SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Evaluation of Julia as a Suitable Language for Developing a Molecular Dynamics Simulator with AutoPas as a Backend

Simon Hang

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Evaluation of Julia as a Suitable Language for Developing a Molecular Dynamics Simulator with AutoPas as a Backend

Evaluierung von Julia als geeignete Sprache für die Entwicklung eines Molekulardynamik Simulators mit AutoPas als Backend

Author:	Simon Hang
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Samuel James Newcome, M.Sc.
Date:	17.04.2023

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

Munich, 17.04.2023 Simon Hang

Acknowledgements

First of all, I like to thank Sam for his great support and interesting discussions throughout the thesis. Thank you for making the topic possible which belongs to my favorite area of interest. Additionally, I like to thank Fabio for his helpful explanations.

Furthermore, I would like to extend my thanks to the chair of Scientific Computing and Professor Dr. Hans-Joachim Bungartz for the opportunity to research this fascinating topic.

Lastly, I gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre.

Abstract

In general, molecular dynamics simulations are computationally expensive tasks so that efficient algorithms are applied to accelerate the simulation as much as possible. Furthermore, high-performant programming languages are necessary to develop high-performant code. Typically, these languages are rather complex which leads to a decrease in productivity. Julia is a high-performant programming language and is also used to write high-level code. In this thesis, we use Julia to implement a simulator and use functions of the C++ library AutoPas as a backend. Additionally, we apply shared memory parallelization strategies to speed up the Julia simulator. After running performance tests and comparing the results to a C++ reference implementation, we discover that for small simulations and quick tests, the Julia simulator is a convenient choice to use because of its low initial compile time. Moreover, the Julia simulator is able to beat the C++ implementation in the force calculation in our chosen scenario.

Contents

Part I.

Introduction and Background

1. Introduction

In order to explain (bio-) chemical processes, we can choose the classical approach to execute experiments in a laboratory. However, some phenomena cannot be analyzed by experiments, which is why we can apply molecular dynamics (MD) simulations on a computer. In the research paper [21], the author uses molecular dynamics simulations to understand the origin of neurodegenerative diseases, because biophysical structural analysis of the molecules causing the diseases is unrealizable. Another practical case is the use of simulations in drug design. Discovering, how molecules can pass a membrane passively is important so that the molecule can get into a cell and act as expected. [13] As both examples show, the use of MD simulations have great importance for medical research in order to provide the best possible treatment for patients, however, the simulations are not limited to the medical field.

Not every researcher has deep programming knowledge or is a computer scientist so one goal is to provide an easy-to-use simulation framework. On the one hand, Python is a high-level programming language and it is easy to read and write code, which is suitable for non-expert programmers. On the other hand, we need to ensure that the code is performant, because MD simulations are associated with high computational effort. Historically, languages like C or C++ were used to write high-performance code and may be used for applications like MD simulations. The downside is, that languages like C++ are 'more complex' i.e., it is not easy to understand or write C++ code. [2]

With this thesis, we address this issue, by using the Julia programming language to implement a molecular dynamics simulator. Julia promises to provide the possibility to write high-performance code, as well as high-level code and may be the perfect choice to solve the mentioned problem. [2] Additionally, we use the C++ library AutoPas which provides much functionality necessary for efficient molecular dynamics simulations as well as node-level auto-tuning. [6]. Due to the fact, that the AutoPas library is written in C++, we need to create a wrapper, to make it possible to call its functionalities.

In the beginning, we introduce the theoretical background and terminology used in this thesis. Furthermore, we discuss the choice of the wrapper and present related research in this field. Chapter 3 discusses the implementation of the AutoPas wrapper, the simulator, and parallelization strategies. Moreover, we examine the usability of the chosen approach as well as present the results of the implementation in terms of performance in chapter 4. Finally, we look ahead to which additional aspects can be analyzed in future research works and point out the most relevant learning of this thesis.

2. Theoretical Background

This chapter contains the theoretical framework necessary to understand the subsequent implementation and results. Starting with a general introduction to molecular dynamics simulations, we continue with the explanation of parallelization strategies, the C++ library AutoPas, the Julia programming language, and the CxxWrapper. Finally, we present similar research results in the related work section.

2.1. Fundamentals of Molecular Dynamics Simulation

Molecular dynamic (MD) simulations calculate the movement of molecules for a defined period. It is necessary to discretize the time and divide the total simulation time into smaller time steps called simulation steps. If the position and the velocity of a particle[1] is known, it is possible to calculate the new position of the particle. However, the velocity also changes due to interactions with other particles. With Newtons second law 2.1

$$F = m \cdot a \tag{2.1}$$

the relation between the acceleration and the force is given and as the acceleration describes the change of the velocity over time, it is necessary to calculate the force between the particles. This leads to three necessary operations of every simulation: calculation of position, force, and velocity in every simulation step. [3]

In this thesis, we use the **Lennard-Jones potential**, as shown in formula 2.2, to calculate the interaction between particles and the negative gradient of the potential equals to the force.

$$U_{LJ}(r_{ij}) = 4\epsilon \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right) \tag{2.2}$$

The variable r_{ij} in formula 2.2 describes the distance between the two particles i and j, hence only describing pairwise interactions. To calculate the total force acting on a particle, it is necessary to sum all pairwise forces between all particles. If the number of particles is N, $\mathcal{O}(N)$ interactions need to be considered to calculate the total force for one particle. For all N particles, $\mathcal{O}(N^2)$ calculations are necessary so that the algorithm has quadratic complexity. [3]

Figure 2.1 shows the Lennard-Jones potential between two particles and as the distance increases, the force tends to zero. Therefore, it is possible to calculate the force only between particles within a so-called **cutoff radius**. If the distance between two particles exceeds this

[1]Depending on the scenario, we use particle as a placeholder which can be an atom or a small molecule.

cutoff radius, the force is so small that we can neglect it and assume it to be zero. Using the cutoff, results in a decrease in pairwise force calculations and ideally linear runtime. How to determine if a particle is inside the cutoff radius depends on the implementation, e.g. the **Linked Cells algorithm** assigns every particle to a cube in the global space and only the force between particles in adjacent cells is calculated. [3]

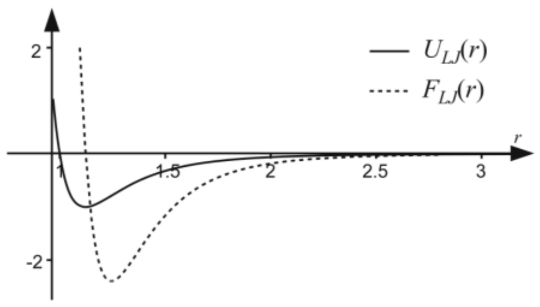

Figure 2.1.: Lennard-Jones Potential. With increasing distance r between two particles the force between two particles tends to zero.
Source: [3]

Up to this point, the necessary operations of a simulator were discussed. Usually, we restrict the particles to a pre-defined space, which is called **simulation domain**. Particles moving outside of this domain need to be processed differently than all the other particles, which is called applying a **boundary condition**. In this thesis, the outflow and periodic boundary condition are used.

The **outflow boundary** condition describes that all particles moving outside of the simulation domain are removed from the simulation. **Periodic boundary** condition means that a particle is reinserted into the simulation domain on the opposite side of its exit position. In figure 2.2 the black marked square is the simulated domain and all the squares around are also domains, but not simulated. Therefore, the periodic boundary condition mocks the behaviour that the simulated domain is part of a much bigger domain and represents only a smaller part with all parts having the same particle configuration. In this case, it is necessary to adjust the force calculation for the particles at the borders of the simulation domain. As indicated in figure 2.2, particle A is at the left border of the simulation domain and particle B is at the right border. Particle B from the left neighbor domain (which is not simulated), is theoretically within the cutoff of particle A, so it is necessary to adapt the calculation to get correct results.q [3]

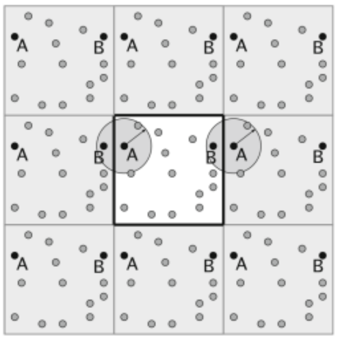

Figure 2.2.: Periodic Boundary Condition. Particles moving out of the simulation domain are reinserted on the opposite side.
Source: [3]

2.2. Speeding up MD Simulations through Parallelization Strategy

The goal of MD simulations is to simulate a big number of particles for many simulation steps (magnitude of 10^5) to receive good insights into the physical processes. Due to this computationally intensive task, it is necessary to apply parallelization strategies to reduce the computation time which are discussed in the following section. [3]

In the following, the term processor does not mean a CPU as a hardware component. We understand a processor as a placeholder for a computation unit, which can range from a thread of a CPU core to a node of a cluster.

2.2.1. Shared Memory Parallelization

Shared memory parallelization is a possibility to speed up the simulation and can be realized using OpenMP in C++. In this case, the calculation is distributed on several processors and all of them can access the same memory. This memory contains the necessary data for the calculation. The advantage of this approach is generally the easy adaption of the source code to be executed in parallel. However, one disadvantage is the occurrence of race conditions. This means that multiple processors simultaneously read from and write to the same memory address. Furthermore, the memory bandwidth may not be sufficient to supply all processors sufficiently. [7]

2.2.2. Distributed Memory Parallelization

Distributed memory parallelization describes the use of several processors with each processor having its own memory. To address the mentioned disadvantage of shared memory parallelization, every processor only gets the data needed for the current calculation. As every processor has only a share of the whole data, communication, and data transfer between processors may be necessary e.g., via a bus. This increases the complexity to adapt the source code for parallel execution. To use this parallelization for a MD simulation, the simulation domain is divided into smaller subdomains, which is called **domain decomposition**. [7]

Figure 2.3 shows one of the easiest domain decompositions where the whole simulation domain is divided along the x-axis and each subdomain S_i is assigned to a processor P_j. The respective processor P_j only knows the data of all the particles of its subdomain and calculates the position, force, and velocity of those particles.

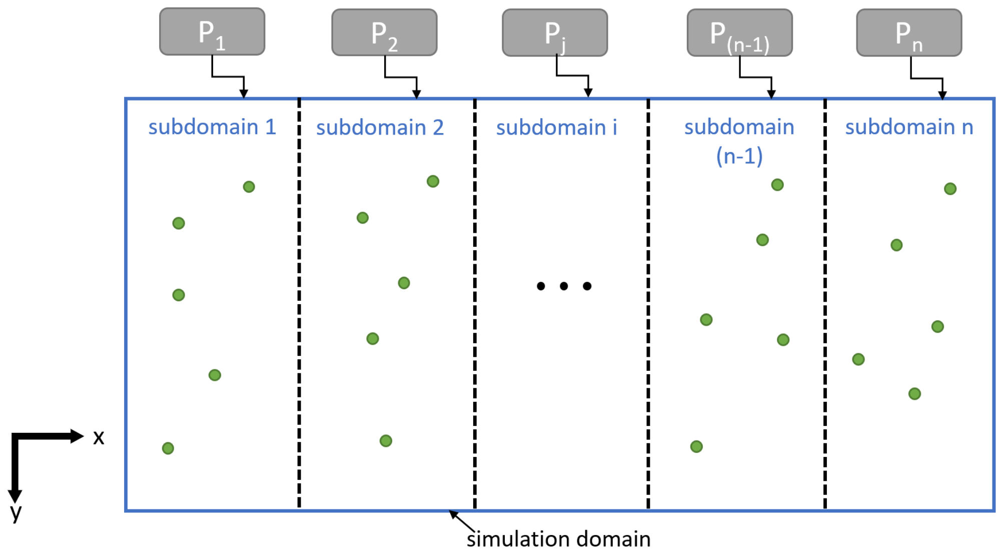

Figure 2.3.: Example of a domain decomposition. The simulation domain is divided into n subdomains in the x dimension.
Source: adapted from [3]

Up to this point, we only calculated the interactions between particles within one subdomain. However, particles may interact across subdomains and these interactions have to be considered in the force calculation as well. Figure 2.4 demonstrates the interactions between particles from different subdomains using the Linked Cells algorithm. Processor P_1 does the calculations for all particles inside its subdomain S_1 and processor P_2 performs the calculations for subdomain S_2. The green particle in the blue-marked cell of S_1 interacts with particles in the blue-marked cell of S_2 as the particles are in adjacent cells (from a global perspective). For this reason, P_2 needs to send the data for the mentioned particles to P_1, so that the force calculation for the green particle at the boundary of S_1 is correct. Additionally, particles can move from S_2 to S_1, as indicated by the black particles. It is necessary that P_2 sends the data for this particle to P_1 and deletes the data from its memory. Of course, P_1 has to perform the same steps as P_2. Handling boundary conditions may lead to additional changes in the implementation.

The communication between processors can be realized by sending and receiving messages. The message passing interface MPI defines methods, such as MPI_send(), to send a message from one processor to another one. There are multiple implementations for this interface like MPICH. [7]

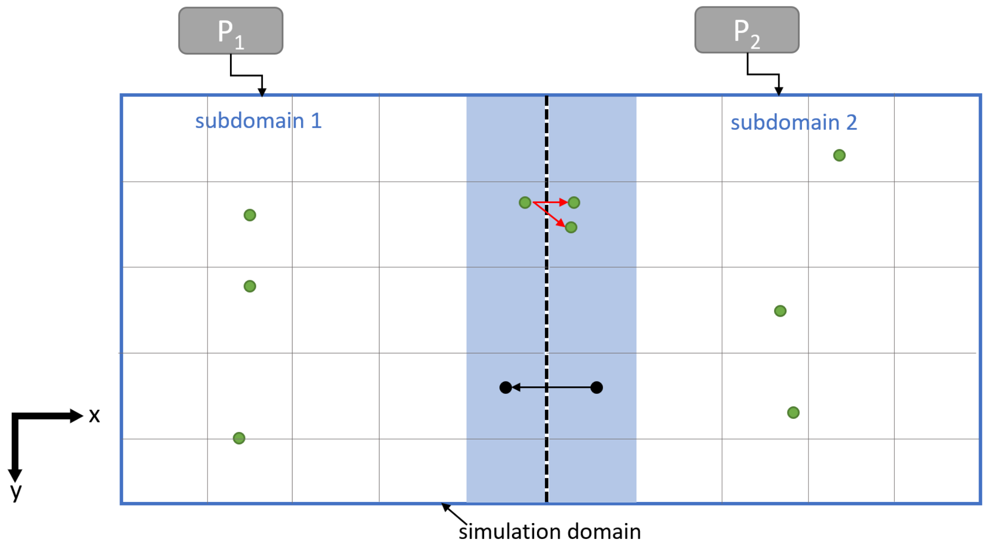

Figure 2.4.: Example of a simulation domain, which is divided into two subdomains. Particles from both subdomains interact with each other or move from one subdomain into the other one.
Source: own illustration, adapted from [3]

As described above, there are two ways of parallelization. It is also possible to combine both strategies to make the program even more performant e.g., by dividing the simulation domain into subdomains, and one subdomain is assigned to a node of a cluster and every node assigns tasks to the threads of a CPU. After the computation is executed, necessary data is exchanged.

2.3. Introduction to AutoPas

As we have already shown in the previous sections, an efficient simulation is important to picture the reality. In this thesis, we use the C++ library AutoPas which provides functionality for efficient MD simulations. We motivate the usage of AutoPas below.

Particles of a simulation need to be stored in a data structure. AutoPas provides an abstraction for this data structure called **container** and a function to iterate over all the particles stored in the container. The chosen data structure depends on the selected algorithm for the force calculation. For example, in the case of the Linked Cells algorithm, the container is a vector of cells and each cell stores the particles inside of its own vector. The order of iterating over the particles depends on the selected **traversal strategy**. With this traversal, we try to achieve an efficient iteration over all particles and avoid race conditions if parallelization strategies are applied.[2] [6]

The interface of AutoPas is designed as a façade-pattern, so users do not need to understand the implementation details and are still able to use AutoPas efficiently. Moreover, we like to point out the special feature of AutoPas. Every simulation scenario needs a different

[2]Further configuration options and advantages can be found in [6]

set of parameters, like the data structure or the traversal, to be simulated most efficiently. Autopas can select the best configuration automatically, in the beginning, and during runtime, so users do not need to have deep algorithmic knowledge and do not need to select the best configuration themselves. This tuning happens on the node level. As the optimal configuration may change over the time of the simulation, AutoPas can adapt the parameters to be optimal again. Some parts of the simulation domain may be empty, but using the Linked Cell algorithm, these parts are still checked if particles are inside the cells. [6]

2.4. The Julia Programming Language

Algorithms, data structures, and parallelization strategies have an influence on the performance of a molecular dynamics simulation. However, the used programming language is important as well and can increase the speed of the simulation. As already mentioned in the introduction, we work with the Julia programming language to implement the molecular dynamics simulator and motivate the utilization of Julia in the following.

In [1], Bezanson et al. state that Julia provides the advantage of statically typed languages like C, which are known for high performance, as well as of dynamically typed languages like Python, which are known to be used to write high-level code. Because of this promising statement, we like to point out the advantages of the Julia programming language in detail.

The reason, for the high performance of Julia, is based on three major reasons. The first one is that Julia uses **just in time (JIT) compilation**, which means that parts of the source code are compiled during runtime. [18] The second reason is the usage of **multiple dynamic dispatch**. If a function is defined multiple times with the same name but different argument types, multiple dynamic dispatch chooses the correct function definition based on all arguments during run-time. Even if the compiler does not know the type of single variables before run-time, information about the types is automatically provided during run-time e.g., if a function is called with a set of real values. These values are of a specific type, hence providing type information. The last reason is that the compiler uses the deduced types to create specialized code. For the code specialization, as well as for the correct function selection, Julia has a specific rule book and strategy, which is discussed in detail in [1].

- type annotation is optional and not necessary for performance [2]
- user-defined types are as performant as built-in types [2]
- no need for a build system [1]
- built-in and easy-to-use parallelization techniques, on instruction level with simd as well as shared and distributed memory parallelization [2]
- the Julia REPL (read, eval, print loop) provides great interactivity, as expressions can be evaluated one after another. This can also make MD simulations more interactive, as the simulation state can be inspected after some time and further action can be decided spontaneously. [8]

2.5. Wrapping C++ Code

We already learned about AutoPas and that it is written in C++. Furthermore, we explained the advantage of Julia as a fast programming language. In order to call functions or types from AutoPas, we need to wrap the C++ code with a wrapper. This wrapping is done with the CxxWrap.jl package.

2.5.1. Overview and Discussion of Available Wrappers

Multiple packages and strategies exist, to use Julia and C++ together. In this subsection we present different wrapper options, explain our decision to use CxxWrap, and why we do not use other wrapper options.

The Julia C Interface[3] is the foundation to call C or C++ functions from Julia. As a programmer, we can use `ccall` to call a C/C++ function, if we know the symbol name or have a pointer to the function. The `ccall` method from Julia is the basis for all following wrappers. In theory, it is possible to directly write the wrapper with the help of `ccall`, but the solution may not be the most elegant one.

The first Julia wrapper option is jluna [5]. According to the documentation, the package is used in cases where C++ is the mainly used language and not Julia. For the thesis, we use Julia as the main language and call only some functionality from C++.

Another option is the Cxx.jl package, which may be used to wrap C++ code and Julia is assumed to be the host language. Cxx.jl is interactive, as it provides the possibility to execute C++ code like Julia code in a REPL-style way [14], which may be a very comfortable feature. However, the project seems to be unmaintained and can only be used with Julia versions up to 1.3.[4] So using this strategy in the long term might not be the best choice. Additionally, the author of the bachelor thesis [16] states that a small test using the wrappers Cxx.jl and CxxWrap.jl resulted in a ~ 5 times longer runtime of the code wrapped with Cxx.jl.

CxxInterface.jl[5] is another way to wrap C++ functions. The wrapping code is written in Julia and with automatic string manipulation, C++ code is created, which needs to be compiled. However, at first sight, we need to provide plenty of information about the function e.g., argument types of the functions, and if many functions are wrapped, this may produce plenty of code.

The only left option is CxxWrap.jl[6]. An interesting feature of CxxWrap is that we generally only need one line of C++ code to expose e.g., a function, to Julia. Furthermore, it is still maintained, the GitHub repository provides many examples and in some cases, it is even possible to generate the wrapper code automatically with WrapIt.jl[7]. If this package

[3]https://docs.julialang.org/en/v1/base/c/#C-Interface
[4]https://github.com/JuliaInterop/Cxx.jl
[5]https://github.com/eschnett/CxxInterface.jl
[6]https://github.com/JuliaInterop/CxxWrap.jl
[7]https://github.com/grasph/wrapit

is used, the wrapper code is written in C++ as well as compiled and linked against a shared library. All in all, this seemed to be the most elegant and straightforward solution at the beginning of the thesis. [9]

2.5.2. General Usage of CxxWrap.jl

The goal of CxxWrap.jl, also called CxxWrap, is to wrap an already compiled library, write the wrapper code in C++ and link this code against the library. In the following, we explain how the type `A` from listing 2.1 can be wrapped using CxxWrap.

```cpp
// type A
struct A {
    int x;
    double y;

    // constructor
    Foo(int _x, double _y);

    // member function
    int add(int _x, double _y) {
        x += _x;
        y += _y;
    }
};
o
```

Listing 2.1: Example type to illustrate wrapping with CxxWrap.jl.

First of all, we need to write the wrapping code into the function `define_julia_module`, which takes a `jlcxx::Module` as an argument (see listing 2.2). The `Module` type represents a Julia module in C++ and is used to add types and functions for the usage in Julia. In the code snippet, we assume that the type `A` is in the default namespace, in other cases the fully qualified name needs to be specified, whenever `A` is used. With the command `add_type` we can add a new type to the module and this returns an object we called `aType`. To this `aType` object we can add constructors with `aType.constructor<arg1, arg2, ...>()` or functions with `aType.method()`. The names in quotation marks are the names exposed to Julia and can be chosen freely e.g., the C++ function `add` can be called with `addCpp` in Julia. As we can see, the syntax for the constructor is special, as all used argument types need to be explicitly written in `< >` brackets, whereas we only need to specify the C++ function name, but not the argument types if a function is added (assuming the function is not overloaded. More on how to wrap overloaded functions in section 3.1). All files containing wrapper code need to be compiled and linked against the library. [10]

```cpp
JLCXX_MODULE define_julia_module(jlcxx::Module& mod) {
    // add type A to the C++ module
    auto aType = mod.add_type<A>("ACpp");

    // add the constructor with the arguments of type int and double to aType
    aType.constructor<int, double>();

    // add the member function add, which is called addCpp in Julia
    aType.method("addCpp", &Foo::add);
}
```

Listing 2.2: C++ code for wrapping of type A and its functions. adapted from: [10]

The last necessary step is to create the module in Julia, as listing 2.3 shows. The macro `@wrapmodule` takes the path to the shared library as an argument and constructs a static table of all wrapped functions (for pre-compilation) and the macro `@initcxx` fills this table at run-time. [10]

```
module FooWrapper
    using CxxWrap
    @wrapmodule(joinpath("path/to/lib", "shared_lib_name.so"))

    function __init__()
        @initcxx
    end
end
```

Listing 2.3: Exposing the wrapped functions to Julia. adapted from: [10]

2.5.3. Short Introduction into CxxWraps Internals

This section provides a short introduction how CxxWrap is implemented. The following code is written on the C++ side.

With `ccall` it is possible to call a C function from Julia. It is necessary to provide the name of the function in C and the library, where the function is defined. Furthermore, the types of the function arguments, the type of the return value, and the real values of the arguments need to be specified. CxxWrap needs two steps to call a C++ function from Julia. At first, we need to define a function on the C++ side, that returns a pointer to the function that is invoked from Julia. This is necessary as the names of symbols in C++ libraries differ depending on the used operating system (name mangling of C++ compiler), and hence symbol names cannot be hard-coded. The second step is to use `ccall` to invoke the desired function as discussed, with the exception that we use the pointer to the function instead of the symbol name. [10]

The basic idea of CxxWrap is explained in the paragraph above, however, we also need to handle type conversions as the Julia type of a variable may not be mapped to the correct C++ type on all operating systems. For example, the C++ type `long long` is mapped to the Julia type `CxxLongLong` on Linux (64-bit). On MacOS, the type is mapped to the Julia type `Int64`, as different operating systems use different sizes for some integer values. This means that after calling a wrapped function from Julia with `ccall`, we have to convert the 'Julia types' to the corresponding 'C++ types' on the C++ side and of course, this applies the other way around for the return type of a function. [10]

Additionally, we need to handle the call of member functions of a C++ class. CxxWrap uses the `std::function` class from C++, as shown in listing 2.4 for this scenario. The `std::function` takes as arguments the original function arguments plus a reference to the object of the class of the member function, in this example of type `A`. Inside the function, we use the referenced object to call the proper function with the correct arguments. [10]

```cpp
std::function<void (A&, int, double)> addWrapper([] (A& a, int _x, double _y)
{
    return a.add(_x, _y);
});
```

Listing 2.4: CxxWrap internal: wrapping of a member function with `std::function` adapted from: [10].

Other special cases like error handling or using lambdas are important to mention but not discussed here. [10]

2.6. Related Work

For the last part of this chapter, we like to present related research results. We structure the text in two parts, the first one is about Julia's interoperability and the second one is about using Julia for high-performance code.

2.6.1. Julia Interoperability in Current Research

The CxxWrap.jl package was used to wrap two C++ libraries FastJet and LCIO to use this functionality from Julia for high-energy physics analyses. This paper discusses the usability of the wrapper, as well as the runtime of the Julia implementation using wrapped functions against the pure C++ implementation and a Python implementation. The researchers state that the wrapping of C++ libraries can be easily accomplished with CxxWrap. The overhead of the wrapping is available, especially if the problem size is small, however, for bigger and longer running simulations the overhead is very low. This leads to the result that the Julia code even outperforms the C++ code or is equally fast for big input sizes. [20]

In two bachelor theses, CxxWrap.jl is also used to wrap C++ libraries. In one thesis, the preCICE library, which is used for multi-physics simulations [12] is wrapped with CxxWrap, as well as the normal C interface is used. The author does prefer the C interface over the CxxWrap package because one of the multiple reasons is that the interaction with the C interface is more similar to the Julia not object-oriented design. The second thesis wraps the Omnet++ library, which is used to simulate networks and distributed systems [16]. Furthermore, the overhead of the wrapper and of the Julia implementation is evaluated. In conclusion, the overhead is quite high, but again can be neglected for longer simulations. The author states that the Julia implementation cannot compete with the C++ implementation in terms of performance.

2.6.2. The Julia Programming Language in Scientific Computing

In [19] Julia is used to running quantum computational simulations for chemistry. Further, they apply parallelization strategies in the Julia code and show a decent scaling of

92% for one of their algorithms using more than three hundred thousand processes. Low communication between processors makes the scaling of the algorithm very efficient. With the developed code, it is possible to simulate protein-ligand interactions. Especially, the program was applied to study how ligands bound to the protease of the SARS-CoV-2 virus.

Another topic related to using Julia for science, again theoretical chemistry, is discussed in [17]. The researchers implemented a quantum chemistry package called JuliaChem.jl and compared the program (in terms of efficiency) to the existing package GAMESS. The results show that the Julia implementation is as performant as the GAMESS implementation and depending on the simulated scenario and the used GAMESS class, the Julia implementation was 20.8% slower in the worst case and 42.1% faster in the best case. Additionally, the overhead of the JIT compilation is discussed.

One result which is closer to AutoPas is Molly.jl, which is a molecular dynamics package[8] completely implemented in Julia. Similarly to AutoPas, we can use the Lennard-Jones potential or periodic boundary conditions for the simulation domain. Additionally, some neighbor list implementations are used to accelerate the simulations. Further Julia packages related to MD simulations can be found in the reference below.[9]

The CellListMap.jl package, presented in [15], is used to implement cell lists algorithms, which are used for particle simulations. With this Julia package, it is possible to create custom and efficient cell list-based algorithms with ease. The researcher concludes that the implementation of CellListMap.jl is efficient and can be compared to existing performant packages for computing neighboring particles. This package is also included in the Molly.jl package.

[8]`https://juliamolsim.github.io/Molly.jl/stable/`
[9]`https://juliamolsim.github.io/`

Part II.

Implementation, Discussion, and Results

3. Implementation

This chapter discusses the implementation of the molecular dynamics simulator. Firstly, we explain how the wrapper for the AutoPas library is implemented. Secondly, the fundamental components of the simulator are explained. The last two sections are about changes to apply shared and distributed memory parallelization.

3.1. Implementaiton of the Wrapper

One target of this thesis is to be able to call functions of the AutoPas library, written in C++, from the Julia programming language. We already discussed the package CxxWrap.jl and in this section, we like to explain in detail, how the AutoPas wrapper is implemented, and which workarounds and components are necessary to use AutoPas fully functional.

3.1.1. Exposing the AutoPas Class to Julia

In this subsection, we use the term module or C++ module as a synonym for the C++ class jlcxx::Module, representing a Julia module in C++. Listing 3.1 shows an extract of the AutoPas class and we notice that it is a parametrized class, expecting a particle as a parameter type. We already know how a type is added to the C++ module from section 2.5, but template types are different, as we can see in listing 3.2. First of all, we need to specify all combinations of template types we may use in Julia, which in this case are `AutoPas<MoleculeJ<double>>` and `AutoPas<MoleculeJ<float>>`.[1] Additionally, we create a struct called `WrapAutoPas`, which implements the `operator()` function and is basically a functor. Furthermore, we add all functions to the module inside this new struct, similar to adding a function to a 'normal' module. Lastly, the functor adds the functions for all template types to Julia automatically. The `init` function can be added to the module without any further effort and with the already-known syntax. Adding an overloaded function to the module is possible as well, however, the function pointer needs to be casted to the correct argument types, as shown with the function `deleteParticle`. In this case, the function takes a reference to a particle as an argument and returns nothing. Similarly, the function begin is overloaded and we cast the pointer to the correct types. We see, that the function has an Option class as an argument and as return type `iterator_t`, both types are unknown for now and are discussed later.

```
template <class Particle>
class AutoPas {
  public:

    void init();
```

[1]The type `MoleculeJ` is discussed in the subsequent subsection.

```

    void deleteParticle(Particle& particle);

    iterator_t begin(IteratorBehavior behavior);

    void setBoxMin(const std::array<double,3> &boxMin);

    template <class Functor>
    bool iteratePairwise(Functor *f);

};
```

Listing 3.1: Extract of the AutoPas class. adapted from [6]

```

struct WrapAutoPas {
    template<typename T>
    void operator()(T&& autoPas) {

        using AutoPasType = typename T::type;
        using iterator_t = typename autopas::IteratorTraits<typename
            AutoPasType::Particle_t >::iterator_t;

        // adding the init method to the module
        autoPas.method("init", &AutoPasType::init);

        // adding the overloaded deleteParticle method to the module
        autoPas.method("deleteParticle", static_cast<void (AutoPasType::*)(
            typename AutoPasType::Particle_t&)> (&AutoPasType::deleteParticle)
            );
        ...
};

void setBoxMin(autopas::AutoPas<MoleculeJ<double>>& autoPasContainer, jlcxx::
    ArrayRef<double,1> boxMin) {
    autoPasContainer.setBoxMin({boxMin[0], boxMin[1], boxMin[1]});
}

bool iteratePairwise(autopas::AutoPas<MoleculeJ<double>>& autoPasContainer,
    ParticlePropertiesLibrary<> particlePropertiesLibrary) {
    autopas::LJFunctor<MoleculeJ<double>, true, true> functor{
        autoPasContainer.getCutoff(), particlePropertiesLibrary
    };
    return autoPasContainer.iteratePairwise(&functor);
}

JLCXX_MODULE define_module_autopas(jlcxx::Module& mod)
{
    using jlcxx::Parametric;
    using jlcxx::TypeVar;

    /**
     * add AutoPas type to Julia with the template parameter
     * MoleculeJ<double> and MoleculeJ<float>
     */
    mod.add_type<Parametric<TypeVar<1>>>("AutoPas")
```

```
            .apply<autopas::AutoPas<MoleculeJ<double>>, autopas::AutoPas<
                MoleculeJ<float>>>(WrapAutoPas());

    // add the setBoxMin function to the module
    mod.method("setBoxMin", &setBoxMin);

    // add the iteratePairwise function to the module
    mod.method("iteratePairwise", &iteratePairwise);
}
```

Listing 3.2: Wrapping of the templated class AutoPas.

Some of the member functions have a std::array as a function argument or return type, like `setBoxMin`. However, CxxWrap only maps a few C++ standard library (STL) types to corresponding Julia types, but not std::array. Therefore, we define a new helper function in C++, which takes a reference to an AutoPas container and an instance of jlcxx::ArrayRef. The jlcxx::ArrayRef is a data type, defined by CxxWrap, which maps a Julia vector to a C++ type. Inside the helper function, we call the original function from the AutoPas container object and transfer the values of the ArrayRef object into a std::array object.

One of the most important functions of AutoPas is the `iteratePairwise` function, which iterates over all particles in an AutoPas container to calculate the force between particle pairs efficiently. As shown in listing 3.1 the function itself is parameterized and with CxxWrap it is even possible to wrap a parametrized function, but the author of the thesis was not able to wrap a parametrized function within a parametrized class. Hence, we choose the same methodology as above and call the `iteratePairwise` function out of a helper function. For now, the functor is hardcoded, but this can easily be changed.

3.1.2. Inplementation of the MoleculeJ Class and its Wrapper

In the previous subsection, we mentioned that the `AutoPas` class expects a template parameter and in this thesis, we use the type `MoleculeJ`[2]. In [6] it is explained that a custom particle class may be implemented to use AutoPas and it is easy if the class inherits from the class `ParticleBase`, which is already provided by AutoPas.

Listing 3.3 indicates that our newly defined class `MoleculeJ` inherits from `ParticleBase`. Moreover, a big part of this class is similar to the already provided class in the AutoPas repository called `MoleculeLJ`. A few adjustments are necessary, to use the particle type for the Julia simulator. As already mentioned, we need to exchange the type `std::array` for the type `jlcxx::ArrayRef`, which is the case for the `getters` and `setters` of the attributes `_r` (position), `_v` (velocity), `_f` (force) and `oldForce`, as well as for the `constructor`. Implementing the `getters` is easy, as we only need the pointer to the array, as well as its size. For the `setters`, we need to construct a new `std::array` object and this object is used to set the variable.

[2]Although this section is dedicated to the wrapper code, we also include how the class `MoleculeJ` is implemented, as it does not fit better in any other context

The `typeId` and `moleculeId` attributes are important to mention as well. The first attribute can be used to distinguish between two different molecule types or molecular structures (but in this thesis, we only use one molecule type) and the second one is used to assign every particle an individual index, which is used in section 3.3 for parallelization strategies.

Additionally, we added a further getter implementation, which only returns one single floating point value, and this getter takes an integer as an argument, which represents the index of the desired value of the array e.g., `getP(1)` returns the second element of the position array. To get all the positions, we need to call this getter three times. It may be interesting to analyze if the two getter implementations perform differently, which is discussed in section 4.3.

Due to the fact that `MoleculeJ` is a parametrized type, we follow the same procedure to wrap the code as explained in the previous section.

```cpp

template<typename floatType>
class MoleculeJ : public autopas::Particle {
    public:

    MoleculeJ(jlcxx::ArrayRef<floatType,1> pos, jlcxx::ArrayRef<floatType,1> v
        ,
        unsigned long moleculeId, unsigned long typeId = 0) : _typeId(typeId)
            {
        std::array<floatType, 3> pos_{pos[0], pos[1], pos[2]};
        std::array<floatType, 3> v_{v[0], v[1], v[2]};
        setPosition(pos);
        setVelocity(v);
        setID(moleculeId);
    }

    void setPosition(jlcxx::ArrayRef<floatType,1> pos) {
        ParticleBase::setR({pos[0], pos[1], pos[2]});
    }

    // getter implementation returning ArrayRef object
    jlcxx::ArrayRef<double,1> getPosition() {
        return {_r.data(), _r.size()};
    }

    // getter implementation returning single values
    double getP(int i) {
      return _r[i];
    }
};
```

Listing 3.3: Abstract of the `MoleculeJ` class and the different getter implementations. adapted from [6]

3.1.3. Wrapping of Enums with CxxWrap

As mentioned in section 3.1.1, `AutoPas` uses options e.g., the class `IteratorBehavior`, which we discuss as an example in the following. This class is used to tell the iterator which particles need to be considered for the iteration, for example, it is possible to iterate only over particles outside of the simulation domain. Listing 3.4 shows the class `IteratorBehavior` and it has an enum defined inside, called `Value`. It is necessary to wrap the type `IteratorBehavior` as well as the enum `Value` if we want to specify the behaviour of the iterator from Julia. The type can be added to the C++ module as usual. CxxWrap represents an enum as an `isBits` type and the syntax to add an enum is different than for a normal type, as shown in the same listing.

```

namespace autopas {
inline namespace options {

class IteratorBehavior : public Option<IteratorBehavior> {
 public:

  using Value_t = unsigned int;

  /**
   * Different possibilities for iterator behaviors.
   */
  enum Value : Value_t {

    owned = 0b0001,
    halo = 0b0010,
    ...
  }
  ...
};


JLCXX_MODULE define_module_options(jlcxx::Module& mod) {
    /**
     * add the enum of IteratorBehavior to the module
     */
    mod.add_bits<autopas::options::IteratorBehavior::Value>("
        IteratorBehaviorValue", jlcxx::julia_type("CppEnum"));
    mod.set_const("owned", autopas::options::IteratorBehavior::Value::owned);
    mod.set_const("halo", autopas::options::IteratorBehavior::Value::halo);

    /**
     * add IteratorBehavior type and constructor to Julia
     */
    mod.add_type<autopas::options::IteratorBehavior>("IteratorBehavior")
              .constructor<autopas::options::IteratorBehavior::Value>();
}
```

Listing 3.4: The IteratorBehavior class and how to wrap an enum with CxxWrap. adapted from [6]

Furthermore, we can see that `IteratorBehavior` inherits from the `Option` class and this class has the template parameter `IteratorBehavior`. This is also called **curiously recurring template pattern (CRTP)**. CxxWrap supports inheritance and specifying the supertype may be the standard way to wrap this type. However, in CxxWrap, we always need to add a type or a function to a C++ module before we use it e.g., as an argument type of a function or as a type for a template parameter. Adding the `Option` class to the module before the class `IteratorBehavior` does not work, as we need to specify all combinations of the templated types when the type is added to the module. The template type `IteratorBehavior` is not known at this time, as it is defined later. If the class `IteratorBehavior` is added before the `Option` class, we need to specify the supertype, which is `Option` and `Option` is not known as well. This only explains why we have not used the method to specify the inheritance structure.

3.1.4. Overview of the Wrapped Modules

Up to this point, three different types were wrapped. In addition, we need the class `ParticlesPropertiesLibrary`, which contains the data about the particles, like coefficients for the Lennard-Jones potential or the mass of the particle. The classes `ParticleIteratorInterface` and `ParticleIteratorWrapper` are needed as well, because they implement the operators `++` and `*` so that we can actually use the iterator returned by `begin`. However, no new special approach is needed to wrap these types and we do not go into more detail.

The five discussed components: AutoPasInterface, Particles, AutoPasOptions, ParticlePropertiesLibrary, and Iterators, are all added to their own C++ module. There is another difference to the approach explained in section 2.5.2 as the method `define_julia_module` is named differently e.g., `define_module_iterators` for the module containing the logic regarding the iterator. This creates a better structure of the code and it is easier to add new functions or types to the wrapper.

3.1.5. Module Creation on the Julia Side

In the last step, we need to write Julia code to make the functions from AutoPas available in Julia. We define the module `Simulator`, which contains all the subsequent modules. In section 2.5.2 we already showed how to specify the path to the shared library and this needs to be done for all five created modules as well. Because the `define_julia_module` function is replaced by e.g., `define_module_particles` for the module containing the particle functionality, we need to specify this in the `@wrapmodule` macro, shown in listing 3.5. Of course, the name is adapted, depending on the module name.

```julia

module Simulator

module Particles
  using CxxWrap
  @wrapmodule(joinpath("path/to/lib","libname.so"), :define_module_particles)

  function __init__()
    @initcxx
  end
```

```
end # module Particles

module Iterators

    ...

end # module Iterators

    ...

end # module Simulator
```

Listing 3.5: Creating Julia modules of the wrapped C++ functionality of AutoPas.

3.2. Simulator

The following section describes how we implemented the simulator and use the created wrapper from section 3.1 to call the AutoPas functions and types from Julia. If not stated otherwise, we can assume that we always use the Linked Cells algorithm in this and the following sections.

3.2.1. Initialization of Simulation Parameters

Before starting the simulation, we need to initialize the AutoPas container and set all relevant parameters for the simulation.

Most important for a MD simulation is the creation of particles. We use so-called generators to create particles in a specific arrangement and we support a cube grid[3] and a cube uniform[4] generator. For each arrangement, we created a struct e.g., the `CubeGridInput` struct form listing 3.6, which inherits from the abstract type `ParticleObjectInput` and has all the relevant data to create the particles at the correct position and with the correct velocity. The values for this struct need to be written into the code directly, as we do not support parsing of command line arguments or an input file parser. We added a constructor without any arguments with the `new()` keyword so that we can create an uninitialized object and specify every argument one after another. The advantage is to initialize an arrangement in the Julia REPL interactively and easily.

```
# definition of the cube grid arrangement
mutable struct CubeGridInput <: ParticleObjectInput
    particlesPerDimension::Vector{Int64}
    particleSpacing::Float64
    bottomLeftCorner::Vector{Float64}
    velocity::Vector{Float64}
    particleType::Int64
    particleEpsilon::Float64
```

[3]To create a cube grid, we define how many particles we like to have in each of the three spatial dimensions, as well as a constant distance between each particle

[4]To create a cube uniform, we define the length of a cuboid and insert particles with randomly uniformly distributed positions into the cuboid.

```
    particleSigma::Float64
    particleMass::Float64
    factorBrownianMotion::Float64
    CubeGridInput() = new()
end

# initialization of the cube grid struct in the code

cubeGrid = CubeGridInput()

cubeGrid.particlesPerDimension = [10, 10, 10]
cubeGrid.particleSpacing = 1.12
    ...
```

Listing 3.6: Definition of the cube grid struct and how it can be initialized.

Other input parameters like the AutoPas container, are set in the struct called `input-Parameter` and again the struct is initialized in the code directly, as already described in the previous paragraph. Furthermore, we implemented a function called `parseInput`, which creates an AutoPas container object, adds the generated particles to the container, and returns the container, as well as an instance of the `ParticlesPropertiesLibrary`.

3.2.2. Calculation of Position and Velocity

The force calculation is already implemented in AutoPas so we still need to implement the update of the position r, formula 3.1 and velocity v, formula 3.2. Both formulas are adapted from the Velocity-Störmer-Verlet method from [3].

$$r(t+\delta t) = r(t) + \delta t v(t) + \frac{\delta t^2}{2m} F(t) \tag{3.1}$$

$$v(t+\delta t) = v(t) + \frac{\delta t}{2m} (F(t) + F(t+\delta t)) \tag{3.2}$$

With listing 3.7 we explain how the position calculation is realized in Julia, the velocity calculation is implemented similarly. The iterator needs to be initialized as the very first step and with `isValid`, we can verify if the iterator still points to a particle. With `Sim-ulator.Iterators.:*` we can dereference the iterator and get the particle of the current iteration. For the calculation, we need the velocity and the force as seen in formula 3.1. We set the value of the `oldForce` to the value of the current force because formula 3.2 needs the value for the velocity calculation. To calculate e.g., $v \cdot \delta t$, we can simply use the `.*` operator from Julia base and this computes the element-wise multiplication of a vector with a scalar, similarly we can add two vectors with `.+`. We use `getTypeId` to get the ID of the type of the molecule and with this ID we can get the correct mass. With `addPosition`, a member function of `MoleculeJ`, we can add the adjustment of the position to the current position to get the updated position. The `++` operator is called so that the iterator points to the next particle. As the iterator operators are not part of Julia base, we need to specify the path to the module which defines the operators, which is `Simulator.Iterators` due to the module structure discussed in section 3.1.5.

```julia
# in this code snippet pPL stands for particlesPropertiesLibrary
function updatePositions(autoPasContainer, deltaT, pPL)
    iterOption = IteratorBehavior(Options.owned)
    iter = AutoPasInterface.begin(autoPasContainer, iterOption)
    while isValid(iter)
        particle = Simulator.Iterators.:*(iter)
        velocity = getVelocity(particle) # velocity is of type Vector{Float64}
        force = getForce(particle) # force is of type Vector{Float64}
        setOldF(particle, force)
        velocity .*=  deltaT
        typeId = getTypeId(particles)
        force .*= (deltaT * deltaT) / (2*getMass(pPL, typeId))
        force .+= velocity
        addPosition(particle, force)
        Simulator.Iterators.:++(iter)
    end
end
```

Listing 3.7: Calculation of the new position in the Julia programming language.

3.2.3. Implementation of Boundary Conditions

Depending on the simulated scenario, we need to handle the particles at the boundary of the simulation domain, or the ones crossing the boundary. Below, we discuss how we implemented the outflow and periodic boundary condition.

AutoPas already provides a function called `updateContainer` which collects all particles outside the domain. Furthermore, it deletes the particles from the AutoPas container and returns a vector of these particles. As the outflow boundary condition means deleting all particles outside of the simulation domain, we can easily use this function to implement this boundary condition.

For the following boundary condition, we define the term **halo particle**. This particle is a copy of a particle that is near the boundary inside the simulation domain. The halo particle is used as a dummy particle to calculate the force correctly. Halo particles are always outside of the simulation domain.

Another convention we use related to boundary conditions are the terms 'upper' and 'lower', e.g., the upper boundary area in the x direction describes the boundary area on the right side, and the lower one describes the left side. For the y dimension, in a two-dimensional case, the 'upper' boundary area is indeed the upper area. In three dimensions the 'upper' area describes the area in the back of a cuboid.

The periodic boundary condition is explained with the help of figures 3.1 and 3.2. The orange-shaded area is called the boundary area and the purple one is called the outer area. The blue line indicates the boundary of the simulation domain so that the boundary area is still inside the simulation domain, but the outer area is outside of it. Both areas have a width of one cutoff unit and we assume that a single particle can only move as much as one cutoff unit in one iteration.

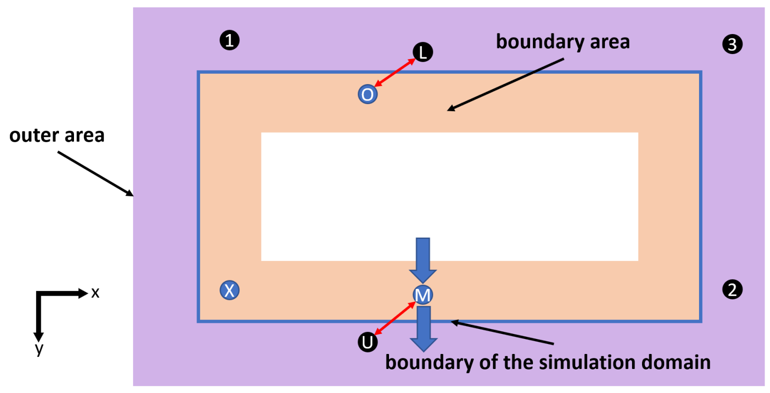

Figure 3.1.: Inserting halo particles for all particles in the boundary area of the simulation domain.

Particle M moves into the lower boundary area and this means that M would theoretically interact with particles in the upper boundary area of a neighboring simulation domain. However, as mentioned in section 2.1 and illustrated in figure 2.2, we only simulate one simulation domain. For correct results of the force calculation, we create a copy of particle M called halo particle L in black and add the length of the y dimension of the simulation domain to its y-coordinate, as illustrated in figure 3.1. Analogously, we need to subtract the same value from the same coordinate for particles in the upper boundary area or we need to change another coordinate e.g., for particles in the left or right boundary areas. As the halo particle is outside of the simulation domain, we need to set the ownership state of the particle to `halo` and use the `addHaloParticle` function of AutoPas to add it to the container. For the force calculation for particle O in the upper boundary area, the halo particle L is used and of course, our implementation inserts a halo particle for particle O (represented by particle U) in the lower outer area which is used for the force calculation for particle M. After every iteration step, the halo particles are deleted, because they are 'fixed' e.g., we do not calculate the force or a new position for them. The deletion is done with the function `updateContainer`, as per definition the halo particles are always outside of the simulation domain. If particle M happens to be in the boundary area in the next simulation step, we insert a new halo particle with the same procedure.

The next scenario is illustrated by using particle P coloured in blue in figure 3.2, which moved out of the simulation domain and is in the outer area. Per definition, we insert it on the opposite side, which is equivalent to adding the x length of the simulation domain to its x-coordinate. The new position is indicated by the red P on the right side. After moving the particle to the new position, we need to insert a halo particle at the opposite boundary, as indicated with particle P in black, because the blue particle P is now in the boundary area. Again, depending if we are in the upper or lower area, we need to subtract or add the length of the corresponding side from or to the coordinate of the particle. The coordinate depends on the fact if it is e.g., a boundary in the x direction or in the y direction.

A case we have not considered in the two-dimensional space is if a particle is in the corner of the boundary area, like the blue particle X. For this particle, we need to insert three halo particles, at the positions of the black particles 1, 2, and 3, as seen in figure 3.1. The reason is again the interaction with neighboring simulation domains, and there are exactly three of them. For a more detailed description of why this is the case, [7] gives a good and detailed overview.

A similar concept applies to particles that are in the corner of an outer area. In the case of two dimensions, we need to modify both coordinates to move the particle to the correct new position. This is illustrated with particle S in figure 3.2 and the end position is indicated by particle 2 in the same figure.

In the case of three dimensions, we need to consider one more dimension e.g., a particle in the corner of the boundary area in a cuboid 'creates' halo particles, at every other remaining corner of the cuboid. A particle in the corner of the outer area of a cuboid is moved in all three dimensions.

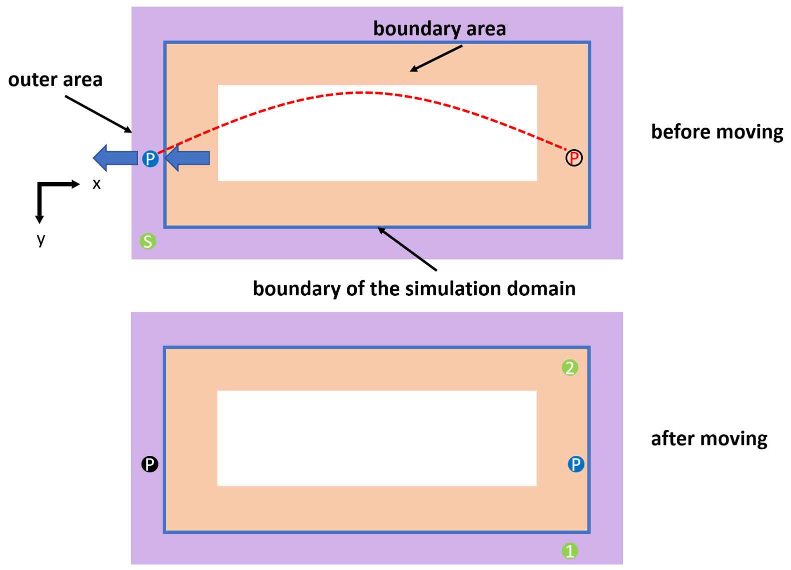

Figure 3.2.: Particle moves out of the simulation domain and is reinserted at the opposite side of the domain.

After the general logic is explained, we talk about the detail of the implementation now. We created a function called `applyPeriodicBoundary`, which is composed of two steps: moving outer particles and inserting halo particles. For both functions, AutoPas provides an overly helpful function, the `regionIterator`. If we pass a minimum and maximum coordinate of an area to the iterator, the iterator only returns particles inside the defined area and this iterator is more performant than the normal iterator.

For the first function, called `moveOuterParticles`, we iterate over a loop, once for each spacial dimension, and in this loop, we first calculate the minimum and maximum coordinate of the 'upper' and 'lower' outer area. In the next step, we pass these points to the `regionIterator` and iterate over the returned particles. All three positions of the position attribute are checked if they are out of the simulation domain and the position is corrected immediately.

The second function is called `insertHaloParticles`. Again, we calculate the 'lower' and 'upper' areas of all the halo areas and iterate over the particles returned by the `regionIterator`. If we handle the halo area in the x dimension, we additionally check, if the particle is in a halo area of a y and/or z dimension and insert the halo particles accordingly. If we iterate over particles of a halo area of the y dimension, we further check if the particle is also in a halo area of the z dimension. Now we covered all cases and insert the halo particles properly in all dimensions.

3.2.4. Description of the Simulation Loop

We know all the necessary parts of a simulator, but we need to use them together in the right order to create a working simulator, as presented in listing 3.8. As already mentioned, we need to initialize the `autoPasContainer` and create the `particlePropertiesLibrary`, which is executed in the `parseInput` function. The first operation in the simulation step is the position calculation. The handling of the boundary condition follows and we like to use the outflow boundary condition for this simulator. Hence, we use `updateContainer`, to delete particles outside of the simulation domain. The last two steps include the force and velocity calculation.

```julia
function simulate(inputParameters)
    # pPL stands for particlePropertiesLibrary
    autoPasContainer, pPL = parseInput(inputParameters)

    deltaT = inputParameters.deltaT

    for iteration in 1:inputParameters.iterations

        updatePositions(autoPasContainer, deltaT, pPL)
        updateContainer(autoPasContainer)
        updateForces(autoPasContainer, pPL)
        updateVelocities(autoPasContainer, deltaT, pPL)
    end
end
```

Listing 3.8: Simple Julia Simulator with outflow boundary condition.

3.2.5. Custom Molecule Implemented in Julia

Chapter 4 discusses the performance of the implemented Julia simulator. In order to measure the overhead of the wrapper, we implemented a Julia struct representing a particle, as shown in listing 3.9. Furthermore, we implemented the `updatePositions` function for this particle type, similar as already discussed in 3.2.2. Instead of using a 'normal' Julia `Vector` or `Array` for attributes like position or velocity, we use `SVector`, which is a type from the module `StaticArrays`. `StaticArrays` promises better performance compared to standard Julia data structures.[5]

```julia
mutable struct Molecule
    position::SVector{3,Float64}
    velocity::SVector{3,Float64}
    force::SVector{3,Float64}
    oldForce::SVector{3,Float64}
    moleculeId::Int64
    typeId::Int64
end
```

Listing 3.9: Julia struct representing a molecule.

3.3. Usage of Multithreading to Speed up the Simulation

To speed up the simulator discussed in the previous section, we use Julia's built-in multithreading functionality to parallelize the code, hence, we utilize the term **thread** instead of the more abstract term processor in this section.

As the force calculation is already executed by Autopas, we can only parallelize the code we implemented ourselves, which is the position and velocity update. In listing 3.10 we use the function `updatePositionsParallel` to illustrate the parallelization strategy, but this also applies analogously to the velocity update.

As shown in listing 3.7, we use a `while` loop to iterate over all particles in the AutoPas container and this loop is the operation we like to parallelize, such that particles are processed by different threads simultaneously. In Julia, it is only possible to parallelize a `for` loop so we need to transform the `while` loop into a `for` loop. But the problem is, that a `for` loop in Julia is not like the classic C++ `for` loop `for (int i = 0; i < x; i++)`, but rather a range-based one. The solution to this problem is to use a `for` loop ranging from 1 to the number of used threads and inside this loop we call the function `updatePositionSequential`, which does the computation for a subset of particles. With the Julia macro `@threads`, we can annotate the `for` loop which means that the loop is parallelized and the tasks are distributed to multiple threads.

For the iteration over the particles we use the iterator defined by AutoPas and we need to pass an iterator instance to each function call of `updatePositionSequential`. As every iteration of the `for` loop is managed by a different thread, we need to make sure that the threads do not process the same particles. There are two options to solve this.

1. Starting at index i, Thread T_i calculates only the position of every $(n \cdot t)$-th element of the container, with $n \in \mathbb{N}$ and t the number of threads, as shown in figure 3.3.

[5] `https://github.com/JuliaArrays/StaticArrays.jl`

2. Thread T_i calculates the positions of the particles in the range of the $i\frac{N}{t}$-th to $(i+1)\frac{N}{t} - 1$-th particle of the container, with t the number of threads and N the number of particles as shown in figure 3.4. If N is not a multiple of t, one thread needs to calculate the position of the remaining particles.

Both options are implemented and the results are compared in section 4.3.

Listing 3.10 presents the first of the two options. At first, we increase the iterator instance in a loop, so that it points to the correct particle e.g., in the first iteration we do not increase the iterator so that it points to the first particle. In the second iteration, we increase it once so that the iterator points to the second particle in the container. This iterator instance is passed to the function `updatePositionsSequential` and we proceed with the calculation as discussed in 3.7 up to the point, the iterator needs to be incremented. In a loop, we increment the iterator as many times as threads are available.

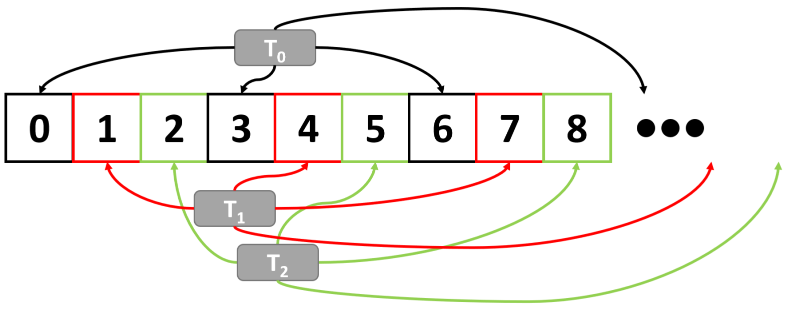

Figure 3.3.: Parallelization Strategy 1 for t = 3: the particles are distributed to three threads and each thread processes every third particle.

Option two is slightly different, as we need to know the number of particles in advance. In `updatePositionsParallel` we calculate the first and the last index of the chunk and increase the iterator until the correct position (equals to the calculated first index) is reached. This iterator instance is passed to `updatePositionsSequential`. In this function, we can change the loop to iterate from the calculated first index to the last index without checking if the iterator is still valid in every iteration. Furthermore, the iterator object is incremented by one every single iteration.

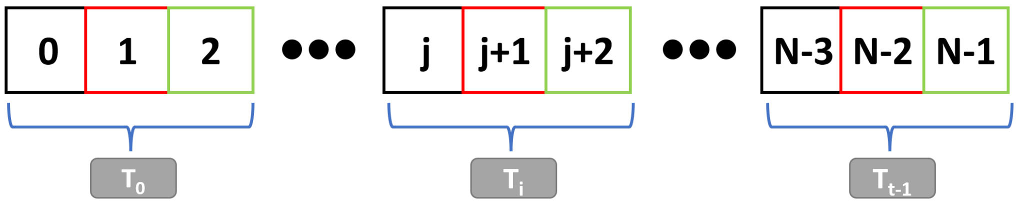

Figure 3.4.: Parallelization Strategy 2: the particles are divided into chunks of the same size (if possible) and each thread processes one chunk.

```

function updatePositionsSequential(autoPasContainer, deltaT, pPL, nthreads,
    iter)

    while isValid(iter)
       # original force calculation
             ...
        for i in 1:nthreads
            Simulator.Iterators.:++(iter)
        end
    end
end

function updatePositionsParallel(autoPasContainer, deltaT, pPL)
    nthreads = Threads.nthreads()
    iterBehavior = IteratorBehavior(owned)
    Threads.@threads for i in 1:nthreads
        iter = AutoPasInterface.begin(autoPasContainer, iterBehavior))
        for j in 1:(i-1)
            Simulator.Iterators.:++(iter)
        end
        updatePositionsSequential(autoPasContainer, deltaT, pPL, nthreads,
            iter)
    end
end
```

Listing 3.10: Parallelization strategy 1 of the `updatePositions` function.

In the following simulator run, we used parallelization strategy 1 if not stated otherwise.

Additionally, we parallelized the position calculation for the Julia particle. As the particles are stored in a vector, we can easily iterate over it with a `for` loop, and after annotating the `for` loop with `@threads` the loop is easily parallelized.

3.4. Distribution of the Simulation to Multiple Processors

With distributed memory parallelization, we like to further optimize and speed up the simulator. This section discusses the implementation of the domain decomposition, the communication between processors, as well as the necessary adjustments for correct boundary conditions.

3.4.1. Explanation of Using MPI.jl for Distributed Memory Parallelization

For the distributed memory parallelization strategy, we use the Julia package MPI.jl[6] which is a wrapper for the Message Passing Interface (MPI) of C. The following reasons motivate the utilization of MPI.jl. Firstly, according to [4], MPI.jl is a better choice than the Distributed.jl package[7] from Julia if more complex and bigger parallelization is desired. Secondly, MPI.jl promises no overhead in comparison to C implementations (if no structs are transferred).

[6] `https://github.com/JuliaParallel/MPI.jl`
[7] `https://docs.julialang.org/en/v1/manual/distributed-computing/`

Thirdly, for C/C++ users, MPI.jl provides similar syntax and similar or the same function names, to make a smooth transition from C/C++ to Julia.

3.4.2. Implementation of the Domain Decomposition

As already discussed in section 2.2.2, the simulation domain needs to be decomposed into smaller parts with each part being simulated on a different processor. In this thesis, we use one of the easiest domain decomposition strategies and only divide the global simulation domain along the x-axis, as shown in figure 2.3. The number of subdomains corresponds to the number of available processors. To execute the code on different processors, we use MPI.jl and call the function `MPI.Init()` in the very beginning of the simulation (even before the initialization of the AutoPas container), because this function distributes the code to all used processors, and the code after the function call is executed on each processor in parallel.

3.4.3. Implementation of the Communication between Processors

We know, how the domain is decomposed into subdomains and the next step, the communication between the processors, is discussed in this section.

In our scenario, we only need to communicate with the left and right neighboring subdomains.

Figure 3.5 shows the two-dimensional simulation domain and subdomain i, which is managed by processor P_i. The black dotted lines represent the boundaries of the subdomains in x direction. Furthermore, we can see a green-shaded area, which is called migration area, and a blue-shaded area, the halo area. The scenario is explained between subdomain i and $i-1$, but the same applies to subdomain i and $i+1$.

Particles, like the green particle M, move outside of the subdomain i into the green-shaded area, indicated by particle M in black. This means it enters the subdomain $i-1$ and processor P_i needs to send the data of this particle to P_{i-1}. As this particle does not need to be processed by P_i anymore, it can be deleted from its container.

The particles K and U are managed by P_{i-1}. The force calculation for particle K is only correct if we consider the interaction between K and U as well as between K and H. However, H is not inside the memory of P_{i-1} so P_i needs to send the data of particle H to its neighbour. As H is outside of subdomain S_{i-1}, from the perspective of S_{i-1}, it is inserted as a halo particle. It is only used to calculate the force of particle K and the halo particle is deleted after every simulation step. P_i cannot delete H, as it is still inside S_i.

The functions `exchangeMigratingParticles` and `exchangeHaloParticles` are used to represent these two scenarios in the source code. In the following, we discuss how the `exchangeMigratingParticles` function is implemented, and listing A.1 shows a simplified

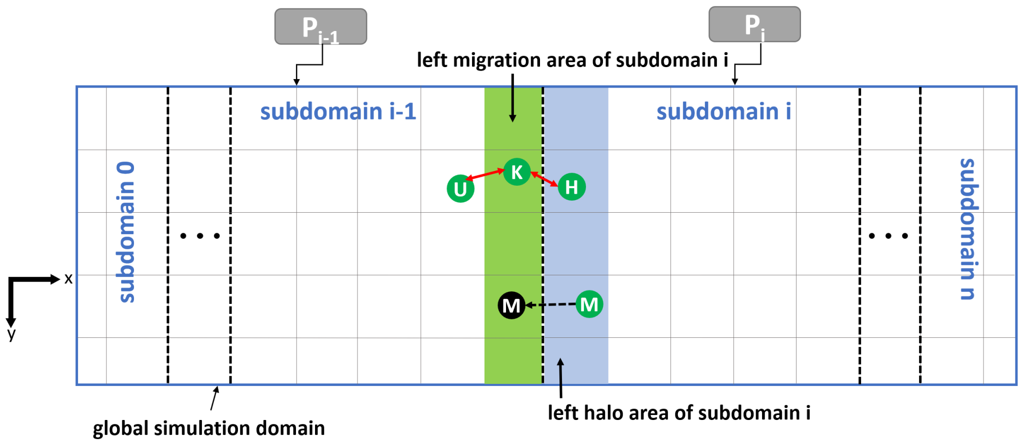

Figure 3.5.: Visualizing cross-border interactions between subdomains and the migration of a particle from subdomain i to subdomain $i-1$.

version of the code. `exchangeHaloparticles` is implemented similarly.

At first, we like to remember that the width of the migrating area is as big as the defined cutoff. After calculating the edge points of this area, we pass these values to the `regionIterator` to only get the particles in this area. The second step is to send these particles to another processor. We need to change the data so that MPI can send it, which is called serialization. As we can send arrays with MPI, we write the values of the attributes in the memory subsequently, e.g. at first the three position coordinates, after this the three velocity values, etc. This is repeated for every particle that is sent to a neighboring processor.

The serialized particles are passed to the `sendAndReceiveParticles` function which actually does the communication via MPI. The communication is realized with the functions `MPI.Irecv!` to receive a message and with `MPI.Isend` to send a message. Depending on the scenario, the number of exchanged particles differs, so we exchange the number of particles as the first message so that the receiving processor can allocate a big enough array. The send and receive calls need to be done two times, for the left and right neighbour exactly once. With `MPI.Waitall`, we wait until the communication is completed. After this, we use the same functions to exchange the particles and again, it is necessary to call the send and receive functions two times and to wait until all the communication is completed.

The received messages represent serialized particles, hence, we need to deserialize them back into particle objects and add the particles to the AutoPas container. In the case of the `exchangeMigratingParticles` function, we need to delete the sent particles from the container, as they are out of the current subdomain. This step is not relevant in the `exchangeHaloParticles` function.

```

function exchangeMigratingParticles(autoPasContainer, domain, comm)
```

```

    leftMin, leftMax = calculateLeftArea(domain)
    rightMin, rightMax = calculateRightArea(domain)

    ePL = [] # exchangeParticlesLeft
    iter = regionIterator(autoPasContainer, leftMin, leftMax, IteratorBehavior
        (owned))
    while isValid(iter)
        push!(ePL, Simulator.Iterators.:*(iter))
    end

    # loop over the right side as well

    receiveParticles = sendAndReceiveParticles(ePL, ePR, domain, comm)

    for p in ePL
        deleteParticle(autoPasContainer, p)
    end

    # same for the right particles

    for p in receiveParticles
        addParticle(autoPasContainer, p)
    end

end
```

Listing 3.11: Exchanging particles between processes with MPI.jl.

3.4.4. Adjustments for Periodic Boundary Conditions

As mentioned in section 3.2.3, we implemented the outflow and periodic boundary condition. To deliver correct results using distributed memory parallelization and the boundary conditions, we may need to make some changes.

The outflow boundary condition is still implemented with the `updateContainer` function because we set the boundaries of the AutoPas container to the values of its local boundaries. Hence, particles outside of the container are deleted and no adjustments are necessary.

However, we need to modify the code for the periodic boundary condition, because up to this point, the implementation would produce wrong results. To understand the problem, we like to bring some terms to mind. In section 3.4.3 we introduced the terms halo and migration area which are used if we talk about subdomains. In section 3.2.3 we defined boundary and outer area which are used in the context of the global simulation domain.

In the case of particles being in the boundary or outer area of the y or z dimension, the procedure of the boundary condition stays the same. If a particle is in the halo area of the x dimension, we do not apply the periodic boundary condition if the x boundary is not a boundary of the global simulation domain.

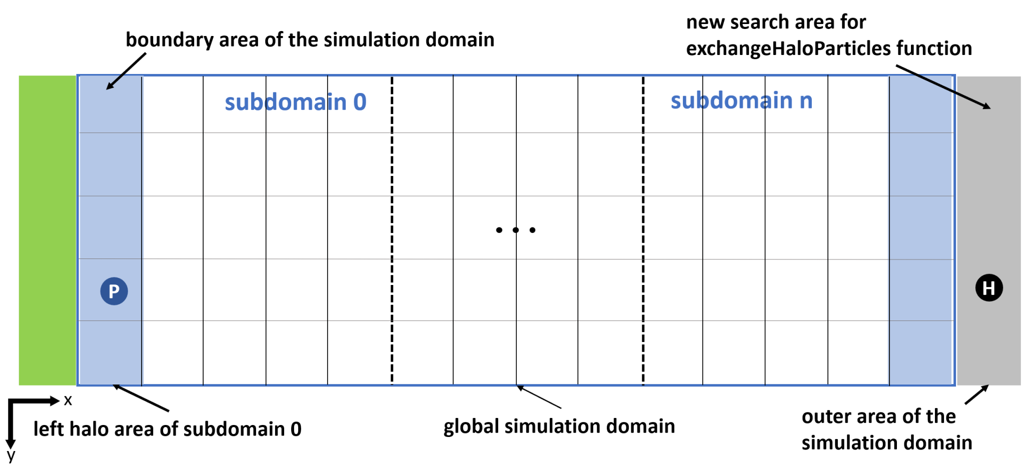
boundary area of the simulation domain
new search area for exchangeHaloParticles function
subdomain 0
subdomain n
P
H
x
y
left halo area of subdomain 0
global simulation domain
outer area of the simulation domain

Figure 3.6.: Search space for periodic boundary conditions for outermost subdomains.

The need for a change occurs if a particle is in the halo area and adjacent to a global x boundary. In this case, we need to apply the boundary condition for all three dimensions. The outer halo areas of the outermost subdomains and the boundary areas of the global simulation domain include the same space, as shown in figure 3.6. For demonstration, we use the blue particle P in subdomain 0. The basic principle of the boundary condition still applies i.e., particle P 'produces' a halo particle on the opposite side indicated with the black particle H. But the created halo particle is still managed by the processor of the subdomain 0, but according to the defined domain decomposition in section 3.4.2, H would be managed by the processor of subdomain N. This means that this particle needs to be sent to another subdomain.

In case that a particle is inside the green-shaded migration area/outer area, we need to apply the periodic movement to the particle i.e., inserting the particle into the simulation box on the opposite side. Again, the particle would move across the boundary of the subdomain and we need to send the data of the particle to the correct processor.

In the implementation, we need to change the code at three positions. The first one is the function `insertHaloParticles` and we check if the current x boundary is a global boundary. If not, we do not insert halo particles, in the other case we proceed as described in 3.2.3. The second function we change is `exchangeMigratingParticles` and we verify if the particles in the migration area are outside of the simulation domain. If this is the case, we change their position already in this function and send the already-moved particles to the next processor. Lastly, we change the `exchangeHaloParticles` function, specifically the part that calculates the area, we need to look for halo particles. If the x boundary happens to be a global boundary, we change the coordinates of the search area to be on the opposite side, indicated by the gray-shaded area in figure 3.6.

3.5. Annotation to the Implementation of the Periodic Boundary Condition and the Usage of MPI

The implementation of the periodic boundary condition does not produce correct results. If we use the `regionIterator`, we do not iterate over all particles in the specified area, even if we added the particles to the container and the normal iterator returns them. That is why we only use the outflow boundary condition in the subsequent chapter for the performance tests.

Furthermore, using the simulator with more than one rank, resulted in a loss of particles during the simulation, which is not the case if it is used with just one rank. Due to the fact that more than 40% of the particles disappeared, we dispense performance tests for this implementation, as the reference implementation does not lose any particles and the results would not be based on equal scenarios.

4. Discussion and Results

This chapter includes a discussion of the utilization of Julia and the CxxWrap.jl package. We further present the results of test runs of the Julia simulator and a C++ reference implementation. First, we describe the used tools and the strategy for the test. Second, we have a detailed look at the `updatePositions` function of the Julia simulator. In the last two sections, we discuss the performance of the Julia simulator with an increasing number of particles as well as using multiple threads.

4.1. Reviewing the Usage of Julia in this Thesis

This first section discusses if the used approach, calling library functions from Julia with the help of CxxWrap, is beneficial in terms of the efficient development of MD simulations.

4.1.1. Using Julia for MD simulations

One part of this thesis is to find out if the Julia programming language is suitable for MD simulations. This section discusses our opinion as programmers with no prior Julia experience.

First of all, we start with a more general view of Julia. It is fairly easy to get started with programming in Julia, as first of all, we can easily run a program and e.g., in comparison to C++, we do not need any kind of build system.

Additionally, we noticed that the time for the compilation of the simulator code is rather low. After changing some code, the compilation does not take as long as for a reference implementation written in C++. Especially if templates need to be instantiated in C++, compilation may take some more time. This feature of Julia makes it possible to test small code changes very fast.

Furthermore, plenty of wrappers like the MPI.jl package already exist, and with the `ccall` function, we can easily use C functions (and with some more work C++ functions) from Julia. This allows us to adopt already existing code and write highly performant code.

The built-in parallelization options, like the multi-threading option in Julia, allow someone to speed up the code, with just a few commands. As we discussed in section 3.3 this is not necessarily applicable in every case, as the rather range-based style `for` loop in Julia cannot be used with the iterator from AutoPas and causes more implementation effort.

In the end, we like to point out, that Julia offers the possibility to create structs, equivalent to languages like C++. However, member functions cannot be created in a C++ sense. Additionally, we cannot find a way to use inheritance as done in AutoPas or C++ in general.

This may not be a disadvantage at all, it is just another style of programming which needs to be adapted if bigger and more complex programs are created in Julia.

4.1.2. Evaluation of the Julia Interoperability with CxxWrap

One of the first challenges encountered, while working on this thesis, is the wrapping of the C++ library AutoPas to be able to call its functions from Julia. The usability of CxxWrap is discussed in this section.

As already mentioned, it is possible to expose e.g., a function, with just one line of C++ code, with the package CxxWrap.jl. This is very convenient, particularly if many functions have to be wrapped. For 'no special cases', unlike the examples discussed in section 3.1, it is even possible to use the WrapIt.jl package to generate some parts of the wrapper code automatically.

We also used the option of CxxWrap to create multiple modules in Julia e.g., the module for all the enums, and are able to structure the code on the Julia side, as well as the wrapper code in C++ in a clean way. This provides a functional differentiation of the code, an easier extendable wrapper, and the option to only use some functionality e.g., only functions and types related to particles.

While writing the code, we may produce some wrong code and like to use a debugger. There are multiple ways to use a debugger in Julia[1]. With one package it is even possible to change code during debugging to see if the change eliminates the bug without restarting the debugging session.[2] Debugging code is not specifically a problem of CxxWrap. However, as we use Julia code as well as C++ code together in one program, it may be more complicated to find bugs, as we need to search for the error on multiple 'sides'.

Furthermore, the CxxWrap GitHub page and source code provide many examples of how to use CxxWrap, but proper documentation is not available. On the one hand, the documentation may be helpful if someone is searching for specific functions or syntax. Hence, we may need some more time to understand the wrapper or look for special cases. On the other hand, there are two videos of a talk about the CxxWrap.jl package[3] and these definitely help to understand how the wrapper works and how to use it.

[1]https://julialang.org/blog/2019/03/debuggers/#codetracking

[2]https://timholy.github.io/Rebugger.jl/dev/

[3]https://www.youtube.com/watch?v=u7IaXwKSUU0 and https://www.youtube.com/watch?v=VoXmXtqLhdo

4.2. Introductory Notes for Test Execution

In this section, we like to outline how we measured the time, which hardware was used, and mention the necessary software versions.

First of all, the Julia simulator is compared to a C++ implementation called md-flexible. This is a MD simulator included in the AutoPas repository and implements functionality like domain decomposition and position calculation. The input file for the md-flexible simulator can be found in the appendix section A. The same input parameters are used for the Julia simulator, to simulate the exact same scenario and to have a 'fair' comparison between both implementations.

Additionally, we ran the performance tests multiple times to get a statistically meaningful result.[4] First, we used only three runs per scenario and a hundred thousand particles, which would be sufficient for md-flexible as the results do not deviate strongly. However, the Julia results do differ strongly, hence, we decided to increase the particle size and number of repetitions. This resolved the issue, at least for scenarios under 16 threads for the Julia implementation.

To reproduce the results of the Julia simulator, we always ran the simulation loop, as explained in section 3.2.4, with one single iteration so that some parts of the code are already pre-compiled and the execution of the performance test is as quick as possible.

All performance tests were executed on the CoolMUC-2 cluster of the LRZ. For hardware details, please refer to the documentation.[5]

The used software versions are:

1. Julia: 1.8.5
2. CMake: 3.21.4
3. GCC: 11.2.0
4. CxxWrap.jl: 0.13.3 (code cloned on 05.03.2023)

To get timings from the md-flexible simulator, we used the implemented `Timer` class of AutoPas. At the end of a simulation run, all important timings are printed to the console. To measure the time in Julia, we used the package TimerOutputs.jl[6] which is similar to the `Timer` class of AutoPas and accumulates the time spend in the specified functions.

4.3. Inspecting the updatePositions Function of the Julia Simulator

In this section, we compare the runtime of different `updatePositions` implementations and try to find out if some optimizations can be used for better performance.

[4]There is one exception, to which reference is made at the respective location.
[5]`https://doku.lrz.de/display/PUBLIC/CoolMUC-2`
[6]`https://github.com/KristofferC/TimerOutputs.jl`

4.3.1. Comparing two Parallelization Strategies for the Position Calculation

The runtime of the two parallelization strategies of the `updatePositions` function, which are described in section 3.3, as well as profiling results are presented in this section. Furthermore, we use the term first strategy, to reference the algorithm illustrated by figure 3.3, and the term second strategy for the strategy illustrated by figure 3.4.

In figure 4.1, we can see the profiling results of the `updatePositionsSequential` parts of both strategies, ran with one thread. Two noticeable operations are the get operations of the velocity and the force of the particle, where we spend approx. 60% of the running time. In contrast, the time for an operation related to the actual calculation e.g., $\delta t v(t)$, from formula 3.1, takes only approx. 1% of the total time, hence very low. These results show, that the implementation is mainly limited by memory access. Furthermore, a big amount of time is used for iterator operations, like dereferencing and incrementing.

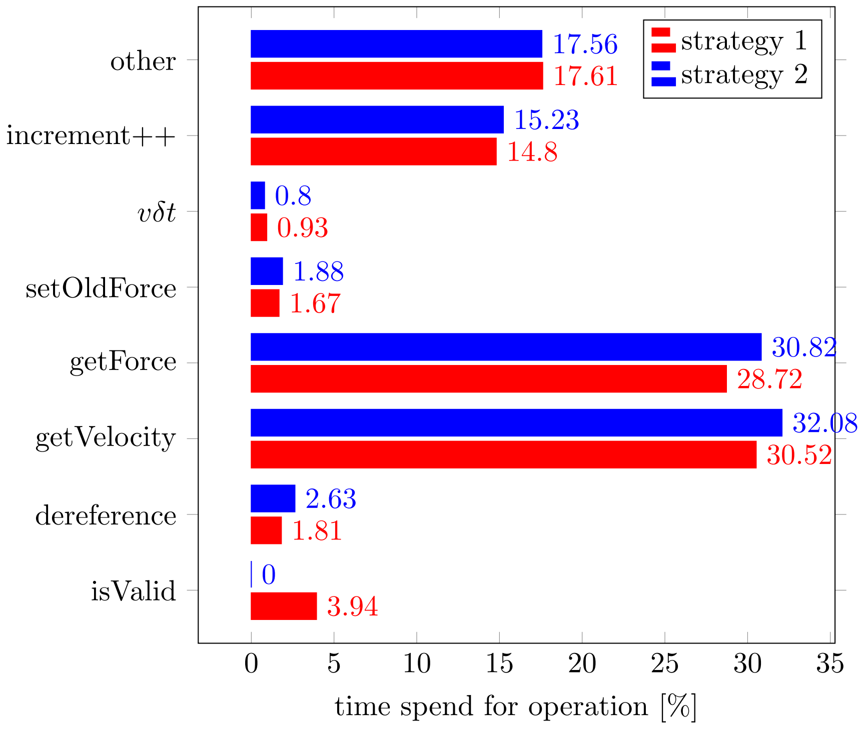

Figure 4.1.: Profiling results of the position calculation function of both parallel executable strategies. Both functions spend an enormous time in the getter functions

We tested both implementations with one million particles, one thousand iterations, as well as with one, two, and four threads. The results are presented in figure 4.2. Additionally, we included a plot of the 'ideal' runtime behaviour, which means, that if the number of threads increases by a factor of x e.g., two, the runtime decreases by a factor of x e.g., two. We can see, that both strategies do not show a big increase in performance, compared to the ideal behaviour. The runtime of the first strategy only decreases by approx. 21% and of the second by 27% if the number of threads is increased to two. The reduction of the

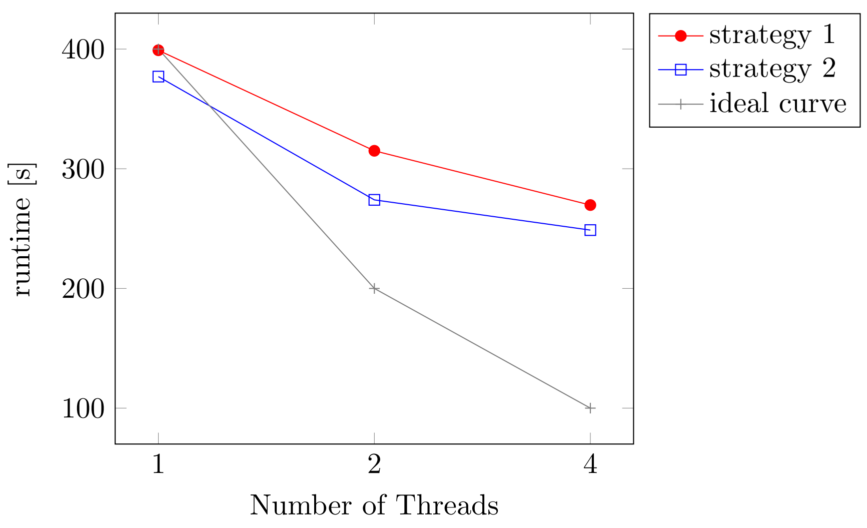

Figure 4.2.: Comparison of the two parallel strategies. With an increasing number of threads, the runtime decreases but the scaling is not optimal.

runtime is even less if we increase the used threads to four.

One reason for the suboptimal performance may be related to the iterator. In the following, we use N as the number of particles and t as the number of threads. In strategy one, we increase the iterator in total (on all threads) $N \cdot t$ times if we use t threads. This means, that even if we do not need to call the getter functions in every iteration, we need to increment the iterator every time, which is one of the more time-consuming operations. In contrast, in strategy two we only increase the iterator $\frac{N \cdot (t-1)}{2}$ times in total which is roughly half of the necessary increments of strategy one. The reason is, that we increment the iterator $i \cdot \frac{N}{t}$ times for thread T_i, and for all t threads, this can be expressed by: $\sum_{i=0}^{t-1} \frac{N}{t} \cdot i = \frac{N}{t} \cdot \sum_{i=0}^{t-1} i = \frac{N \cdot (t-1)}{2}$. However, threads processing particles that are 'further back' in the container, need more time than threads processing particles that are stored 'at the beginning' of the container. Another reason why the implementation may be slow is that with an increasing number of threads, the number of cache misses also increases. All the hypotheses still have to be verified in the future.

4.3.2. Comparison of the Runtime of the Position Calculation using Different Getter Methods

As described in section 3.1.2, we can get the position of a particle in Julia with two different getters functions. In the previous subsection, we found out that the memory access e.g., with getters, takes the most amount of time in the function, thus we are interested in optimizations. The runtime and the profiling results are discussed in the following.

For instance, if we like to get the position of a particle, we can call a getter function, which either returns a `jlcxx::ArrayRef` object or a single floating point value. For the last option, we have to call the getter three times, once for each coordinate. We run a simulation with one million particles and one thousand iterations for both getter variants. The result is that the implementation, using `jlcxx::ArrayRef` as return type, takes on average approx. 969.5 seconds[7]. In contrast, the other variant only takes approx. 377 seconds on average. This means that we can simply reduce the time of the most time-consuming operation by 61.1% by changing the implementation of the getter function. Figure 4.3 shows the profiling results of both implementations and again we can see, that the getters consume the most time. Especially the implementation using `jlcxx::ArrayRef` spends more than 80% only accessing data. Interestingly, we can also see that setting the value of the `oldForce` variable only takes about $1-2\%$ of the total runtime and we use the `jlcxx::ArrayRef` type as well. It may be interesting to find out the reason why the execution time of getters and setters is so different.

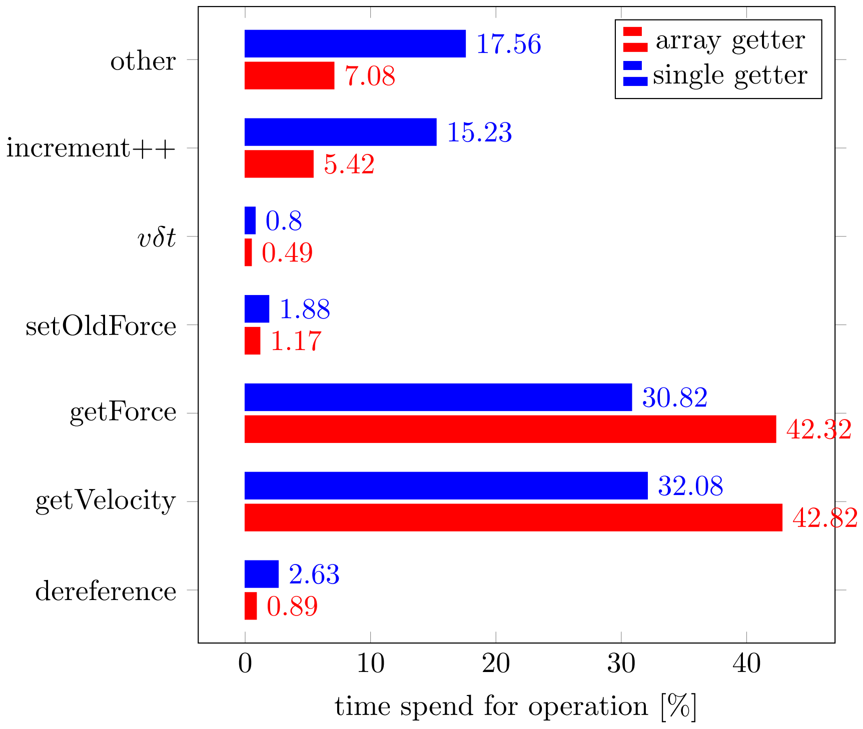

Figure 4.3.: Profiling results of the position calculation function of both getter versions. The array getter spends more time in the getter function than the single value getter.

A closer look into the implementation of the `MoleculeJ` class (listing 3.3) may provide an answer to the higher computing time of the 'array getter' implementation. As we cannot return a `std::array` from the getter, firstly, we create an object of type `jlcxx::ArrayRef`, and secondly, we need to copy this object, as we do not return a reference to `jlcxx::ArrayRef`. Both steps take some additional time and may be the reason for a slower 'array getter' performance.

[7]For this average value, we only consider two timings

4.3.3. Comparing the Runtime of the Position Calculation with a Pure Julia Implementation

In section 3.2.5, we described the implementation of a Julia struct representing a particle in Julia. We like to see how long the `updatePositions` function takes if we use pure Julia operations, without the use of wrapper code and further, we inspect the behavior if we use multiple threads.

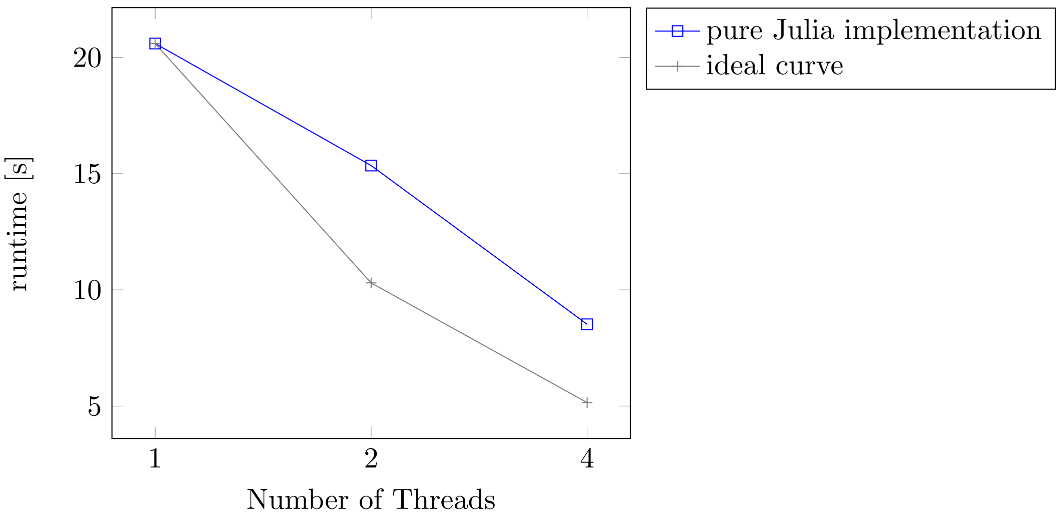

Figure 4.4.: Runtime results of the position calculation in pure Julia.

We used the same simulation scenario as in the previous subsection and we can see interesting results in figure 4.4. In comparison to the hybrid approach of the `updatePositions` function, we only need about 5% of the time, and the scaling with increasing threads is also decent up to four threads. The step from one to two threads only decreases the runtime by 25.5%, but from two to four threads, the runtime decreases by 44.5%. Even if the results look promising, we need to be careful with their interpretation. Firstly, the particles are simply stored inside a vector and not inside a complex data structure like the AutoPas container. This means, many operations, like accessing the correct particle, are not necessary for the Julia implementation, hence we have good performance. Secondly, we use the type `SVector` for our array types, which seems to be fast, but according to [11], `SVector` is allocated on the stack and this may limit its scalability if a very big number of particles is used. Thirdly, this implementation is intended to show that multi-threading is possible in Julia and is a reasonable as well as good choice, as this implementation compares very well with the 'ideal' curve.

4.4. Analysis of the Runtime with an Increasing Number of Particles

We already discussed the runtime of the `updatePositions` function in the previous section, in the following we have a look at how the Julia simulator compares to the C++ implementation md-flexible.

Prior to looking into the results, we like to state one hypothesis, which is based on our intuition. If we use functionality from the AutoPas library and compare the runtime from the Julia call of this functionality to the runtime of the md-flexible call, the runtime of the Julia simulator may not be lower than the one of the md-flexible simulator. As the code is exactly the same for both simulators, we expect that the Julia call may need slightly more time, as we need to consider the additional function calls and operations from the CxxWrapper e.g., getting the pointer to the C++ function or the conversion between Julia and C++ types, as already stated in section 2.5.3.

In figure 4.5, we can see the total execution time as well as the time needed for the force calculation of both simulators for an increasing number of particles. Not surprisingly, the execution time increases with the number of particles, however, it is surprising, that the Julia simulator and force calculation are quicker than the C++ implementation. Furthermore, our mentioned hypothesis does not apply to these timings. Due to the fact, that the force calculation comprises the largest part of the simulation in both simulators, we need to have a deeper look at the Julia force calculation.

One reason, why the force calculation is lower than expected, is that the force calculation is not executed properly e.g., we only calculate the forces between a subset of particles. To verify this, we simulated ten particles for one hundred iterations with both simulators and inspected the end configuration of the particles. After a comparison of the values, we can only determine that the values are all equal for both simulators. Additionally, we simulated one hundred thousand particles with both simulators and randomly selected particles with the same molecule ID and verified their attribute values. Again, we can see that the values are the same. Obviously, we need to apply further research and tests on the equality of both results to make a more robust statement.

Another reason may be that the memory access is advantageous for the force calculation, compared to the md-flexible implementation. However, this is only speculation and is not further investigated in this thesis.

The promising results of a fast Julia simulator are not reflected by the results for the position and velocity calculation in figure 4.6. We can see that the Julia implementation takes approx. 550% longer than the md-flexible implementation. In addition to time measurements, the used Julia package TimerOutputs, discussed in section 4.2, also provides information about memory allocations. We can see, that the allocated memory increases for an increasing number of particles. For example, in the positions calculation function, 207 KiB of data is allocated over one hundred iterations for one thousand particles and

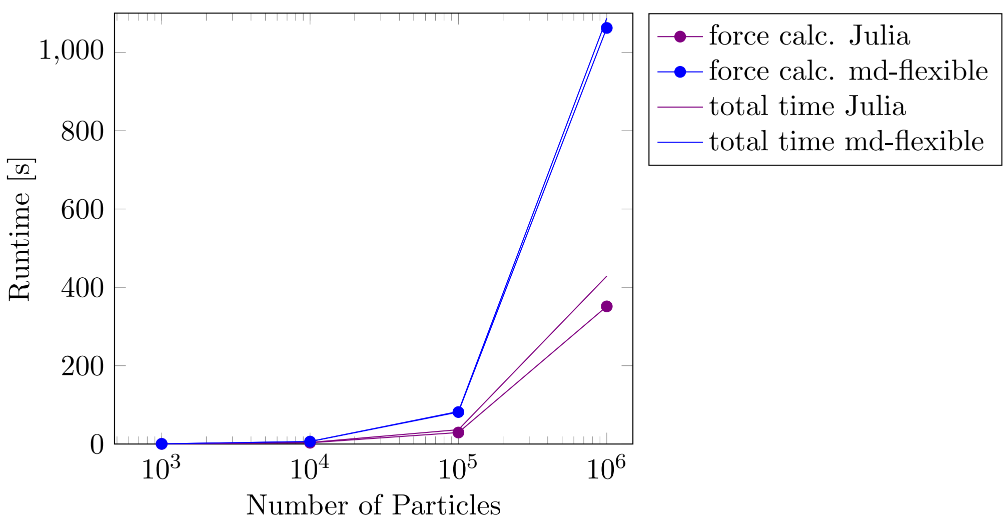

Figure 4.5.: Comparison of the total runtime and the force calculation of the Julia and md-flexible simulator for an increasing number of particles (logarithmic scale).

15.3 MiB for one hundred thousand particles. Increasing allocations can make sense for an increasing number of particles, however, the allocations also increase with increasing iterations. This may be a reason for the higher runtime of the Julia calculations. Already discussed in section 4.3, we spend the most amount of time to get the values of the velocity and the force if we calculate the position and the md-flexible simulator may need less time for the memory access, hence can perform better.

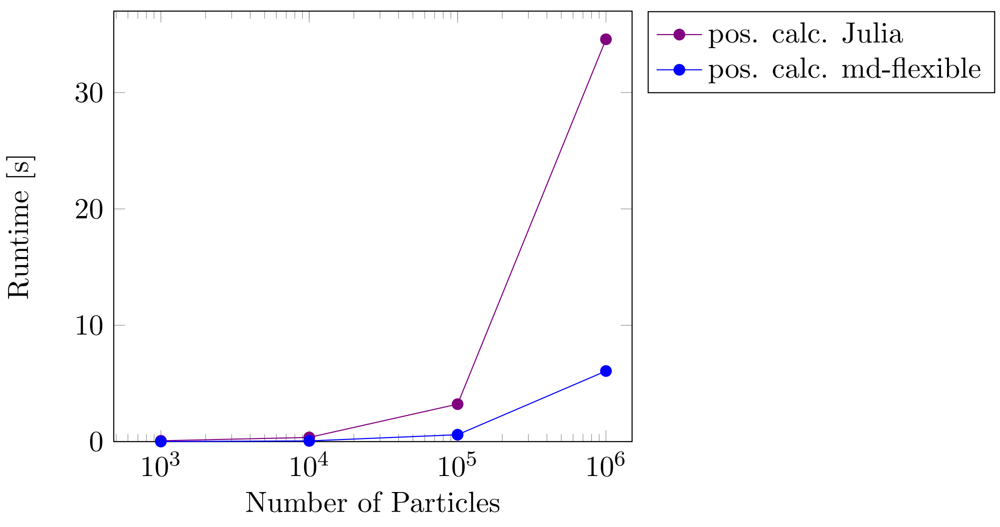

Figure 4.6.: Timings of the position calculations of both simulators, with an increasing number of particles (logarithmic scale).

The last part of this section discusses the function `addParticles`, which adds particles to the AutoPas container. In contrast to the discussed examples, we call `addParticle` without any other AutoPas or Julia functions (besides looping over all particles in Julia) and it is rather related to memory access than the calculation. Adding one hundred thousand particles to the AutoPas container takes 0.029s and adding one million particles 0.29s if we use the C++ simulator. In the case of using the Julia simulator, adding one hundred thousand particles takes 0.072s and one million particles 0.74s. We can see, that the Julia implementation takes about 2.5 times longer for this operation. This may be a good rule of thumb for the overhead of calling simple and non-computationally expensive C++ functions from Julia.

4.5. Scaling with Increasing Thread Count

In this section, we analyze how the Julia simulator compares to the md-flexible implementation if we use multiple threads. Again, we used one million particles and one hundred iterations for the test runs.

Figure 4.7 shows the total time of the simulation and the time spent in the force calculation for both simulators. As already discussed in the previous section, the Julia implementation is faster than the md-flexible implementation if we use one thread. With an increasing number of threads, the total simulation times get closer, until the md-flexible simulator is faster after using more than eight threads. However, we can see that the force calculation of the Julia simulator performs better than the Julia simulator as a whole, as the time still decreases with more than eight threads. We do not know the exact reason, why the force calculation is faster if called from Julia, but we can see that this advantageous optimization is less efficient if more threads are used. Hence, the runtime of both force calculations tends towards the same limit. As we see in the following, the limiting factors for the Julia simulator are indeed the calculation for the velocity and the position if multiple threads are used, at least for this specific scenario.

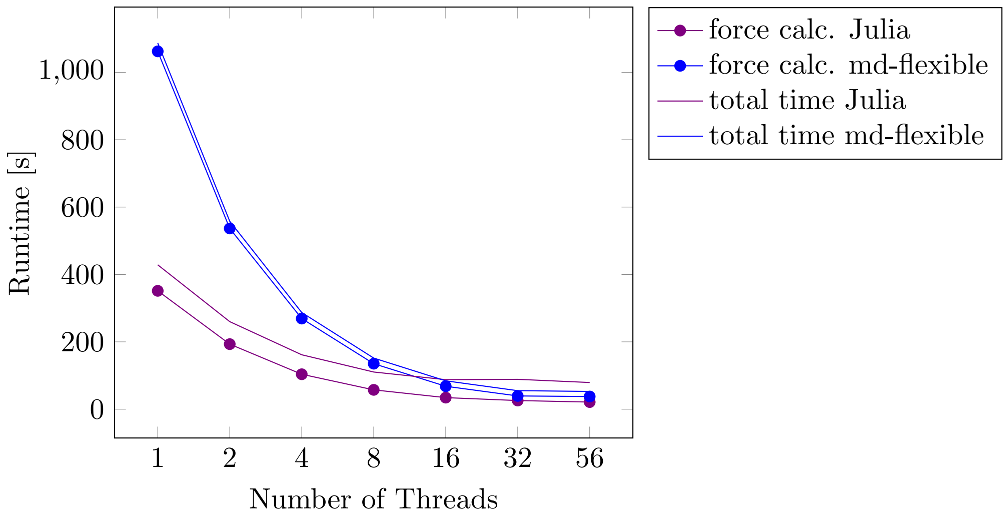

Figure 4.7.: Simulation of one million particles for one hundred iterations of the md-flexible and Julia simulator. The total time and the time of the force calculation are displayed.

Figure 4.8 shows the runtime with an increasing number of threads of the position and velocity calculation of the Julia simulator as well as of the md-flexible simulator, side by side. We can see, that the runtime of the Julia implementation decreases pretty well until 16 threads. If more threads are used, the runtime increases massively. In contrast, both calculations in the md-flexible simulator show a decrease in runtime even if we use more threads. Further, we can see, that the md-flexible implementation creates a nearly ideal scaling up to 56 threads. This seems to be the reason, why the Julia simulator gets slower than the md-flexible simulator for high numbers of threads, as the position and velocity calculations in Julia are actually longer than the force calculation.

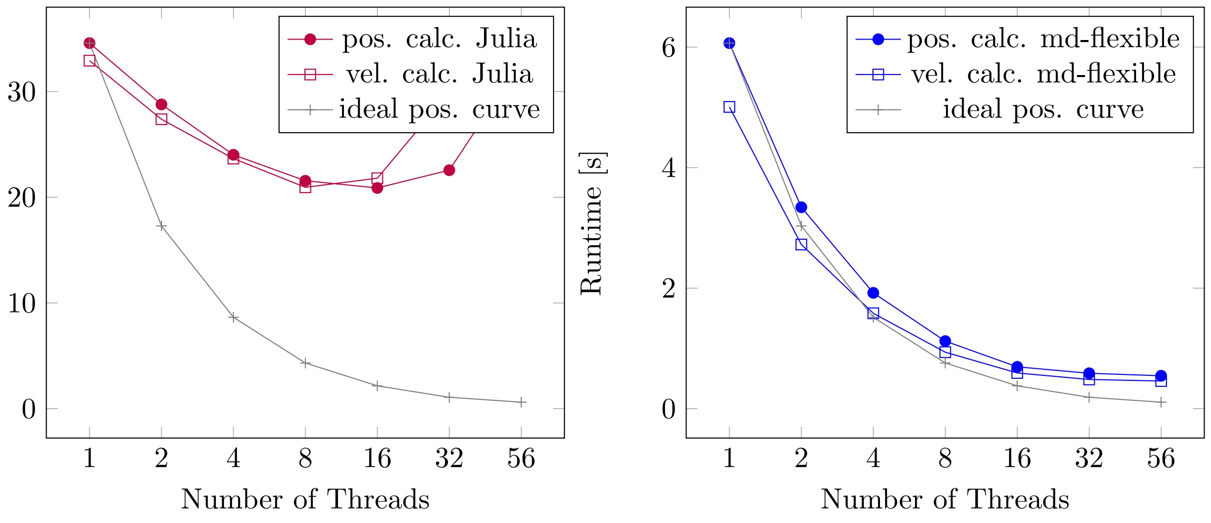

Figure 4.8.: Runtime results of the position and velocity calculation of both simulators. Comparing the position calculations against the ideal curve.

We further notice that the position calculation of the Julia simulator takes approx. six times as long as the same calculation in md-flexible, which is best illustrated in figure 4.9. In section 4.3, we already discussed some reasons for the suboptimal scaling, but with this figure, we like to point out the massive difference compared to md-flexible.

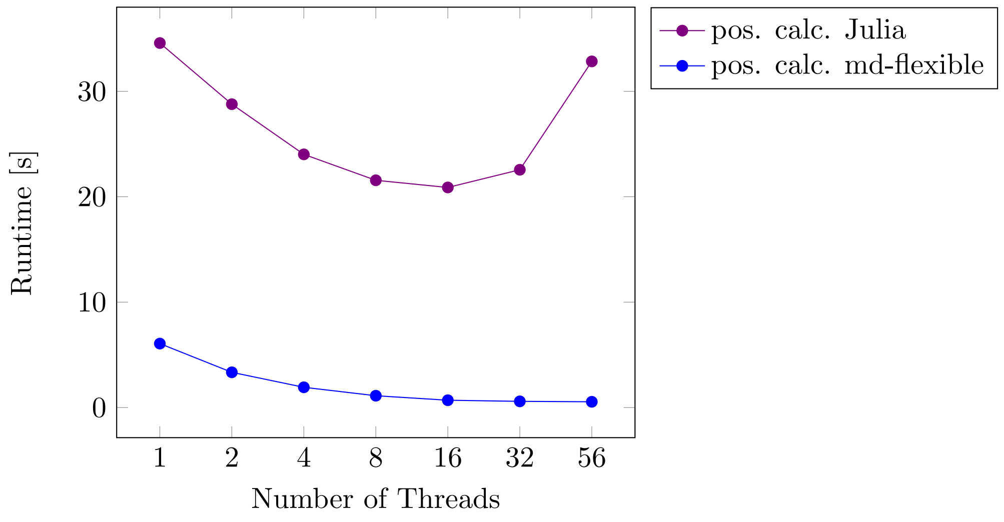

Figure 4.9.: Comparing the runtime for the position calculation of the Julia and md-flexible simulator in one graph. The C++ implementation is very quick compared to the Julia implementation.

5. Future Work

This thesis gives an introduction to the utilization of the AutoPas library from a Julia simulator. Some parts of the code may need to be refined and extended in the future, as well as verified with more sophisticated approaches.

As discussed in 3.5, the distributed memory parallelization is not working properly, and if the hybrid approach of using Julia and AutoPas together is further investigated, parallelization with MPI is necessary to implement. Additionally, it is interesting to test if the MPI.jl has indeed no overhead compared to the C implementation as stated in [4]. To run a program with MPI.jl we need to look for a way to distribute the code to different compute nodes. If the Distributed.jl package is used the ClusterManagers.jl[1] package may be used to distribute the program to several nodes, but a usage with MPI.jl still needs to be checked.

Additionally, the periodic boundary conditions need to be implemented properly. As explained in 3.5, the `regionIterator` function is not working as expected and this behaviour needs to be explainable. Solving this problem may help to implement MPI parallelization as well.

Up to this point, we only wrapped a few functions and types of the AutoPas class and repository. For more complex simulations we may need more functions that have to be wrapped too. Another aspect is to think of different ways to handle types not mapped by CxxWrap e.g., the `std::array` type.

If we like to use the simulator for real applications, we should find a way to reduce the runtimes of the position and velocity calculation, as they are extremely high compared to the md-flexible implementation. Furthermore, the scaling of the calculation with multiple threads needs to be improved to provide a more suitable simulator. A different implementation for the `MoleculeJ` class may be helpful as we have seen that the single value getters are more performant than the `ArrayRef` getter. We may consider returning a reference to the object or using only single values instead of arrays.

The potentially most interesting point to focus on is to find out why the force calculation is faster in Julia. As md-flexible and the Julia simulator use the exact same function, this result is unexpected. The first step can be to ensure the correctness of the force calculation i.e., are all particles processed and are the results for the simulated particles equal to the results of a simulation with md-flexible. As the second step, it is interesting to see if the result can be reproduced with other container data structures like Verlet Lists[2]. If the previous steps reinforce the result of a faster Julia force calculation, the last step is to find

[1]`https://github.com/JuliaParallel/ClusterManagers.jl`
[2]Further information to Verlet Lists can be found in [6]

out the real reason for this phenomenon.

Lastly, we may like to test the simulator with different inputs e.g., more particles or tuning strategies. In this thesis, we analyze a rather easy simulated scenario. More difficult as well as diverse scenarios may give more meaningful results in terms of performance and application to real-world examples.

6. Summary

In the last chapter, we like to summarize the approach to using Julia for a molecular dynamics simulator and AutoPas as a backend.

Using just one line of C++ code to wrap a function or a type with the CxxWrap.jl package turned out not to work in all cases. For types of the C++ standard library that are not supported by CxxWrap or are templated functions inside a templated class, we needed to think of a workaround. Even if quite a lot of types and functions need to be wrapped to fully work with the AutoPas library, we managed to wrap all of them.

Benefitting from Julia's advantages, we are able to use the REPL and can interactively initialize input parameters and run the desired simulation.

The first parallelization method we applied in the thesis is multi-threading. Even if we can easily parallelize a for loop in Julia, we needed to adapt our implementation, because we originally used a while loop instead of a for loop, due to the usage of the AutoPas iterator. Hence, we implemented two versions for the parallelization of the position and velocity calculation and tested their performance in chapter 4.

The plan was to apply distributed memory parallelization with the package MPI.jl. For this strategy, we implemented a simple domain decomposition, which can be easily extended for all three dimensions. Additionally, we can (de-) serialize particles as well as send and receive them from other processors. The code was adapted to the boundary conditions for distributed execution. However, we observed that some particles disappeared if MPI and the periodic boundary conditions is applied and consequently, we produced wrong results.

We discussed the usability of Julia and of the CxxWrap.jl package. In summary, it turned out that it is indeed a suitable alternative to use Julia together with C++ in terms of productivity, especially if we try to find bugs or want to test small simulations.

The runtime results are unexpected and surprising. The force calculation of the Julia implementation is faster than the one of the C++ implementation, even if both simulators use the exact same C++ code. On the contrary, the results of the position and velocity calculations in Julia are slower than the one of md-flexible. After profiling the Julia code, we realized that the memory access of the getter functions needs the most time of all operations. However, using a pure Julia implementation of the position calculation showed that pure Julia code can definitely compete with the C++ implementation, at least in this very simplified test case.

The runtime behaviour of the Julia simulator with an increasing number of threads shows that the decrease in runtime is mostly dependent on good scaling of the force calculation.

The position and velocity calculations do scale medium well using multi-threading, but only up to 16 threads.

Finally, we pointed out what has to be improved so that the Julia simulator can be used in scientific research.

Part III.

Appendix

A. Input File for the Simulation

```
container                       :  [LinkedCells]
selector-strategy               :  Fastest-Absolute-Value
data-layout                     :  [AoS]
traversal                       :  [lc_c08]
tuning-strategy                 :  full-Search
functor                         :  Lennard-Jones
newton3                         :  [disabled]
cutoff                          :  3
box-min                         :  [0, 0, 0]
box-max                         :  [113.5, 113.5, 12.7]
cell-size                       :  [1]
deltaT                          :  0.0001
iterations                      :  100
boundary-type                   : [none,none,none]
globalForce                     :  [0,0,-0.001]
Objects:
  CubeGrid:
    0:
      particles-per-dimension   :  [100, 100, 10]
      particle-spacing          :  1.12
      bottomLeftCorner          :  [1.0, 1.0, 1.0]
      velocity                  :  [0.1, 0.1, 0.1]
      particle-type             :  0
      particle-epsilon          :  1
      particle-sigma            :  1
      particle-mass             :  1
no-flops                        :  false
no-end-config                   :  true
log-level                       :  info
subdivide-dimension             :  [true, false, false]
```

Listing A.1: Input file used for the performance test of md-flexible.

List of Figures

Listings

Bibliography

[1] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL `http://arxiv.org/abs/1209.5145`.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014. URL `http://arxiv.org/abs/1411.1607`.

[3] Hans-Joachim Bungartz, Stefan Zimmer, Martin Buchholz, and Dirk Pflüger. *Modellbildung und Simulation: eine anwendungsorientierte Einführung*. Springer-Verlag, 2nd edition, 2013.

[4] Simon Byrne, Lucas C Wilcox, and Valentin Churavy. Mpi.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021.

[5] C. Cords. jluna: A modern Julia Wrapper for C++. `https://www.clemens-cords.com/jluna`, 2022. URL `https://www.github.com/clemapfel/jluna`. v1.0.0.

[6] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2021. doi: 10.1016/j.cpc.2021.108262.

[7] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical Simulation in Molecular Dynamics*. Springer Berlin Heidelberg New York, 1st edition, 2007.

[8] Clemens Heitzinger. *Algorithms with JULIA: Optimization, Machine Learning, and Differential Equations Using the JULIA Language*, page 10. Springer International Publishing, Cham, 2022. ISBN 978-3-031-16560-3. doi: 10.1007/978-3-031-16560-3_1. URL `https://doi.org/10.1007/978-3-031-16560-3_1`.

[9] Bart Janssens. Cxxwrap.jl, 2020. URL `https://github.com/JuliaInterop/CxxWrap.jl`.

[10] Bart Janssens. CXXWRAP.JL Julia and C++: a technical overview of CxxWrap.jl, 2020. URL `https://barche.github.io/juliacon2020-cxxwrap-talk/#/2/1`.

[11] Anton Karlsson. The julia programming language for embedded high-performance signal processing in radar systems. Master's thesis, 2020.

[12] Pavel Kharitenko. Coupling Julia-based simulations via preCICE. B.S. thesis, 2021.

[13] Stephanie M Linker, Christian Schellhaas, Anna S Kamenik, Mac M Veldhuizen, Franz Waibl, Hans-Jorg Roth, Marianne Fouché, Stephane Rodde, and Sereina Riniker. Lessons for Oral Bioavailability: How Conformationally Flexible Cyclic Peptides Enter and Cross Lipid Membranes. *Journal of Medicinal Chemistry*, 66(4):2773–2788, 2023.

[14] Antonello Lobianco and Antonello Lobianco. Interfacing julia with other languages. *Julia Quick Syntax Reference: A Pocket Guide for Data Science Programming*, pages 91–111, 2019.

[15] Leandro Martínez. CellListmap. jl: Efficient and customizable cell list implementation for calculation of pairwise particle properties within a cutoff. *Computer Physics Communications*, 279:108452, 2022.

[16] Adriaan Nieß. Simulation of control systems in IEEE 802.1 Qbv networks using OMNeT++. B.S. thesis, 2019.

[17] David Poole, Jorge L Galvez Vallejo, and Mark S Gordon. A New Kid on the Block: Application of Julia to Hartree–Fock Calculations. *Journal of Chemical Theory and Computation*, 16(8):5006–5013, 2020.

[18] Neeraja Ramanan. Jit through the ages: Evolution of just-in-time compilation from theoretical performance improvements to smartphone runtime and browser optimizations. *Web1. Cs. Columbia. Edu.*

[19] Honghui Shang, Li Shen, Yi Fan, Zhiqian Xu, Chu Guo, Jie Liu, Wenhao Zhou, Huan Ma, Rongfen Lin, Yuling Yang, et al. Large-Scale Simulation of Quantum Computational Chemistry on a New Sunway Supercomputer. *arXiv preprint arXiv:2207.03711*, 2022.

[20] Marcel Stanitzki and Jan Strube. Performance of Julia for high energy physics analyses. *Computing and Software for Big Science*, 5:1–11, 2021.

[21] Martina Stork. *Molekulardynamik-Simulationen von amyloidogenen Proteinen in Lösung: Stabilitätsuntersuchungen und Weiterentwicklung einer Kontinuumsmethode*. PhD thesis, LMU, 2007.