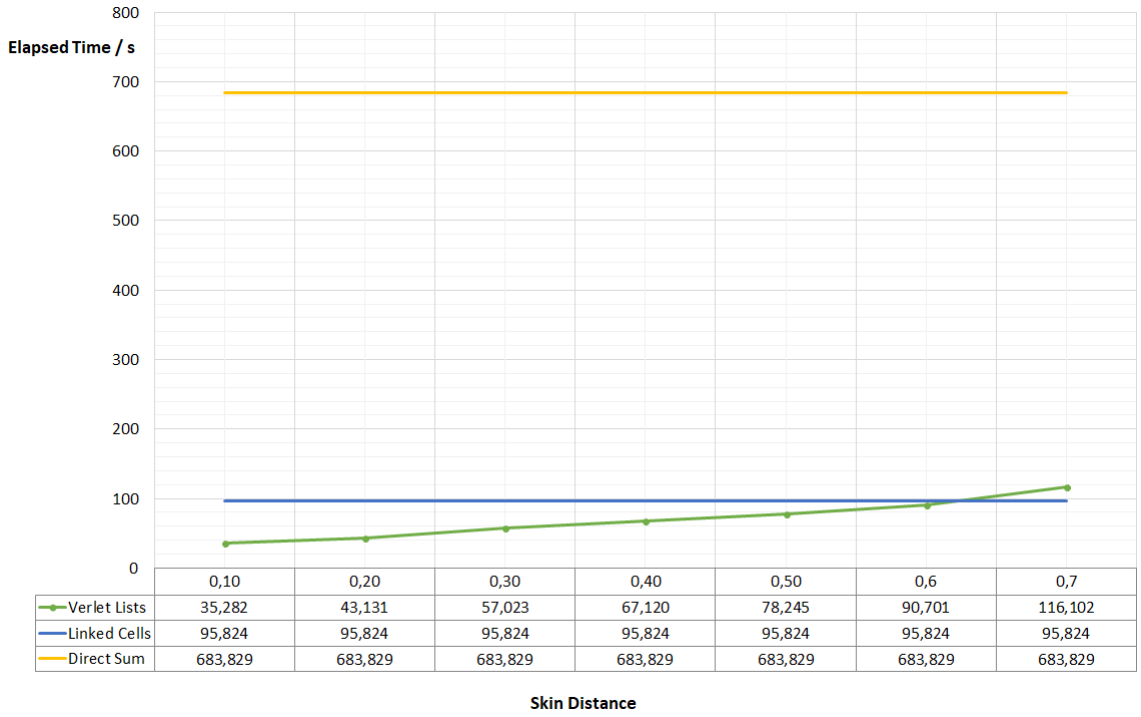
Figure 5.2.: Varying the Cutoff Radius

5.3. Skin Distance

For this test, the skin distance was varied between 0.1 and 0.7. Again, the number of particles was set to 5000, and 50 timesteps were run. The cutoff radius was set to 1.0 and the rebuild interval to 10. The results can be seen in Fig. 5.3. Because neither the direct sum nor the linked cells method depends on the skin distance, they stay constant at an elapsed time of about 683 resp. 95 seconds. With a skin distance of 0.1, the Verlet list method is about 2.7 times faster than the linked cells method. With increasing skin distance, the elapsed time approaches towards the linked cells method. At a skin distance of 0.6 both methods are about equal in runtime. For skin distances greater than 0.6 the Verlet list method becomes inefficient and the linked cells method performs better.

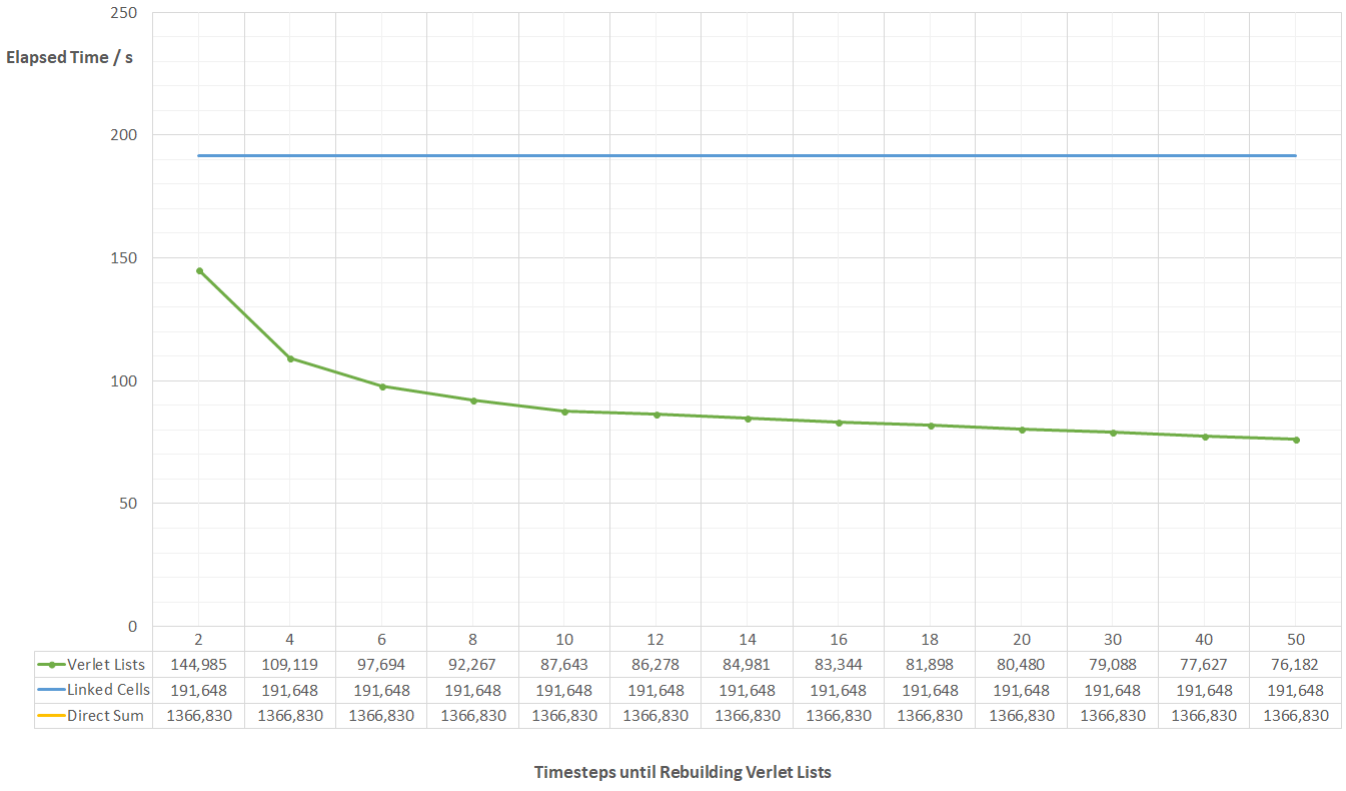
5.4. Rebuild Interval

In the last test, the rebuild interval was varied. This test was also run with 5000 particles, a cutoff radius of 1.0 and a skin distance of 0.2. This time, 100 timesteps were run. The rebuild interval was varied between 2 and 50. The results are shown in Fig. 5.4. Again, neither the direct sum nor the linked cells method depends on the rebuild interval, so they stay constant at an elapsed time of about 1366 resp. 191 seconds. With increasing interval, the number of rebuilds decreases and so the elapsed time in total decreases. But even if the Verlet lists are rebuilt every second timestep, the Verlet list method still performs about 25% better than the linked cells method.



	0,10	0,20	0,30	0,40	0,50	0,6	0,7
Verlet Lists	35,282	43,131	57,023	67,120	78,245	90,701	116,102
Linked Cells	95,824	95,824	95,824	95,824	95,824	95,824	95,824
Direct Sum	683,829	683,829	683,829	683,829	683,829	683,829	683,829

Figure 5.3.: Varying the Skin Distance



	2	4	6	8	10	12	14	16	18	20	30	40	50
Verlet Lists	144,985	109,119	97,694	92,267	87,643	86,278	84,981	83,344	81,898	80,480	79,088	77,627	76,182
Linked Cells	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648	191,648
Direct Sum	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830	1366,830

Figure 5.4.: Varying the Number of Timesteps between rebuilding the Verlet Lists

6. Conclusion and Outlook

In this thesis, either the performance of MD simulations on Intel Xeon Phi processors in contrast to other processors in general as well as the performance of the Verlet list approach in comparison to the linked cells method was investigated. It was shown, that the Xeon Phi processor can improve the performance of MD simulations significantly. Also, it was revealed, that the Verlet list approach has the potential to lead to a significant performance gain. Further studies that are possible are, for example, to investigate the performance on more than one processor, i.e. more than one node of CoolMuc3 as well as to examine and exploit the AVX-512 instructions in a more detailed way. Also, further studies could deal with the inter-process parallelisation with OpenMP threads on a shared memory level.

A. Source code of the implemented classes

A.1. class VLTParticle

```
class VLTParticle : public PrintableMolecule
{
public:
    VLTParticle();
    VLTParticle(std::array<double, 3> r, std::array<double, 3> v, unsigned long
        i);
    VLTParticle(double x, double y, double z, unsigned long id);

    unsigned long verletID;

};


VLTParticle::VLTParticle() : VLTParticle ({0.0,0.0,0.0}, {0.0,0.0,0.0}, 0){}

VLTParticle::VLTParticle(std::array<double, 3> r, std::array<double, 3> v,
    unsigned long i) : PrintableMolecule(r, v, i) {}

VLTParticle::VLTParticle(double x, double y, double z, unsigned long id) :
    PrintableMolecule({x,y,z},{0.0,0.0,0.0}, id) {}
```

A.2. class VerletLists

```
template<class Particle, class ParticleCell>
class VerletLists : public ParticleContainer<Particle, ParticleCell> {
public:
    VerletLists(const std::array<double, 3> &boxMin, const std::array<double, 3>
         &boxMax, double cutoff,
                double skin);

    void addParticle(Particle &p) override {

        p.verletID = vIDcount++;
        _cells.addParticle(p);
        _particles.push_back(p);
        _vlists.push_back(std::vector<unsigned long>());
    }

    void buildVerletListsSimple() {

        std::array<double, 3> dist;
        double distsq;

        for (unsigned long i = 0; i < _particles.size(); i++) {
            _vlists[_particles[i].verletID].clear();

            for (unsigned long j = i + 1; j < _particles.size(); j++) {

                dist = arrayMath::sub(_particles[i].getR(), _particles[j].getR()
                    );
                distsq = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] * dist
                    [2];

                if (distsq < cs_square)
                    _vlists[_particles[i].verletID].push_back(_particles[j].
                        verletID);
            }
        }
    }

    void buildVerletLists() {

        for(auto &vlist: _vlists)
            vlist.clear();

        VerletFunctor<Particle> vfunctor(&_vlists, cs_square);
        _cells.iteratePairwiseAoS2(&vfunctor);
    }

    void iteratePairwiseAoS(Functor<Particle> *f) override {}
    void iteratePairwiseSoA(Functor<Particle> *f) override {}

    template <class ParticleFunctor> void iteratePairwiseAoS2(ParticleFunctor *f
        ) {

        Particle p1, p2;
        std::array<double, 3> dist;
        double distsq;

        for (unsigned long i = 0; i < _particles.size(); i++ )
        {
            p1 = _particles[i];
```

```
                for (auto j: _vlists[i])
                {
                    p2 = _particles[j];
                    dist = arrayMath::sub(p1.getR(), p2.getR());
                    distsq = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] * dist
                        [2];

                    if (distsq < cs_square)
                        f->AoSFunctor(p1,p2);
                }
            }
        }

    private:
        LinkedCells <Particle, ParticleCell> _cells;
        std::vector<std::vector<unsigned long>> _vlists;
        std::vector<Particle> _particles;

        unsigned long vIDcount = 0;
        double cs_square;
    };

    template<class Particle, class ParticleCell>
    VerletLists<Particle, ParticleCell>::VerletLists(const std::array<double, 3> &
        boxMin, const std::array<double, 3> &boxMax, double cutoff, double skin)
            : ParticleContainer<Particle, ParticleCell>(boxMin, boxMax, cutoff +
                skin), _cells(boxMin, boxMax, cutoff + skin) {

        cs_square = (cutoff + skin) * (cutoff + skin);
    }
```

A.3. class VerletFunctor

```
template<class Particle>
class VerletFunctor : public Functor<Particle> {
public:

    VerletFunctor(std::vector<std::vector<unsigned long>> *lists, double
        cssquare)
    {
        cs_square = cssquare;
        vlists = lists;
    }

    inline void AoSFunctor(Particle &p1, Particle &p2) override {

        std::array<double, 3> dist = arrayMath::sub(p1.getR(), p2.getR());
        double distsquare = dist[0] * dist[0] + dist[1] * dist[1] + dist[2] *
            dist[2];

        if (distsquare < cs_square)
            (*vlists)[p1.verletID].push_back(p2.verletID);
    }

private:
    double cs_square;
    std::vector<std::vector<unsigned long>> *vlists;
};
```

Bibliography

[1] Michael Bader, Lecture 'Scientific Computing II', Technichal University of Munich, 2017

[2] Jim Jeffers, 'Intel Xeon Phi Processor High Performance Computing', 2016

[3] Nikola Tchipev, Steffen Seckler, 'TweTriS: Twenty Trillion-atom Simulation', 2018

[4] Steffen Seckler, 'Load Balancing for Molecular Dynamics Simulations on Heterogeneous Simulations', 2016

[5] J. Avinash Sodani, 'Knights-Landing (KNL): 2nd Generation Intel Xeon Phi Processor

[6] Joseph Kider, 'NVIDIA Fermi Architecture', University of Pennsylvania, 2011

[7] Michael Gerndt, Lecture 'Advanced Computer architecture', Technical University of Munich, 2017

[8] Anastasis Keliris, Michail Maniatakos, 'Investigating Large Integer Arithmetic on Intel Xeon Phi SIMD Extensions', 2014

[9] Stephen Lucas, Gerald Kotas, 'NVidia's GPU Microarchitectures'

[10] https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview/

Annotation: All graphics in this thesis, where no explicit reference is given, were created by myself either at the online platform 'www.draw.io', with the pyplot-library from the Python programing language or MS Excel. The UML-Diagrams were created with a free account at the online platform 'www.creately.com'.