Parallelizing a particle simulation on shared memory architecture using OpenMP

Thursday 2nd January, 2025 - 15:02

Vogel Benjamin
University of Luxembourg
Email: benjamin.vogel.001@student.uni.lu

This report has been produced under the supervision of:
Dr. Krishnasamy Ezhilmathi
University of Luxembourg
Email: ezhilmathi.krishnasamy@uni.lu

***Abstract*—As computational problems become more complex, program performance becomes more important. Hence, tools are needed to solve larger-scale and more complex problems. This reports scientific section aims to explain how parallel computing and notably multi-threading can improve software performance. Afterward, the technical half of the program covers the production of a particle simulation that aims to simulate the movement of particles inside electromagnetic fields and serving as an example for the parallelization of a program. Moreover, the use of parallelization through OpenMP is also a topic of this paper and is covered in both sections. All in all, multi-threading and in turn parallel computing, provide many benefits such as program speedup, energy efficiency, more efficient hardware utilization, and the capacity to solve increasingly more complex problems. In conclusion, parallel computing is and will remain an integral domain in computer science as without it, many computational problems would not be realistically solvable.**

1. Introduction

Computer programming is an incredible tool to solve many diverse problems. Initially, most programs had a simple structure where one instruction was executed after the other in a sequential manner. Computer programs have had a major impact, notably in science and engineering. However, nowadays, the scale of research has increased immensely, causing the requirements for computational power also to increase. Especially data analysis and computer simulations are two examples where the need for high-performance computing is apparent. To elaborate, the domain of high-performance computing (hpc) concerns itself with the aggregation of computing power to solve large computational problems. Because high-performance computing is only going to become more important in the future, this report provides an entry into hpc by covering the basics of parallel computing as well as an introduction to multi-threading on multiple processors.

The scientific section of this paper seeks to answer how parallel programming can improve software performance by providing information within the domain of parallel computing and multithreading. This includes computer architecture, program scalability, and the notions of processes and threads. Furthermore, the report seeks to cover the topic of multi-threading with OpenMP and provides examples of the information covered. The technical section covers the production of a particle simulation that is parallelized using the OpenMP API. In short, the technical half of this paper covers the information needed to produce such a simulation as well as the thought processes used during the parallelization of the program. Finally, a critical evaluation of the program is done, where the performance of the program is analyzed and the output evaluated.

2. Project description

2.1. Domains

2.1.1. Scientific. The scientific part of this report revolves around the domains of parallel computing, multithreading, and computer architecture.

2.1.2. Technical. The technical part of this report covers the subject of computer simulations, C-programming as well as the use of OpenMP in order to parallelize a program on shared memory architecture.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. How can multithreading be used to improve software performance? This project aims to answer that question. Accordingly, the scientific deliverable explains the concept of multithreading. Prior to this explanation, in view of the fact that this document is designed for an audience with no prior knowledge in the domain of parallel computing, there is an introduction to the domain of parallel computing. Furthermore, the

deliverable presents the computer architecture needed in order to run a parallel program. The document from Barney Frederick [1] as well as the book 'Parallel Programming in OpenMP' [4] provide information on the terminology within the domain of parallel computing and provide a better understanding of what is happening on a hardware level when parallelizing a program.

After these fundamentals are covered, the deliverable goes in-depth on how multithreading works and what to look out for when turning serial code into parallel code, for example, the scope of data or the partitioning of problems. This information is gathered from 'Introduction to parallel programming with OpenMP' [8] and the book 'Using OpenMP: Portable Shared Memory Parallel Programming' [4]. The reason is that both give good explanations of how a program performs its tasks using multiple threads and of the obstacles that can be encountered when parallelizing a program.

At this point, the reader should have a good grasp of the topic of the deliverable. Before giving a final answer to the main question, one or multiple forms of measurement for software performance are established. Finally, from the research performed, an answer to the main question will be given.

2.2.2. Technical deliverables. The technical deliverable of this project is a simulation that simulates the movement of particles in a uniform electric and magnetic field. The project aims to take a serial implementation of the simulation source code, written in C, and parallelize it using the OpenMP API. Subsequently, Shared memory architecture can run these simulations, which improves performance when simulating a large number of particles.

A user should be able to use the software in order to determine how variables like the mass and charge of a particle affect its trajectory within these fields.

The user decides on the particles to be simulated before the simulation is launched. The information of these particles, i.e. mass, charge, size, etc. should be stored within a text file allowing for easy storage. Afterward, the user can set the attributes of the respective fields within a terminal. Finally, the simulation takes the information given by the user and simulates the movement of the particles, taking into account the electric and magnetic fields as well as the forces between the particles. 'Electricity and Magnetism' [13] and 'lennard jones potential' [17] are used for the calculation of the forces as they contain the necessary theory and formulas for these computations

The computation of the simulation is made through a vector calculation and physics management library produced over the course of the project. These libraries contain all the necessary tools in order to properly compute the trajectory of the particles along with their interactions with the environment.

3. Pre-requisites

3.1. Scientific pre-requisites

Before starting this project, it is required to have a basis in linear algebra and physics. Concerning linear algebra, knowledge of basic vector calculation and transformation is recommended. Additionally, because the technical half of the project deals with the movement of particles it is required to have foundational knowledge in the domain of mechanics, notably Newton's laws of motion as well as how to utilize a potential energy function to find the force applied on a system.

3.2. Technical pre-requisites

Regarding the technical knowledge that should have been acquired before starting this project, it is recommended to already have experience coding in languages such as C or C++, as simple programming concepts will not be covered in this project. Furthermore, a basic understanding of how to use a command line is also endorsed.

4. Parallel Computing and Multi-threading

4.1. Requirements

This Scientific Deliverable revolves around the domain of parallel computing. Notably, this deliverable covers the subject of multithreading on shared memory architecture. The main question this document will answer is thus, "how can multithreading improve software performance?".

The scientific deliverables' goal is to provide the necessary information to understand why parallel computing is a powerful tool and some basics on multi-threading with OpenMP.

First, as an introduction to parallel computing, the report describes what a serial program is and explains how parallel computing aims to solve the issues that characterize serial programs. Moreover, understanding the architecture of a parallel computer is integral to grasping how instructions can be run in parallel. For that reason, the report covers the von Neumann computer architecture, Flynn's taxonomy as well as shared memory architecture.

The scientific deliverable also covers the subject of scalability and speedup, in order to provide the reader with some information to quantify the effect parallelization has on a program. In this section, the report not only explains weak and strong scaling but also covers the subject of Overhead and its causes. Furthermore, this paper covers general techniques to parallelize computational problems

and provides examples.

After describing the notion of processes as well as threads, the report explains what multithreading is and how it is used in concurrent and parallel programming. Along with this explanation, the report also provides an introduction to OpenMP and its threading model for shared memory architecture.

Finally, the scientific deliverable proposes measurements for performance in a program, which are used to answer the initial question posed for this project precisely. The report concludes with a collection of problems solved through the use of multithreading/parallel computing. However, this document also covers the obstacles that can be encountered when trying to implement multithreading and what to avoid when parallelizing a program.

4.2. Design

In order to produce the scientific deliverable, various sources have been gathered and used. This section describes the sources and their usage within this report.

First and foremost, 'Introduction to parallel computing' [1] by Blaise Barney contains diverse kinds of information on most topics essential to parallel computing. To name a few subjects: the notion of parallel computing, computer architecture, parallel program designs, and parallel programming models. This document provides a great starting point on this report's subject, however in order to provide more detailed and precise information, other sources are required.

Thus, 'First Draft of a Report on the EDVAC' [16] by John Von Neumann as well as 'Some computer organizations and their effectiveness' [5] by Micheal J. Flynn provide more context and information on computer architecture. Notably von Neumanns computer architecture as well as Flynns Taxonomy.

Regarding Scalability and program speedup, the paper 'Scaling theory and machine abstractions' [9] provides definitions and explanations on weak and strong scaling and their differences.

Moreover, the book 'operating system concepts' [14] by JL Peterson and A. Silberschatz contains fundamental information on processes and threads, as well as multi-threading.

Finally, 'An Introduction to Parallel Programming with OpenMP' [8] by Alina Kiessling as well as 'Parallel programming in OpenMP' [3] provide the most important information on parallel computing using multi-threading with openMP.

4.3. Production

4.3.1. Parallel Computing. In a serial program, instructions are executed one after the other by a single processor such that, at most, one instruction can be executed at any moment in time. This, however, is very inefficient, especially when a computational problem is of considerable complexity.

Parallel computing should be considered a tool allowing programs to perform multiple tasks simultaneously. This enables parallel programs to solve more complex problems, compared to serial programs. In essence, parallel computing concerns itself with the use of multiple computing resources to solve computational problems. Noteworthy is the possibility to scale hardware horizontally, i.e. the addition of computing machines, or vertically, i.e. upgrading or adding computing resources to currently existing hardware. However, not all hardware can run parallel code. [1].

4.3.2. Parallel computer architecture. The von Neumann architecture is a model for a digital computer. In summary, according to the von Neumann model, a computer contains [16]:

- A device capable of performing arithmetic and logical operations
- Another device that is in charge of receiving, understanding, and delegating instructions.
- Memory that stores data and instructions
- Input and Output capabilities

In more general terms, the arithmetic-logic unit, along with the control unit, can be assimilated into a CPU. [1]

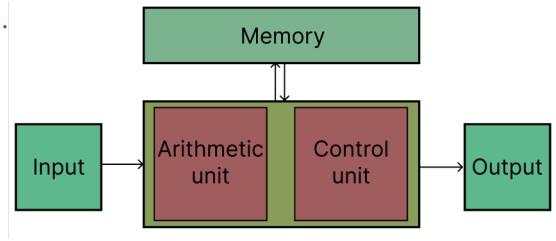


Figure 1: Von Neumann model diagram

Furthermore, a modern computer contains different kinds of memory. Each memory subdivision differs in size and access time from a processor. The difference in access time usually is caused by the distance between the CPU and the memory.

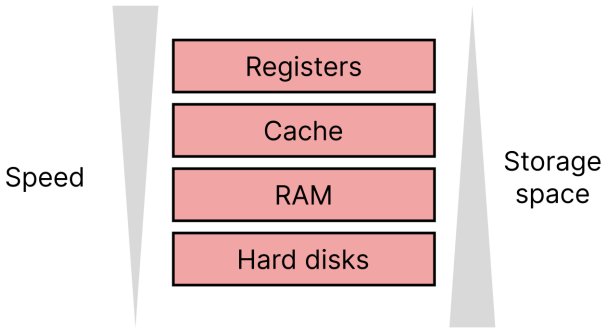


Figure 2: Memory Size/Speed Diagram

Flynn's taxonomy is a classification for parallel computer architecture.

Flynn's taxonomy distinguishes 4 different classifications [5]:

Single instruction stream, single data stream (SISD) represents a serial computer. To elaborate, a processing unit receives instructions from a single stream and thus only performs one instruction at any moment. Similarly, the computer also receives data from only one stream.

Single instruction stream, Multiple data stream (SIMD) differ from SISD as with SIMD architecture a computer possesses multiple processing units. These processing units perform the same task at any given moment as they receive instructions from a common instruction stream. Since, however, these processing units obtain data from separate instruction streams, they are capable of performing their instruction on multiple data instances. Such architecture can be used, for example, to perform vector arithmetic faster.

Multiple instruction stream, single data stream (MISD) is a very rare computer architecture, where multiple processing units perform different instructions on a single stream of data. There are barely any examples of when such an architecture is appropriate. One possibility would be trying to decrypt a message using multiple different algorithms.

Multiple instruction stream, Multiple data stream (MIMD) is the architecture of most modern computers. It is characterized by possessing multiple processing units each having its personal instruction and data stream.

Furthermore, 2 types of memory architecture for a parallel computer are distinguished in parallel computing: shared memory architecture and distributed memory architecture. For the purpose of this project, this section only covers the shared memory model.

The fundamental property of shared memory architecture is a global memory space that is accessible by all processors. Accordingly, modification in the memory caused by a processor is visible to all other processors. What's more, this architecture is divided again into two categories: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA).

Uniform Memory Access is characterized by the presence of multiple processors, that all have equal access to memory as well as equal memory access times. On the other hand, Non-Uniform Memory Access architecture differs as each processor has a different memory access time depending on how far the memory is located from the processor. In other words, some processors can access their local memory faster than other remote memory. Thus we consider the memory access time non-uniform.

Important to know is that UMA architecture maintains cache coherency a lot easier than NUMA architecture because each processor in NUMA architecture has their own local memory. To clarify, an architecture is considered cache coherent if all processors are informed when a location in shared memory is updated.

4.3.3. Scalability and Speedup. In the context of parallel computing, Software scalability refers to how efficiently the software can be sped up through parallelization. Speedup is defined as the ratio between the execution speed of a program using a single processor, t(1), and multiple processors, t(n):

$$Speedup = \frac{t(1)}{t(N)}$$

Most software fall under one of two categories: strong scaling software and weak scaling software.

On the one hand, if an application scales strongly, scaling is achieved through the increase in processors used for a fixed problem. In essence, the increase in processors is supposed to decrease the necessary time to solve a fixed problem. The theoretical speedup for strongly scaling applications can be measured in such a manner:

$$S_{strong}(p, N) = \frac{1}{p_{serial} + \frac{p_{parallel}}{N}}$$

also known as **Amdahl's law** [9], where S is the theoretical speedup achieved by a strongly scaling application when utilizing N processors to execute a program p. Furthermore p_{serial} and $p_{parallel}$ denote non-parallelizable and parallelizable code-region of the program, with $p_{serial}+p_{parallel} = 1$. Worth mentioning is that when N tends to infinity, the Speedup is bounded by the non-parallelizable section of the program:

$$\lim_{N\to\infty} S_{strong}(p, N) = \lim_{N\to\infty} \frac{1}{p_{serial} + \frac{p_{parallel}}{N}} = \frac{1}{p_{serial}}$$

On the other hand, weakly scaling software is characterized by a fixed problem size per processor used. To elaborate, weak scaling applications want to ensure a constant execution time when confronted with increasingly large problems through the use of more computational resources. Accordingly, speed up for weakly scaling applications is measured through **Gustafson's law** [9]:

$$S_{weak}(p, N) = p_{serial} + p_{parallel} * N$$

Where S is the speedup for weakly scaling applications and p and N are the same as for the previous formula. This formula is not bounded compared to Amdahl's law, as this formula takes into account, that usually, the quantity of resources allocated to a problem is proportional to the problem size.

Important to note is that these formulas only give a theoretical Speedup. There are many factors that must be taken into account when considering parallelizing a section of code. Performance loss arises from various factors: [1]

Task start-up/termination time is the time necessary to initialize the environment needed for the execution of parallel tasks. This time varies depending on the complexity of the system used, the number of processors involved, and the communication infrastructure. Therefore, small tasks or processes usually should not be parallelized as the task start-up and termination time are larger than the time saved through parallel computations.

Synchronization refers to the coordination between multiple processes. Synchronization is important to ensure that execution is correct and consistent. However, ensuring synchronization often entails having a unit of execution idle for some period of time.

Load balancing occurs when parallel processes do not possess equal workload. In other words, a program takes longer to execute, if a unit of execution takes a lot longer to finish than other units of execution, as this also results in periods of time where computing resources are idle. Thus even workload distribution is essential to guarantee maximal Speedup

Data communications can also be a source of performance loss. Excessive or unnecessary communication between processes can greatly hinder execution. Especially, because processes have to be synchronized in order to communicate without any risk of unpredictable behavior.

In the domain of parallel computing, the additional time and resources required for parallel computations are called **Overhead**.

4.3.4. Parallelization of computational problems. One of the first actions that should be taken when parallelizing a program is to analyze the problem that is to be solved and decompose it into distinct tasks that can be executed simultaneously.

There are two fundamental ways a computational problem can be partitioned [1]. The first is called Domain Decomposition. Essentially, the data set of a problem is partitioned and distributed among the processing units.
A good example is the application of a function on all elements within a large array. Instead of iterating through the entire array and applying the function, one by one, this process can be sped up by dividing the array into multiple regions that are handled by different parallel tasks.

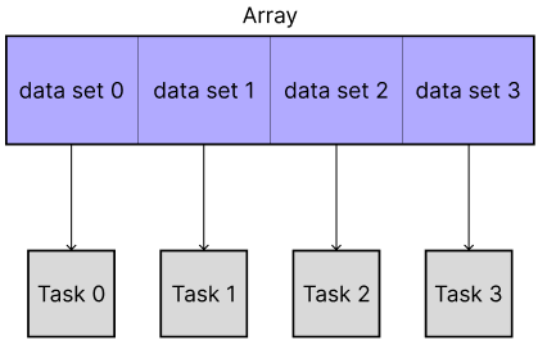


Figure 3: Array Partitioning example [1]

The second partitioning method for computational problems is called Functional Decomposition. Compared to domain decomposition, functional decomposition partitions the problem based on the instruction set of the problem instead of its data set. For instance, running two different applications on a computer can become very slow if only one processing unit is dedicated to them. However, performance should improve considerably by allowing two different processing units to process one application each.

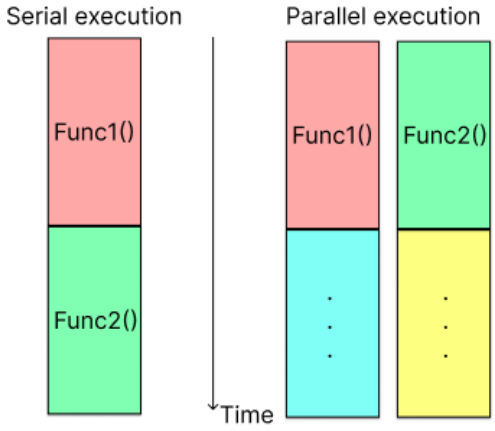


Figure 4: 2 separate executions

Once a problem has been properly partitioned, there are various parallel programming models that can be used to accomplish the parallelization. This document focuses on the thread model.

4.3.5. Processes. Before covering the concept of threads, processes are explained. Fundamentally, a process refers to a program executing on an OS. A process constitutes a lot of information: [14]

- **Text section** refers to the program code or the instructions of the program.
- **Data section** contains all global and static variables
- **Stack** contains the temporary data of a program. For instance, function parameters or local variables.
- **Heap** contains the dynamically allocated memory during program runtime.

A great way of thinking about programs and processes is that a program is a passive entity. The process is an instance of an execution of a program. [14] Important to note is that a process usually does not have access to another process' information. Moreover, processes can be executed by a single thread or multiple threads.

4.3.6. Threads. A thread can be seen as a unit of execution that can be executed independently within a process. Each thread has its own stack to store information, however, the process heap is accessible from all threads of the process. The reader might ask what this abstraction provides. [14]

4.3.7. Multithreading and OpenMP fundamentals. An issue that gets encountered when running a program serially on a single thread is that for certain instructions the processing unit has a period where it does not perform any operations. For example, when data is fetched from a hard drive or a server request has been made. In order to not waste computing resources, the OS allows a thread to execute during the time another thread is idly waiting. The threads run concurrently.

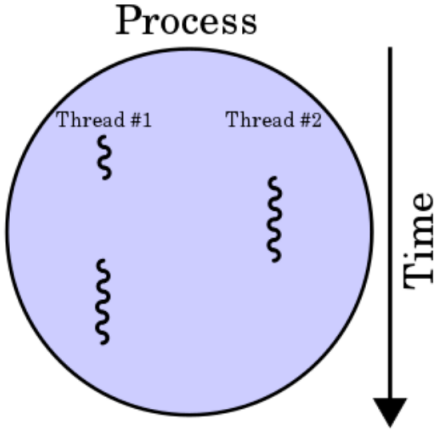


Figure 5: When thread 1 is idle, thread 2 executes [2]

Multi-threading on a single processing unit allows for concurrent computing and thus optimizing CPU usage. The OS usually organizes the threads using a thread scheduler.

What's more, threads can be used in parallel computing. Instead of scheduling threads on a single processing unit, threads can be distributed among different processing units.

This report orients itself on the OpenMP programming model to explain how multithreading can be used to parallelize a program. OpenMP is an application programming interface that supports shared memory multithreading in C, C++, and FORTRAN

In essence, OpenMP allows the programmer to utilize its directives to provide the compiler with further context on how the program is supposed to be parallelized. The basic structure of an OpenMP directive, also known as a 'pragmatic', in C is:

```
#pragma omp directive_name [clauses]
    //code section
```

OpenMP clauses provide further information on how the compiler is supposed to compile the code. Notably, the data distribution among threads and attributes such as the number of threads created for a given task and the distribution of work among these threads are factors that can be controlled by clauses. Additionally, the union of the directive, clauses, and code section is known as a construct. In short, the directive and its clauses allow the compiler to parallelize code

OpenMP parallelizes a program using the Fork-Join model. Every program starts off as a single thread, called the master thread. The master thread executes serially until it encounters a parallel region. These parallel regions are designated by the *parallel* directive. When encountering the parallel region, a team of parallel threads is created, which then proceed to execute their instructions. Once all these threads have accomplished their tasks, the parallel threads join together, and the master thread continues serial execution again. [4]

When creating parallel threads, the data available to these threads have to be managed using clauses. A thread has access to 'private' and 'public' memory. When a variable is public, all threads are able to access its memory address. However, having a memory address accessible to all threads can cause major problems when multiple threads try to read and write a memory address at the same time because data races can occur, which causes unpredictable behaviour within the program. In order to solve such a problem, the threads either must communicate with each other in order to avoid simultaneous access or the data should be set private. When data is set private, every thread will have its own copy of the variable within its personal memory. [3]

For example, the distribution of a problem domain is

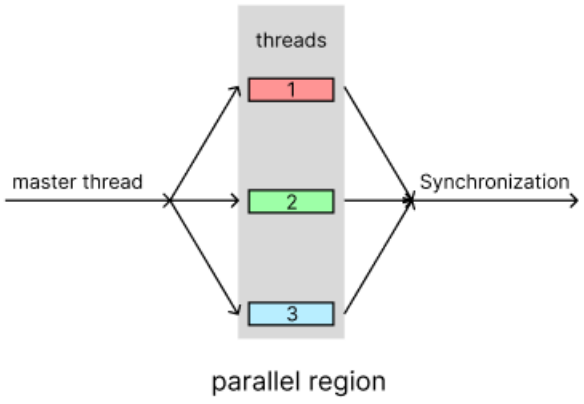


Figure 6: Fork-Join Model

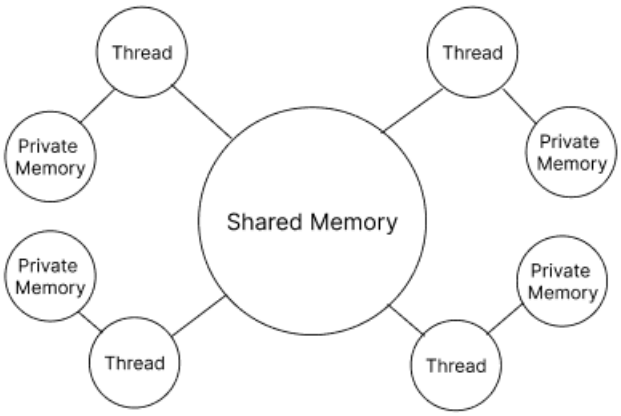


Figure 7: OpenMP threads memory diagram

done through the use of their assigned thread ID. Take the previous example of array partitioning. Here the array itself is public data. The thread makes use of their ID to guarantee that they only access their part of the array:

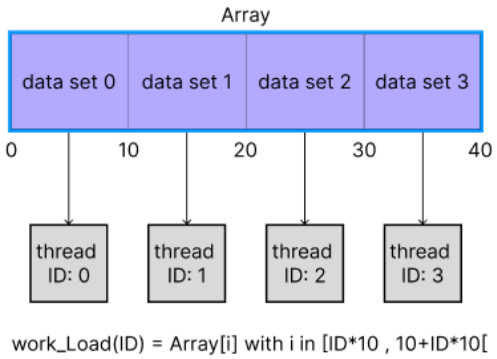


Figure 8: Array Partitioning with threads

As already stated, data races occur when a result depends on a certain sequence or timing of threads, for example when threads read or write shared data. Data races cause undefined behavior and are thus a topic that programmers have to take into account when parallelizing their code. Synchronizing the threads is the most straightforward way of avoiding data races. Barriers and Locks are two tools that can be used to synchronize threads. When a thread encounters a barrier during execution, it halts execution until all threads have reached that barrier and then resumes execution. In OpenMP 'pragma omp barrier' defines a barrier in a parallel region. Conveniently, most OpenMP directives have an implied barrier at the end of their code section. All worker threads halt execution when arriving at the end of the construct until all threads have finished execution. This synchronizes all threads and allows the seamless transition back to serial execution or a new parallel task. On the other hand, a lock is a mechanism that makes sure that only one thread at a time can perform instructions in a 'locked' region; this assures that multiple threads do not access or write the same memory addresses. OpenMP uses 'critical regions'. A thread can only execute a critical region if it possesses the lock, of which there is only one. When execution is done, the lock gets passed to the next thread waiting to execute a critical region. [3]

4.3.8. Measuring performance. When considering the performance of a program there are many points to take into account. For the purpose of this report, the most interesting topics are the execution Speed for a computational problem, which has already been covered in section 4.3.3., efficient usage of resources, a subject that is becoming increasingly important in regard to the environment, and the scale of the computational problems that can be solved by a program.

4.3.9. Multi-threading and performance. The advantages of multi-threading for program performance can be generalized to the advantages of parallel computing. First of all, as seen in section 4.4.3., parallel computing provides the capability to either increase the execution speed of programs or at least provide the capability to increase the scale of a problem without causing a major negative impact on execution speed; this becomes important when the number of tasks becomes too large to be handled by a single computing resource. [1] However, the increase in software complexity has to be taken into account. Parallel computing is quite complex and takes time to properly integrate which can result in increased development time or harder-to-debug code. These are issues that OpenMP thankfully handles quite gracefully. Next, parallel computing can allow for more energy-efficient computing. [10]. This is because the power used by a CPU increase at least quadratically with the increase of the frequency of a CPU. Thus, it is usually more energy efficient to use two CPUs with a frequency of 25MHz instead of one 50MHz CPU. Moreover, the use of multiple computing resources allows the reuse of material. Finally, parallel computing provides the capacity to solve large and complex problems. Things like physics simulations or huge systems of networks need the capacity to execute many

instructions simultaneously, which is only possible to do efficiently through the use of multiple computing resources.

All in all, parallel computing and, through correlation, multi-threading provide a lot in terms of performance and solve many issues of serial computing. However, it should be understood that parallel computing is not a tool that is necessary for all problems and should only be used when necessary. Nonetheless, for the cases where parallel computing is appropriate, it is an incredibly powerful tool.

4.4. Assessment

This section evaluates the research done for the production, based on the requirements defined at the beginning of the scientific section of this report.

Concerning the topics covered, the report covers most information that is necessary to grasp the domain of parallel computing and the concept of multi-threading on shared memory architecture. Notably, the report covers the basics of computer architecture as well as subjects within the domain of computer architecture concerning parallel computing such as Flynn's taxonomy and uniform and non-uniform memory architecture. Furthermore, the notions of scalability and overhead have also been covered. The report also provides examples and diagrams to better illustrate the information given.

However, the definition of threads given within the report is quite vague and mainly serves to better understand how OpenMP functions. A more in-depth look into threads could have been informative.

Nonetheless, this paper managed to give an answer to the question "How can multi-threading improve software performance?", and provides enough context to understand the topic of the question. Thus, all in all, the scientific deliverable is considered a success.

5. Parallelizing a Particle Simulation

5.1. Requirements

The technical deliverable consists of a particle Simulation. This program is to be run on shared memory architecture because the software is parallelized using OpenMP, an application programming interface for parallel programming.

This simulation software will be used by users who want to inspect how particles behave in uniform electric and magnetic fields. For its functional requirements, it is expected that the program is capable of computing the movement of the particles while taking into account the forces acting upon them. Therefore the simulation requires a solid library of functions for vector computations as well as a library possessing functions that determine the following forces:

- **The electrostatic force**, acting between two charged particles
- **The Van der Wals force**, acting between any two particles
- **The Electric force**, applied on charged particles when inside an electric field
- **The Magnetic force**, applied on charged particles when inside a magnetic field.

Furthermore, the simulation should be able to notice and handle collisions.

Because there are many calculations that have to be done during such a simulation, a serial version of the program cannot simulate a large number of particles in a timely manner. Therefore, the program is parallelized using OpenMP for faster execution and the possibility to perform larger-scale simulations.

The software has a simple menu within the terminal when launched. This menu allows the user to either: start a simulation, display information on the particles that can be simulated, or quit the program. If the user chooses to perform a simulation, the user indicates the type and number of particles to simulate, then the initial conditions of these particles are set and then the simulation commences.

The information on the particles is stored in text files and is read by the program when retrieving the constants corresponding to these particles for the simulation.

5.2. Design

The design section for the technical deliverable explains the design choices made during the production of the simulation. In short, the following paragraphs contain explanations on the use of the OpenMP API for program parallelization, the use of hpc infrastructure provided by the University of Luxembourg [15], along with Intel Vtune and Arm Forge for performance analysis. Finally, the use of ParaView for the evaluation and visualization of the simulation is presented.

5.2.1. OpenMP. OpenMP is an application programming interface used for multi-core processing on shared memory architecture. For this technical deliverable, OpenMP is used in order to parallelize the simulation. This program makes use of 4 OpenMP directives: parallel, for, critical and atomic.

The *parallel* directive defines a parallel region that is going to be executed by multiple threads. All threads execute until they arrive at the end of the parallel region at which they encounter an implied barrier. Once all threads are done executing, the master thread takes over again, and serial computing proceeds until another parallel region is encountered. The number of threads created can be

determined either through the numthreads clause or by setting the value of threads to be used in the terminal with the command: 'export OMP_NUM_THREADS=x'. [12]

```
// 5 threads are created and each prints its ID
#pragma parallel numthreads(5)
  printf("Thread_ID: %d\n", omp_get_thread_num());
```

The *for* directive is used within a parallel region to divide the work done by a for-loop among threads. Additionally, the for directive can also be combined with the parallel directive, in order to have a one-liner.

```
int i;
int boundary = 10;
#pragma parallel for private(i) public(boundary)
  for (i = 0, i < boundary, i++){
    printf("Iteration num: %d\n", i);
  }
```

The *public* and *private* clauses are used with the *parallel* and *for* directives to define whether each thread should have its own instance of a variable or if the variable can be accessed by all threads collectively.

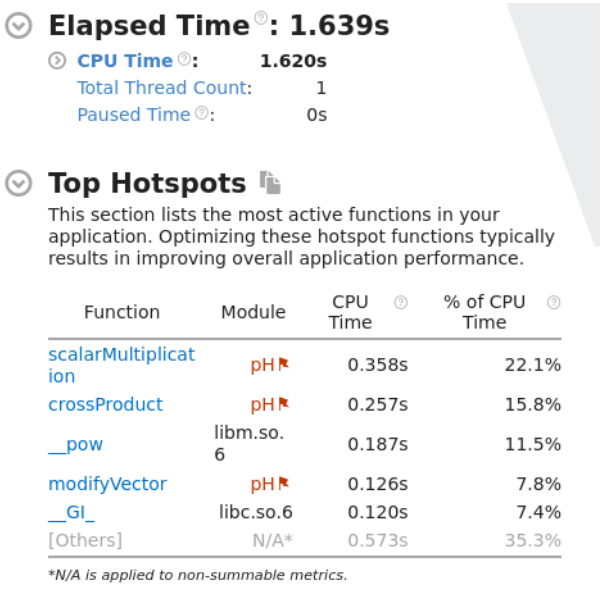
In general, the for loops iterator value is set to private, whereas the boundary for that iterator stays public. The *for* directive possesses other useful clauses that can be used to improve performance, such as *schedule*. With the schedule clause, it is possible to define how the work should be distributed among the threads. For this report, we distinguish static and dynamic scheduling. With static scheduling, the work distributed among all threads is fixed. This means that no matter what, all threads perform the same amount of iterations. However, with dynamic scheduling, the work is distributed at runtime. In other words, threads receive a chunk of work to do, and once they are finished with their chunk of work, they receive a new chunk. This is beneficial when the amount of time spent executing an iteration can vary, as this improves load balancing.

Finally, the *critical* directive defines a critical region. When a thread starts executing instructions within a critical region, no other thread is able to execute instructions within another critical region. Once the thread within the critical region is done executing, the next thread starts the instructions within its critical region. This is used when multiple threads threaten to manipulate an address in memory at the same time, which can cause data races and thus undefined behaviour. When the critical region only consists of one operation, the atomic directive can b used. Atomic has a similar behaviour to critical, however, the overhead caused by it is smaller, as the nature of the operation is specified. [12]

5.2.2. Hpc-infrastructure. The main objective of the technical deliverable is to parallelize a computer program and determine the improvement in performance. However, parallelizing a program only provides benefits if the user has the corresponding hardware. Therefore, testing is done on hpc-infrastructure provided by the University of Luxembourg [15].

The final program is run on the Iris (Intel) and Aion (AMD) supercomputers. These supercomputers are partitioned into compute nodes each possessing a certain amount of computational resources. These resources are not always available. In such a situation, it is possible to schedule a job in advance. Batch scripts are utilized to provide the job scheduler with the necessary information to execute the desired job, such as the planned amount of resources to be used and the commands/programs to be executed.

5.2.3. Performance evaluation with Vtune and Arm Forge. Vtune and Arm Forge are profiling software that provide the tools to analyze how a program performs. Specifically, Vtune can be used to analyze how efficiently the CPU and Memory are used, it can provide insight into what sections of the program execution take the longest. Arm Forge provides information on how much time is spent accessing memory, executing instructions, synchronizing threads, and more. In other words; they can locate bottlenecks in execution. [7] Vtune and Arm Forge are used to evaluate the overall performance of the simulation and comment on possible improvements.



Elapsed Time: 1.639s

CPU Time: 1.620s
Total Thread Count: 1
Paused Time: 0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
scalarMultiplication	pH	0.358s	22.1%
crossProduct	pH	0.257s	15.8%
__pow	libm.so.6	0.187s	11.5%
modifyVector	pH	0.126s	7.8%
__GI_	libc.so.6	0.120s	7.4%
[Others]	N/A*	0.573s	35.3%

*N/A is applied to non-summable metrics.

Figure 9: Vtune indicating what functions took up the most amount of time (serial execution)

5.2.4. Visualization with paraview. A method of visualization is necessary in order to evaluate the output of the simulation. This project makes use of the open-source visualization application, called Paraview, to visualize the data sets produced by the simulation. To elaborate, the simulation creates for every time step a CSV file that

contains the position of all particles simulated. These files can be passed to Paraview to then visualize the data.

In order to visualize the data, Paraview first converts the table of contents within the CSV file to points in space. Because these points are not well visible, glyphs of a sphere is placed where the points reside:

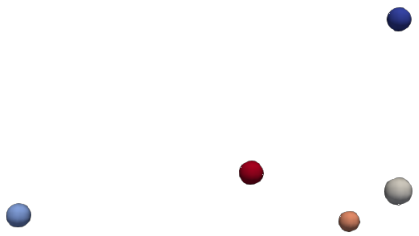


Figure 10: Visualization of particles in paraview

Important to note is that Paraview allows for the parallelization of data set processing. However, as the scale of the information provided by the simulation is small, this feature is not necessary for usage.

5.3. Production

This section describes the process of the production of the simulation.
The main steps of the production consist of:

1) Development of a module for vector operations
2) Development of a module for the computation of forces applied on the particles
3) Continuous collision testing
4) Implementation into a program
5) Parallelization
6) Performance evaluation
7) Visualization

5.3.1. Vector module. The computation of the position, velocity, and acceleration of the particles requires creating a module that allows for the manipulation of vectors. Vectors can be abstracted as a one-dimensional array. Thankfully, as the simulation is fixed to 3 dimensions, we can fix the array size to 3, as all array attributes of a particle possess 3 components.

As C is a statically typed language, the type of values contained within the vectors has to be determined. Because the program is a simulation where precision is required, the type of the values is set to *double*.

Functions within the module include the creation of vectors, vector addition and subtraction, vector norm calculation, vector normalization, euclidean dot-product, cross-product, and angle calculation between two vectors. Note that the operator functions on the vectors manipulate the first vector given as an argument and do not create an entirely new vector containing the result

Furthermore, a function capable of solving augmented matrices is needed. The function takes in an augmented matrix of a system as well as the number of equations and variables, then returns an array containing a solution to the system. In short, the function performs the gaus-jordan algorithm on the matrix. Afterward, the function tries to read the result from the row-reduced matrix. Determining whether a solution exists.

5.3.2. Physics Module. The physics module is in charge of calculating the movement of the particles as well as the forces applied to them. Moreover, the module also defines the data attributed to a particle as well as the electric and magnetic field these particles reside in.

First, an instance of a particle has to contain information on its weight, charge, radius, etc. These attributes are stored in a structure called *particleInfo* which itself is stored in a structure *particle* alongside the position, velocity, and acceleration of the particle. Code: 8.1.1

Next, the electric and magnetic fields are identical as both only have their respective field vector assigned to them. Note that for this simulation, these fields are considered uniform and thus only require a single vector to define them. Nonetheless, the fields receive their own structure in order to separate their usage within functions. Code: 8.1.2

In order to correctly simulate the movement of a particle, the formulas for the variation in position, velocity, and acceleration have to be determined. The first thing to determine is acceleration. Therefore, Newtons second law of motion [6] is used which states:

$$\sum \vec{F}_{ext} = m * \vec{a} \tag{1}$$

Thus, the acceleration of a particle can be determined by dividing by m on both sides:

$$\frac{\sum \vec{F}_{ext}}{m} = \vec{a} \tag{2}$$

Where $\vec{F}_{ext}$ are forces applied to the particle, m being the mass of the particle and $\vec{a}$ the resulting acceleration.

Because acceleration is the change in velocity over time, the formula to determine the velocity of a particle can be found by integrating (2) over time:

$$\vec{v}(t) = \int \frac{\sum \vec{F}_{ext}}{m}\, dt \tag{3}$$

and thus:

$$\vec{v}(t) = \frac{\sum \vec{F}_{ext}}{m} t + \vec{v}_0 \tag{4}$$

Where v_0 is the initial velocity of the particle.

Subsequently, as velocity is the change in position over time, (4) can be integrated again over time to yield the formula determining the position of a particle:

$$\vec{p}(t) = \int \frac{\sum \vec{F}_{ext}}{m} t + \vec{v}_0 \, dt \tag{5}$$

resulting in

$$\vec{p}(t) = \frac{\sum \vec{F}_{ext}}{2m} t^2 + \vec{v}_0 t + \vec{p}_0 \tag{6}$$

Where p_0 is the initial position of the particle

These formulas allow the program to calculate the movement of the particle based on its initial state and the acceleration applied to it. However, these formulas suppose that acceleration is constant, which is not the case. In order to be as precise as possible, the simulation calculates the change in these values in short and regular time steps.

On every time increment Δt the following formulas are used to determine the acceleration, velocity, and position of the particles:

$$\begin{cases} \vec{a} = \frac{\sum \vec{F}_{ext}}{m} \\ \vec{v}(t + \Delta t) = \vec{a}\Delta t + \vec{v}(t) \\ \vec{p}(t + \Delta t) = \frac{\vec{a}}{2}\Delta t^2 + \vec{v}(t)\Delta t + \vec{p}(t) \end{cases} \tag{7}$$

The resulting function can be found in the appendix: 8.1.3.

The next step consists of determining the external forces that are applied to the particles. These forces are determined every time Step, summed together, and used for the calculation of the acceleration of particles:

$$\sum \vec{F}_{ext} = \vec{F}_{electric} + \vec{F}_{magnetic} + \vec{F}_{el.st.} + \vec{F}_{VDW} \tag{8}$$

The electric force $\vec{F}_{electric}$ and magnetic force $\vec{F}_{magnetic}$ are forces applied to charged particles passing through electric and/or magnetic fields respectively. On the other hand, the electrostatic Force $\vec{F}_{el.st.}$ is a force that occurs between two charged particles. These forces can be determined through the following formulas given in the Book 'Electricity and Magnetism' [13]:

$$\begin{cases} \vec{F}_{electric} = q\vec{E} \\ \vec{F}_{magnetic} = q(\vec{v} \times \vec{B}) \\ \vec{F}_{el.st.} = k_e \frac{q_1 q_2}{r^2} \vec{u} \end{cases} \tag{9}$$

Where q is the charge of the particle, $\vec{v}$ denotes the velocity vector of the particle, $\vec{E}$ is the electric field, $\vec{B}$ the magnetic field, and where k_e is the coulomb constant. Code: 8.1.4

Finally, the Van der Wals Force is a force that occurs between any two particles. The Van der Wals Force is substantially weaker than the electrostatic force and can be neglected when working with charged systems. However, the Van der Wals Force is used to determine how non-charged particles interact with each other. This simulation utilizes the Lennard-Jones Potential [17] in order to approximate the forces acting between any two particles:

$$V_{LJ}(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6})] \tag{10}$$

r being the distance between the particles, epsilon and sigma being the depth of the potential well of the particle and the distance at which the particle-particle potential energy is 0. This formula can be used to determine the magnitude of the force:

$$F_{V.D.W.} = -\frac{\mathrm{d}V_{LJ}}{\mathrm{d}r} = 48\epsilon[(\frac{\sigma^{12}}{r^{13}}) - 0.5\frac{\sigma^6}{r^7})] \tag{11}$$

Multiplying the magnitude of the force by $\vec{u}$, being a unit vector with the same direction as the line created by the center of the two particles results in the force vector. Code: 8.1.5

These forces are separated into two functions. One function calculates the 'particle-field' forces, whereas the other function calculates the 'particle-particles' forces.

The *applyFieldForce* function takes a particle, the electric and magnetic field as arguments, and calculates the forces that are applied on the particle by the fields and the subsequent acceleration caused by these forces. Code: 8.1.6

The *applyInterForce* functions take two particles, determine whether these particles are charged, and calculate the force the second particle exerts on the first particle along with the resulting acceleration. Code: 8.1.7

5.3.3. Continuous collisions testing. Initially, the program used discrete collision detection in order to determine collisions between two particles. This type of collision detection tests collisions at fixed time intervals.

```
READ position1, position2, radius1, radius2
COMPUTE distance between particles
COMPUTE the sum of the radii
IF distance <= sum of radii THEN
  CALL collisionHandler with (particle1, particle 2)
ENDIF
RETURN
```

However, relying on this type of collision detection alone is not appropriate with small, high-speed objects like particles. The reason is that the particles can move so fast that between time intervals a collision gets unnoticed. The program misses thus the collision and the particles move as if there was nothing in their way.
Therefore, the program utilizes continuous collision testing. This type of collision detection calculates in advance the

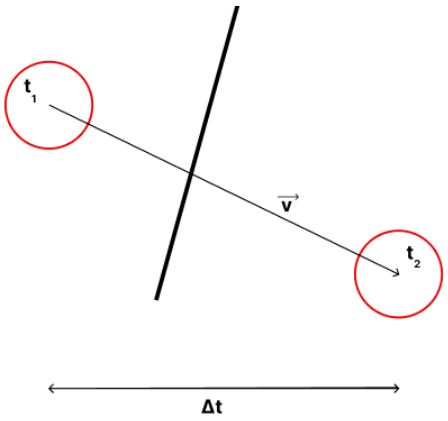


Figure 11: Discrete collision dilemma

movement of particles and determines whether there will be a collision during the next time interval. In this program, the function tasked with determining the collisions also returns the exact time the collision takes place. This value is then passed to a collision handling function that then computes the proper behaviour.

Important to note is that for the collision detection, the program assumes that the collisions are elastic, meaning that there is no loss in energy on collision, only two particles can collide at a time and the particles that are going to be simulated have equal mass.

The function *checkPrioriCollision* takes two particles p1 and p2 as arguments. The function then creates a 2-dimensional array, that serves as an augmented matrix for the following system:

$$\begin{cases} \alpha v_x + \beta v'_x = c'_x - c_x \\ \alpha v_y + \beta v'_y = c'_y - c_y \\ \alpha v_z + \beta v'_z = c'_z - c_z \end{cases} \tag{12}$$

The function then tries to solve the augmented matrix for α and β. If the solution values are equal and between 0 and 1, then a collision will occur within the next time interval. Code: 8.1.8 If a collision is detected, the moment of collision is passed to the next function called *handleCollisions*. This function uses that value to move the particles up to their point of collision, determine the collision behaviour and then displace the particles for the remaining time of the time step according to the collision. Code: 8.1.9

Lastly, the *calcCollision* function is in charge of computing the collision. The simulation calculates the collision using the following algorithm [11]:

1) First, determine the angle between the two particles.
2) Then, calculate the projection of the particles' velocity vectors on the line connecting their centers.
3) Afterwards, calculate the projection of the particles' force vectors on the line perpendicular to the previously calculated vector.
4) Then, switch the colliders' force vectors from step (2).
5) Finally, compose the new vectors into a new velocity.

5.3.4. Implementation into a program. The main function requires a set of arguments in order to execute properly:

- file containing the information on the to-be simulated particle
- the number of particles that should be simulated
- the time period for how long the simulation should go
- the particles' initial speed
- the electricFields' vector
- the magneticFields' vector
- information on whether the simulation data should be printed into a file

When the program is executed in the terminal, it reads the arguments given to it. If the previously mentioned information has not been provided, the program stops its execution and prints an error message. Otherwise, the values are assigned to their respective variables.

If the arguments are valid, the first thing the program does is allocate memory for the array that stores the particles. Then, the memory for these particles, as well as the memory for the electric and magnetic fields, are allocated. Afterward, the particle attributes are initialized which are fetched from the text file that has been specified in the arguments. In order to determine the initial position of the particles and avoid their superposition, a sphere is defined. Through random number generation, the particles are initialized at random positions within that sphere.

Once the initialization has finished, an array is created of size equal to the number of simulated particles. This array initially only contains 0, which can, later on, be changed to 1, as it is used to determine whether the movement of a particle has already been computed.

Now, the execution enters the main loop, which performs the computations for the particles' movements. The loop consists of 3 steps. The first is the computation of the acceleration of all particles. Second is the collision checking and handling of all particles. The last step consists of the computation of the movement of remaining particles that have not been handled yet. Note that every 0.1 seconds of simulated time, a CSV file is created containing the information on all particles being simulated. Code: 8.1.10

5.3.5. Parallelization. In order to parallelize this simulation, the parallelizable regions of the program have to be determined. Optimally, the main loop of the simulation is parallelized as much as possible, as it is the

region of code in which the program resides longest. For this process, the report proposes a general algorithm:

First, determine regions in a program that seem parallelizable. Then, for each region found, analyze its sequence of instructions and determine whether any data races could occur. If afterward, no data races can occur, verify that this region can benefit from parallelization. Otherwise, determine whether the program benefits from parallelization in view of time loss through the use of barriers and locks. Once the regions of code benefiting from parallelization are determined, the data needed by the threads have to be categorized into private and public data. Finally, measure performance changes and adapt parallelization accordingly.

Applying this algorithm to the simulation, the entire main-loop region falls under inspection. The outer loop is already off-limits, as each iteration of this loop represents a time increment. Each increment is dependent on the results of the previous iterations and thus is not independent and consequently not parallelizable. However, within this loop, there are various regions that can be parallelized.

First and foremost, the computation of the particle acceleration caused by the electric and magnetic fields can be parallelized as each iteration operates on a distinct particle. Because all particles within the array are treated independently, and the information on the electric and magnetic fields are only ever read, the array, as well as the fields, can stay public.

```
#pragma omp parallel
    shared(particleArraySize,particleArray,elFi,magFi)
     #pragma omp for schedule(dynamic) private(i,j,k)
       for (j = 0; j < particleArraySize; ++j)
       {
         applyFieldAcceleration(particleArray[j],elFi,magFi);
       }
```

The next potential parallel region requires more thought. The next possible parallel region consists of a nested loop. Because the inner loop is dependent on the outer loop, it is out of the question to parallelize the outer loop. Nonetheless, the inner loop is parallelizable, if the program can guarantee that the memory address for the acceleration vector of *particleArray[j]* is not manipulated simultaneously by multiple threads. This is achieved by defining a critical region where the addition takes place.

```
for (j = 0; j < particleArraySize; ++j)
{
  // checks all the remaining possible collisions and
       performs particle displacement
  #pragma omp parallel
       shared(particleArraySize,particleArray,elFi,magF)
    #pragma omp for schedule(dynamic) private(i,j,k)
      for (k = j+1; k < particleArraySize; ++k)
      {
        applyInterAcceleration(particleArray[j],particleArray[k]);
      }
}
```

```
#pragma omp critical
  vectorAddition(p1->acceleration, temp);
```

The nested loop tasked with handling the collisions between particles does not seem parallelizable, however, by modifying the function *handleCollision* to take the index j as well as the list of handled particles as arguments, using a critical construct, the function can be refactored to allow for parallelization.

```
// checks all the remaining possible collisions and performs
     particle displacement
#pragma omp parallel
     shared(particleArraySize,particleArray,elFi,magF)
  #pragma omp for schedule(dynamic) private(i,j,k)
    for (k = j+1; k < particleArraySize; ++k)
    {
      if(handleCollision(particleArray[j],particleArray[k],j,handled))
      {
        handled[j] = 1;
        handled[k] = 1;
        break;
      }
    }
```

```
#pragma omp critical
    if (handleArray[pID] == 0 && !isnan(eval))
```

Finally, the last loop determining the movement of the particles can be parallelized with the same reasoning as why the first loop was parallelized.

Notice that within the main loop, there are 4 separate parallel regions. This is inefficient because each new parallel region causes overhead for the formation and synchronization of the team of threads. Thus, the loops are fused together in a manner, that threads are only instantiated once per iteration:

```
#pragma omp parallel
    shared(particleArraySize,particleArray,elFi,magFi)
     #pragma omp for schedule(dynamic) private(i,j,k)
       for (j = 0; j < particleArraySize; ++j)
       {
         applyFieldAcceleration(particleArray[j],elFi,magFi);

         for (k = j+1; k < particleArraySize; ++k)
         {
           applyInterAcceleration(particleArray[j],particleArray[k]);
         }
         for (k = j+1; k < particleArraySize; ++k)
         {
           if(handleCollision(particleArray[j],particleArray[k],j,handled))
           {
             handled[j] = 1;
             handled[k] = 1;
             break;
           }
         }
         if (handled[j] == 0)
         {
```

```
22              move(particleArray[j],currTimeStep);
23          }
24      }
```

Nonetheless, the largest issue is the two nested loops. Both loops are dependent on the outer loop, which usually is an indication that parallelization is problematic. In this case, these loops cause load imbalances. The reason is that the workload of a thread decreases with respect to the thread ID:

In order to solve this issue, the particle-particle argument

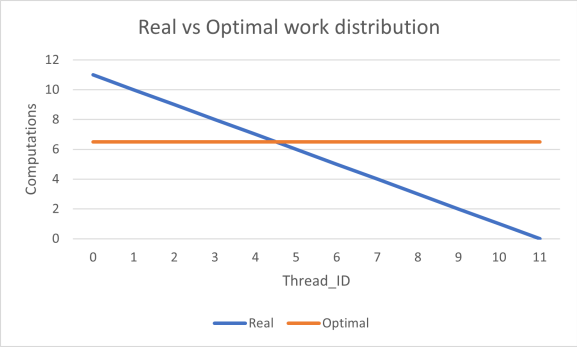


Figure 12: Iterations per Thread: Real vs Optimal

pairs are determined before entering the main loop and put into an array. That way, these pairs can be evenly partitioned between the different threads using their thread ID.

```
1 int id = omp_get_thread_num();
2 int total = omp_get_num_threads();
3 int *p;
4 for (k = chunk*id/total; k < chunk*(id+1)/total; ++k)
5 {
6   idx = particleIdxPartition[k]
7   applyInterAcceleration(particleArray[p[0]],particleArray[p[1]]);
8 }
```

Unfortunately, executing this program for 24 particles using 24 threads on the iris supercomputer takes over 5 minutes largely due to the time used for synchronization between threads in the form of the critical regions inside the *applyInterAcceleration* function. Thus, instead of finding a partition that allows the program to make the least amount of computations, which would be $\frac{n*(n-1)}{2}$ iterations per particle (n being the number of particles simulated), n^2 iterations are made. Because a thread only operates on the particle it is assigned to, no synchronization is needed. In fact, this modification decreased execution time to a bit more than a minute on the iris supercomputer. Code 8.1.11

5.3.6. Performance evaluation. VTune and Arm Forge are the two profiling tools used for the analysis of the simulations' performance. Not only do these tools give an analysis, but they also visualize the data they measured. For this analysis, the program simulates 24 particles utilizing 24 cores that can be used.

According to VTune, the biggest issues with the program are the time used to synchronize threads and the lack of

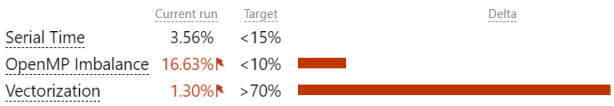


Figure 13: VTune: biggest performance issues

vectorization within the simulation. In order to solve the synchronization issue, either the work distribution among threads or the critical region within the *handleCollision* function has to be improved upon. On the other hand, the lack of vectorization within the program comes from the fact that most of the cores/threads are used to calculate the attributes and behaviour of each particle in parallel. Thus, there are no resources present for the acceleration of vector arithmetic.

Arm Forge provides further interesting information:

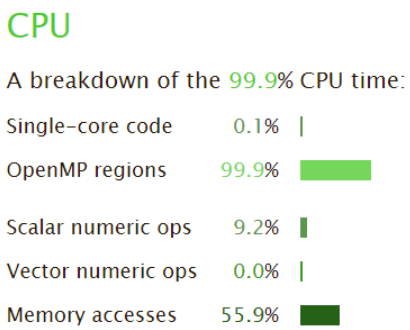


Figure 14: Breakdown of CPU time

From this information, it can be determined that the CPU spends most of its time accessing locations in memory. The reason could be that the program has to frequently fetch memory not within a cache, decreasing performance.

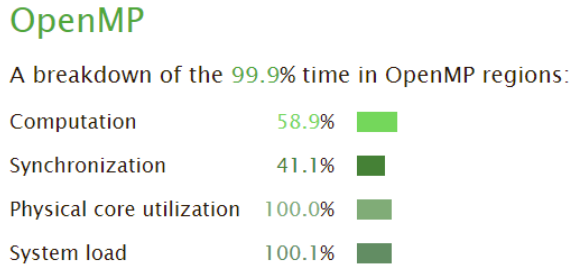


Figure 15: Breakdown of the OpenMP regions

This information shows that the program still could be optimized a lot more. The possible causes could be load imbalances or too many barriers and locks.

However, the most interesting information is to what

extent the parallelization allows to increase the scale of simulations. The following graph depicts the execution time of the simulation simulating 128 particles with respect to the number of cores used:

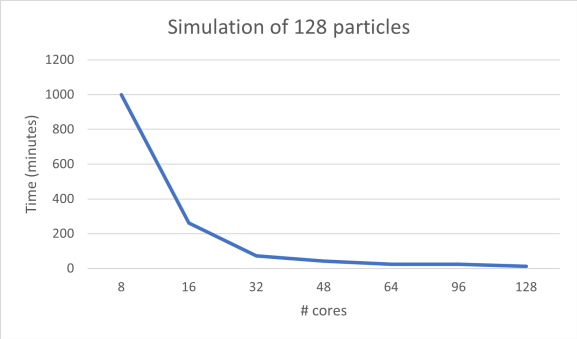


Figure 16: Performance measurement

Even if the program is far from being fully optimized, the parallelization allows for the simulation of up to 128 particles within a reasonable time amount. In fact, for this test, the execution took around 13 minutes.

To conclude, as a future objective, the inefficient memory usage of the program as well as the lack of vectorization could be a target for development. To elaborate, it may be interesting to analyze how performance changes in view of solving these remaining issues.

5.3.7. Visualization. The simulation is used to provide answers to the following 4 questions:

1) How does charge affect movement?
2) How does mass affect movement?
3) How do the direction and magnitude of the fields affect movement?
4) How does the number of particles affect movement?

Thus, simulations are performed for each question in almost identical conditions bare the variation of the attribute under investigation.

After evaluating the output of the simulation, the following has been observed. The acceleration of the particles is proportionate to the charge of the particles. Moreover a negatively charged particle moves in the opposite direction to a positively charged particle. Furthermore, the heavier the particle, the slower it accelerates. Then, the magnitude of the fields determines the magnitude of the forces they apply to the particles. Additionally, the direction of the movement is mainly influenced by the direction of the field. Finally, the particles of the same charge push each other away. When the number of particles is increased, the repelling force between the particles is stronger and they are pushed away faster.

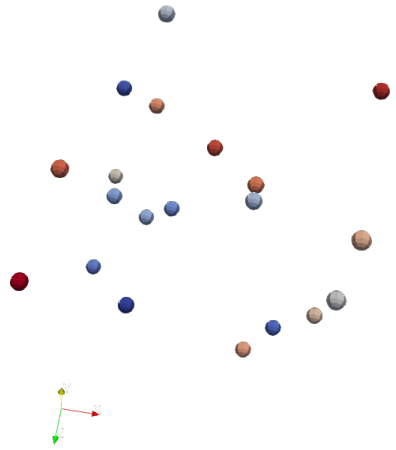


Figure 17: Simulation of a large number of identically charged particles close to each other

5.4. Assessment

This section provides an objective view of the quality and performance of the technical deliverable provided and also comments on whether all requirements have been fulfilled.

The program successfully simulates particles within an electric/magnetic field and takes advantage of being parallelized to perform larger-scale simulations. Unfortunately, one of the requirements was to have a menu for the simulation, which has not been produced. Automating the execution of the simulation using scripts was a priority during the development period. Thus, designing the program in a manner that it can easily be executed and utilized within the command line has been prioritized.

The information of the particles is stored within the text files as required and the user is able to decide the name of the output files for the simulation. Notable is that these files are established such that they can be used within Paraview to visualize the simulation.

All in all the development was successful, however, there is still optimization that could be done. Notably, the synchronization of threads and the inefficient memory access are two issues that could have been avoided with better program design. Otherwise, the program does what it is intended for and thus the technical deliverable is considered satisfactory.

Acknowledgment

This endeavor would not have been possible without Dr. Ezhilmathi Krishnasamy who provided invaluable guidance and feedback during this project.

Furthermore, thanks also goes to the BiCS management and education team for the amazing work done.

Lastly, the experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [15] – see https://hpc.uni.lu

6. Conclusion

This section concludes this Bachelor Semester Project. All in all, the objectives for both deliverables have been accomplished. Not only did this project allow me to learn about the notions of parallel programming and multi-threading but also the use of OpenMP, Paraview, HPC infrastructure, and performance analysis tools.

7. Plagiarism statement

This 350 words section without this first paragraph must be included in the submitted report and placed after the conclusion. This section is not counting in the total words quantity.

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
    1) Not putting quotation marks around a quote from another person's work
    2) Pretending to paraphrase while in fact quoting
    3) Citing incorrectly or incompletely
    4) Failing to cite the source of a quoted or paraphrased work
    5) Copying/reproducing sections of another person's work without acknowledging the source
    6) Paraphrasing another person's work without acknowledging the source
    7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
    8) Using another person's unpublished work without attribution and permission ('stealing')
    9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

References

[1] Blaise Barney et al. Introduction to parallel computing. *Lawrence Livermore National Laboratory*, 6(13):10, 2010.

[2] Cburnett. Multithreaded process.svg.

[3] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[4] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[5] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[6] Steven C Frautschi, Richard P Olenick, Tom M Apostol, and David L Goodstein. *The mechanical universe: Mechanics and heat*. Cambridge University Press, 1986.

[7] Intel®. Fix performance bottlenecks with intel® vtune™ profiler.

[8] Alina Kiessling. An introduction to parallel programming with openmp. In *The University of Edinburgh, A Pedagogical Seminar (accessed 24 September 2020), URL: https://www. roe. ac. uk/ifa/-postgrad/pedagogy/2009_kiessling. pdf*, 2009.

[9] Martha Kim. Scaling theory and machine abstractions, 2013.

[10] Michael Konopik, Till Korten, Eric Lutz, and Heiner Linke. Fundamental energy cost of finite-time parallelizable computing. *Nature Communications*, 14(1):447, 2023.

[11] Department of Atmospheric Sciences University of Illinois at Urbana Champaign. 3d collisions.

[12] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, 2018.

[13] Edward M. Purcell and David J. Morin. *Electricity and Magnetism*. Cambridge University Press, 3 edition, 2013.

[14] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts with Java*. Wiley Publishing, 2009.

[15] S. Varrette, H. Cartiaux, S. Peter, E. Kieffer, T. Valette, and A. Olloh. Management of an Academic HPC & Research Computing Facility: The ULHPC Experience 2.0. In *Proc. of the 6th ACM High Performance Computing and Cluster Technologies Conf. (HPCCT 2022)*, Fuzhou, China, July 2022. Association for Computing Machinery (ACM).

[16] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[17] Xipeng Wang, Simón Ramírez-Hinestrosa, Jure Dobnikar, and Daan Frenkel. The lennard-jones potential: when (not) to use it. *Physical Chemistry Chemical Physics*, 22(19):10624–10633, 2020.

8. Appendix

8.1. Code sections

8.1.1. Particle structure.

```
struct particleInfo {
  double mass;
  double charge;
  double radius;
  double epsilon;
  double sigma;
};

struct particle {
  double *position;
  double *velocity;
  double *acceleration;
  struct particleInfo *attributes;
};
```

8.1.2. Field structures.

```
struct electricField {
  double *fieldVector;
};

struct magneticField {
  double *fieldVector;
};
```

8.1.3. Movement Algortihm.

```
void move(struct particle *particle,double timeStep){
  double velProg[3];
  double posProg[3];
  double posAcc[3];


  // calculation of the change in velocity
  modifyVector(velProg, particle->acceleration);
  scalarMultiplication(timeStep,velProg);

  // calculation of the change in position

  // vt
  modifyVector(posProg, particle->velocity);
  modifyVector(posAcc, particle->acceleration);

  // (at^2)/2
  scalarMultiplication(timeStep,posProg);
  scalarMultiplication(0.5*pow(timeStep,2),posAcc);

  // adding the position and velocity variation to their
  // respective fields
  vectorAddition(particle->position,posProg);
  vectorAddition(particle->position,posAcc);

  vectorAddition(particle->velocity,velProg);
}
```

8.1.4. Charge-Particle Force.

```
double *electricForce(struct particle *p,
                      struct electricField *eF)
{
  double charge;
```

```
  double *elForce;

  // initialization of values for the final computation
  charge = chargeToCoulombs(p->attributes->charge);
  elForce = (double*)calloc(vecSize,sizeof(elForce));

  // qE
  modifyVector(elForce, eF->fieldVector);
  scalarMultiplication(charge,elForce);

  return elForce;
}

double *magneticForce(struct particle *p,
                      struct magneticField *mF)
{
  double charge;
  double *magForce;
  double crossP[3];

  // initialization of values for the final computation
  charge = chargeToCoulombs(p->attributes->charge);
  magForce = (double*)calloc(vecSize,sizeof(magForce));

  // v x B
  crossProduct(p->velocity,mF->fieldVector,crossP);

  // q(v x B)
  modifyVector(magForce,crossP);
  scalarMultiplication(charge,magForce);

  return magForce;
}

double *electroStaticForce(struct particle *p1, struct particle *p2){
  double *direct;
  double eSForce;

  // direction of the force vector
  direct = connectPoints(p1->position,p2->position);
  normalize(direct);

  // force magnitude
  eSForce = electroKonst*
            (chargeToCoulombs(p1->attributes->charge))*
            (chargeToCoulombs(p2->attributes->charge))/
            pow(norm(direct),2);

  // final vector
  scalarMultiplication(eSForce,direct);

  return direct;
}
```

8.1.5. Van Der Wals Force.

```
double *lennardJonesPotentialForce(struct particle *p1, struct particle *p2){
  double *direct;
  double distance;
  double sigma;
  double epsilon;
  double lJForce;

  // initialization of values for the final computation
```

```
  sigma = p1->attributes->sigma;
  epsilon = p1->attributes->epsilon;

  // get distance between the two particles and the direction for the force
  direct = connectPoints(p1->position,p2->position);
  distance = norm(direct);
  distance = meterToAngstrom(distance);
  normalize(direct);

  // calculate potential
  lJForce = 48*epsilon*(pow(sigma,6)/pow(distance,6))*
                        ((pow(sigma,6)/pow(distance,7))-
                         0.5*(pow(distance,-1)));

  // apply the intensity of the force to the direction unit vector
  // to obtain the acctual force vector
  scalarMultiplication(lJForce,direct);

  return direct;
}
```

8.1.6. applyFieldForce.

```
void applyFieldAcceleration(struct particle *p, struct electricField *eF, struct magneticField *mF){
  double massRatio = 1/uToKg(p->attributes->mass);
  double *force = (double*)calloc(vecSize,sizeof(force));
  double *eFo, *mFo;

  // for the 3 directions the acceleration force of both fields are calculated and added to the acceleration of the particle

  eFo = electricForce(p, eF);
  mFo = magneticForce(p, mF);

  vectorAddition(force, eFo);
  vectorAddition(force, mFo);

  scalarMultiplication(massRatio, force);

  // scaling
  scalarMultiplication(nanoToMeter(1),force);

  modifyVector(p->acceleration, force);

  free(eFo);
  free(mFo);
  free(force);
}
```

8.1.7. applyInterForce.

```
void applyInterAcceleration(struct particle *p1, struct particle *p2){
  double *force = (double*)calloc(vecSize,sizeof(force));
  double *temp = (double*)calloc(vecSize,sizeof(force));
  double *ljFo, *elstFo;

  // apply electrostatic force
  if (p1->attributes->charge != 0 && p2->attributes->charge != 0)
  {
    elstFo = electroStaticForce(p1,p2);
    modifyVector(temp, elstFo);

    // p2
    scalarMultiplication(1/uToKg(p2->attributes->mass),elstFo);
```

```
      #pragma omp critical
        vectorAddition(p2->acceleration, elstFo);

      // p1
      scalarMultiplication(-1/uToKg(p1->attributes->mass), temp);
      #pragma omp critical
        vectorAddition(p1->acceleration, temp);
      free(elstFo);
    }
    else {
      // apply lennard-jones force
      ljFo = lennardJonesPotentialForce(p1, p2);
      modifyVector(temp, ljFo);

      // p2
      scalarMultiplication(1/uToKg(p2->attributes->mass), ljFo);
      #pragma omp critical
        vectorAddition(p2->acceleration, ljFo);

      // p1
      scalarMultiplication(-1/uToKg(p1->attributes->mass), temp);
      #pragma omp critical
        vectorAddition(p1->acceleration, temp);
      free(ljFo);
    }

    free(temp);
    free(force);
}
```

8.1.8. Collision detection.

```
double checkPrioriCollision(struct particle *p1, struct particle *p2){
    double colRadius = 1E-12*(p1->attributes->radius + p2->attributes->radius);

    if (nanoToMeter(distance(p1->position,p2->position)) < colRadius)
    {
      return 0;
    }

    double **augementedMatrix = (double**)malloc(sizeof(double*)*3);
    double *c1 = (double*)malloc(sizeof(double)*3);
    double *c2 = (double*)malloc(sizeof(double)*3);
    double *c3 = (double*)malloc(sizeof(double)*3);
    double *solution;
    double ratio;

    // create augemented matrix corresponding to linear system composed of the instantaneous tragectory of two particles
    modifyVector(c1, p1->velocity);
    scalarMultiplication(timeStep, c1);

    modifyVector(c2, p2->velocity);
    scalarMultiplication(-timeStep, c2);

    modifyVector(c3, p2->position);
    vectorSubtraction(c3, p1->position);

    augementedMatrix[0] = c1;
    augementedMatrix[1] = c2;
    augementedMatrix[2] = c3;

    // solve the system to determine, whether the trajectory intersect

    solution = solveAugMatrix(augementedMatrix,3,2);
```

```
33
34    if (solution[0] < 0 || solution[1] < 0 || (solution[0] == 0 && solution[1] == 0 && !compareArray(p1->position,p2->position,3)))
35    {
36      ratio = NAN;
37      freeMatrix(augementedMatrix,3);
38      free(solution);
39
40      return ratio;
41    }
42
43    if (solution[1] == 0)
44    {
45      solution[0] = augementedMatrix[2][0]/
46          (augementedMatrix[0][0]+augementedMatrix[1][0]);
47      solution[1] = augementedMatrix[2][0]/
48          (augementedMatrix[0][0]+augementedMatrix[1][0]);
49      ratio = evaluateCollision(solution);
50      freeMatrix(augementedMatrix,3);
51      return ratio;
52    }
53
54    ratio = evaluateCollision(solution);
55    freeMatrix(augementedMatrix,3);
56
57    return ratio;
58  }
```

8.1.9. Collision handeling.

```
1   bool handleCollision(struct particle *p1, struct particle *p2){
2     double eval = checkPrioriCollision(p1,p2);
3     double *direct;
4     double a[3],b[3],c[3],d[3];
5     if (!isnan(eval))
6     {
7       if (eval)
8       {
9         move(p1,eval*timeStep);
10        move(p2,eval*timeStep);
11      }
12
13      modifyVector(a,p1->position);
14      modifyVector(b,p2->position);
15
16      modifyVector(c,p1->velocity);
17      modifyVector(d,p2->velocity);
18
19      scalarMultiplication(0.001,c);
20      scalarMultiplication(0.001,d);
21
22      vectorSubtraction(a,c);
23      vectorSubtraction(b,d);
24
25      // vector between the two centers of the particles
26      direct = connectPoints(a,b);
27      normalize(direct);
28
29      calcCollision(p1, p2, direct);
30      move(p1,(1-eval)*timeStep);
31      move(p2,(1-eval)*timeStep);
32      return 1;
33    }
34    return 0;
35  }
```

8.1.10. Particle sim.

```
for (i = 0; i < timePeriod*1000; ++i)
  {
    if (i % 100 == 0 && argc != 7)
    {
      printf("\n%lfs:\n",i*0.001);
    }
    // apply the acceleration to all particles
    for (j = 0; j < particleArraySize; ++j)
    {
      applyFieldAcceleration(particleArray[j],elFi,magFi);
    }

    // apply interaction forces
    for (j = 0; j < particleArraySize; ++j)
    {
      // checks all the remaining possible collisions and performs particle displacement
      for (k = j+1; k < particleArraySize; ++k)
      {
        applyInterAcceleration(particleArray[j],particleArray[k]);
      }
    }

    // check collisions and move the particles
    for (j = 0; j < particleArraySize; ++j)
    {
      if (handled[j] == 1)
      {
        continue;
      }
      // checks all the remaining possible collisions and performs particle displacement
      for (k = j+1; k < particleArraySize; ++k)
      {

        if (handleCollision(particleArray[j],particleArray[k]))
        {
          handled[j] = 1;
          handled[k] = 1;
          break;
        }

      }
    }

    for (j = 0; j < particleArraySize; ++j)
    {
      // check whether the movement of a particle has already been computed
      if (handled[j] == 0)
      {
        move(particleArray[j],currTimeStep);
      }
    }

    if (i % 100 == 0 && argc != 7)
    {
      for (int j = 0; j < particleArraySize; ++j)
      {
        printAttributes(particleArray[j]);
        handled[j] = 0;
      }

    }
```

8.1.11. Improved main loop.

```
#pragma omp parallel shared(particleArraySize,particleArray,elF,magF, particleIdxPartition, chunk, i)
    #pragma omp for schedule(static,1) private(j,k)
      for (j = 0; j < particleArraySize; ++j)
      {
        applyFieldAcceleration(particleArray[j],elF,magF);
        int id = omp_get_thread_num();
        int total = omp_get_num_threads();
        int *ptr;
        //printf("id: %d, total: %d, chunk: %d, part: %d\n", id, total, chunk,chunk/total );

              applyInterAcceleration(particleArray[j],particleArray,j,particleArraySize);

        for (k = chunk*id/total; k < chunk*(id+1)/total; ++k)
        {
          ptr = particleIdxPartition[k];
          if (ptr && handled[j] == 0 && handleCollision(particleArray[ptr[0]],particleArray[ptr[1]],ptr[0],handled))
          {
            handled[j] = 1;
            handled[k-chunk*id/total] = 1;
          }
        }
        // check whether the movement of a particle has already been computed
        if (handled[j] == 0)
        {
          move(particleArray[j],currTimeStep);
        }
        handled[j] = 0;
        scalarMultiplication(0,particleArray[j]->acceleration);
      }
```

8.1.12. Source Code.
electromagnetism.h

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include <stdbool.h>
#include "omp.h"


struct particleInfo {
   double mass;
   double charge;
   double radius;
   double epsilon;
   double sigma;
};

struct particle {
   double *position;
   double *velocity;
   double *acceleration;
   struct particleInfo *attributes;
};

struct electricField {
   double *fieldVector;
};

struct magneticField {
   double *fieldVector;
};

double getStep();
void setStep(double val);
struct particle *createParticle();
void freeParticle(struct particle *p);
void freeElField(struct electricField *eF);
void freeMagField(struct magneticField *mF);
void freeArrayOfParticles(struct particle **particleArray, int arraySize);
struct electricField *createElectricField();
struct magneticField *createMagneticField();
double *getParticleAttributes(FILE *particleDoc);
void setParticleInfo(struct particle *p, double *pos, double *vel, double *acc, double *attr);
void setElectricField(struct electricField *eF, double *norm);
void setMagneticField(struct magneticField *mF, double *norm);
void printDevInfo(struct particle *p);
void printAttributes(struct particle *p);
void printPos(struct particle *p, FILE *particleDoc, int ID);
void move(struct particle *particle,double timeStep);
void applyFieldAcceleration(struct particle *p, double *eF, double *mF);
void applyInterAcceleration(struct particle *p1, struct particle **particleArray, int selfIdx, int particleArraySize);
double *electricForce(struct particle *p, double *eF);
double *magneticForce(struct particle *p, double *mF);
double checkPrioriCollision(struct particle *p1, struct particle *p2);
double evaluateCollision(double *sol);
void calcCollision(struct particle *p1, struct particle *p2, double *direct);
bool handleCollision(struct particle *p1, struct particle *p2, int pID, int *handleArray);
void lennardJonesPotentialForce(struct particle *p1, struct particle *p2, double *storage);
void electroStaticForce(struct particle *p1, struct particle *p2, double *storage);
double chargeToCoulombs(double charge);
double nanoToMeter(double value);
double uToKg(double weight);
double meterToAngstrom(double distance);
```

linearAlgebra.h:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include "omp.h"
void setVecSize(int val);
double* createVector();
void freeMatrix(double **matrix, int col);
double* connectPoints(double* pos1, double* pos2);
void setVector(double *vector);
void modifyVector(double *oldVector, double *newVector);
void printVector(double *vector);
double norm(double* vector);
double distance(double* pos1, double* pos2);
void vectorAddition(double *vector1, double *vector2);
void vectorAtomicAddition(double *vector1, double *vector2);
void vectorSubtraction(double *vector1, double *vector2);
void scalarMultiplication(double scalar, double *vector);
double dotProduct(double *vector1, double *vector2);
void crossProduct(double *vector1, double *vector2,double* cP);
void normalize(double *vector);
double angleBetween(double *v1, double * v2);
double solve(double *equation, int eqSize);
double *solveAugMatrix(double **augMatrix, int numEq, int numVar);
```

testing.h:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool compareArray(double *array1, double *array2, int size);
bool compareValue(double v1, double v2, bool *check);
bool checkIfAllNAN(double *arr, int size);
bool isWithIn(double value, double min, double max);
```

particles.c:

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<dirent.h>
#include<time.h>
#include"linearAlgebra.h"
#include"electromagnetism.h"
#include "omp.h"

void printDirectory();
int **createPartition(int size);

double currTimeStep;

int main(int argc, char *argv[]){

   FILE *fp;
   FILE *outPutp;
   char *ptr;

   int menuInput = 0;
   int particleArraySize;
   int i = 0;
   int j = 0;
   int k;
   int timePeriod;
```

```
28      double deltaX;
29      double deltaY;
30      double deltaZ;
31
32      char fileNameInput[100];
33      char pathInput[200] = "./particleCollection/";
34
35      char fileNameOutput[150];
36
37      struct particle **particleArray;
38      struct particle *tempParticle;
39      struct particle *p;
40      struct electricField *elFi = createElectricField();
41      struct magneticField *magFi = createMagneticField();
42
43      currTimeStep = getStep();
44
45      double *posVec = createVector();
46      double *velVec = createVector();
47      double *accVec = createVector();
48      double *attr;
49      double elNorm[3];
50      double magNorm[3];
51      double initialVec[3] = {0,0,0};
52      double spawnSphere[3];
53      double spawnPoint[3];
54
55      if (argc == 2 && !strcmp(argv[1], "–listParticles"))
56      {
57
58        printDirectory();
59        free(elFi);
60        free(magFi);
61        free(posVec);
62        free(velVec);
63        free(accVec);
64        return 0;
65
66      }
67      else if (argc == 13 || argc == 14 || argc == 15)
68      {
69
70        strcpy(fileNameInput, argv[1]);
71        strcat(pathInput,fileNameInput);
72        particleArraySize = atoi(argv[2]);
73        timePeriod = atoi(argv[3]);
74
75        velVec[0] = strtod(argv[4], &ptr);
76        velVec[1] = strtod(argv[5], &ptr);
77        velVec[2] = strtod(argv[6], &ptr);
78
79        elNorm[0] = strtod(argv[7], &ptr);
80        elNorm[1] = strtod(argv[8], &ptr);
81        elNorm[2] = strtod(argv[9], &ptr);
82        for (int i = 0; i < 3; ++i)
83        {
84          spawnSphere[i] = atoi(argv[2])/2;
85        }
86        magNorm[0] = strtod(argv[10], &ptr);
87        magNorm[1] = strtod(argv[11], &ptr);
88        magNorm[2] = strtod(argv[12], &ptr);
89        modifyVector(spawnPoint,spawnSphere);
90      }
91      else
```

```
92  {
93    printf("Invalid Input!\n");
94    strcat(pathInput,"proton.txt\0");
95
96    particleArraySize = 128;
97    timePeriod = 1;
98
99    velVec[0] = 1;
100   velVec[1] = 0;
101   velVec[2] = 1;
102
103   elNorm[0] = 0;
104   elNorm[1] = 1;
105   elNorm[2] = 0;
106   for (int i = 0; i < 3; ++i)
107   {
108     spawnSphere[i] = atoi(argv[2])/2;
109   }
110   magNorm[0] = 1;
111   magNorm[1] = 0;
112   magNorm[2] = 1;
113   modifyVector(spawnPoint,spawnSphere);
114   argc = 15;
115
116 }
117
118 particleArray = (struct particle**)malloc(particleArraySize*sizeof(struct particle*));
119
120 modifyVector(posVec, initialVec);
121 modifyVector(accVec, initialVec);
122
123 setElectricField(elFi, elNorm);
124 setMagneticField(magFi,magNorm);
125
126 fp = fopen(pathInput,"r");
127 attr = getParticleAttributes(fp);
128 fclose(fp);
129
130 srand(time(0));
131
132 for (i = 0; i < particleArraySize; ++i)
133 {
134   tempParticle = createParticle();
135
136
137   deltaX = rand();
138   deltaY = rand();
139   deltaZ = rand();
140
141   spawnPoint[0] *= deltaX;
142   spawnPoint[1] *= deltaY;
143   spawnPoint[2] *= deltaZ;
144
145   scalarMultiplication(5E-10,spawnPoint);
146
147   setParticleInfo(tempParticle, spawnPoint, velVec, accVec, attr);
148
149   particleArray[i] = tempParticle;
150   modifyVector(spawnPoint,spawnSphere);
151 }
152
153 int *handled = (int*)malloc(particleArraySize*sizeof(handled));
154 double *elF = elFi->fieldVector;
155 double *magF = magFi->fieldVector;
```

```
156     int **particleIdxPartition = createPartition(particleArraySize);
157     int chunk = ((particleArraySize*(particleArraySize-1))/2);
158
159     for (int i = 0; i < particleArraySize; ++i)
160     {
161       handled[i] = 0;
162     }
163
164     for (i = 0; i < timePeriod*(1/currTimeStep); ++i)
165     {
166   /*
167       if (i % 100000 == 0 && argc == 15)
168       {
169         printf("\n%lfs:\n",i*currTimeStep);
170       }
171   */
172       // apply the acceleration to all particles
173
174       #pragma omp parallel shared(particleArraySize,particleArray,elF,magF, particleIdxPartition, chunk, i)
175         #pragma omp for schedule(static,1) private(j,k)
176           for (j = 0; j < particleArraySize; ++j)
177           {
178             applyFieldAcceleration(particleArray[j],elF,magF);
179             int id = omp_get_thread_num();
180             int total = omp_get_num_threads();
181             int *ptr;
182             //printf("id: %d, total: %d, chunk: %d, part: %d\n", id, total, chunk,chunk/total );
183
184             applyInterAcceleration(particleArray[j],particleArray,j,particleArraySize);
185
186             for (k = chunk*id/total; k < chunk*(id+1)/total; ++k)
187             {
188               ptr = particleIdxPartition[k];
189               if (ptr && handled[j] == 0 && handleCollision(particleArray[ptr[0]],particleArray[ptr[1]],ptr[0],handled))
190               {
191                 handled[j] = 1;
192                 handled[k-chunk*id/total] = 1;
193               }
194             }
195             // check whether the movement of a particle has already been computed
196             if (handled[j] == 0)
197             {
198               move(particleArray[j],currTimeStep);
199             }
200             handled[j] = 0;
201             scalarMultiplication(0,particleArray[j]->acceleration);
202           }
203
204       if (i % 100000 == 0 && argc == 15)
205       {
206         if (argc == 15)
207         {
208           snprintf(fileNameOutput, 150, "./visualData/%s.csv.%d", argv[14], i/100000);
209         } else {
210           snprintf(fileNameOutput, 150, "./visualData/timeStep.csv.%d", i/100000);
211         }
212
213         outPutp = fopen(fileNameOutput, "w");
214         fputs("x coord, y coord, z coord, scalar\n", outPutp);
215         for (j = 0; j < particleArraySize; ++j)
216         {
217           //printAttributes(particleArray[j]);
218           printPos(particleArray[j], outPutp, j);
219         }
```

```
220           fclose(outPutp);
221         }
222
223       }
224
225       freeArrayOfParticles(particleArray,particleArraySize);
226       free(handled);
227       free(posVec);
228       free(velVec);
229       free(accVec);
230       free(attr);
231       for (int i = 0; i < chunk; ++i)
232       {
233         free(particleIdxPartition[i]);
234       }
235       free(particleIdxPartition);
236
237       freeElField(elFi);
238       freeMagField(magFi);
239
240       return 0;
241     }
242     void printDirectory(){
243       FILE *fp;
244       DIR *folder;
245       struct dirent *dir;
246       char character;
247       char directoryName[] = "./particleCollection/";
248       char *fileLocation;
249       folder = opendir("./particleCollection/");
250
251
252       while((dir = readdir(folder)) != NULL){
253         if (strcmp(dir->d_name,".")!=0 && strcmp(dir->d_name,"..")!=0)
254         {
255           fileLocation = (char *)malloc(50*sizeof(fileLocation));
256           strcpy(fileLocation,directoryName);
257           strcat(fileLocation, dir->d_name);
258           fp = fopen(fileLocation, "r");
259           if (fp != NULL){
260             printf("%s:\n\n",dir->d_name);
261             character = (char)fgetc(fp);
262             while(character != EOF){
263               printf("%c",character);
264               character = (char)fgetc(fp);
265             }
266             printf("\n\n");
267             fclose(fp);
268           }
269
270           free(fileLocation);
271
272         }
273       }
274       closedir(folder);
275     }
276
277     int **createPartition(int size){
278       int partitionSize = (size*(size-1))/2;
279       if (partitionSize % omp_get_max_threads() != 0){
280         partitionSize += omp_get_max_threads() - (partitionSize % omp_get_max_threads());
281       }
282       int counter = 0;
283       int **partition = (int**)malloc(sizeof(int*)*partitionSize);
```

```
284     int *pair;
285     for (int i = 0; i < size; ++i)
286     {
287       for (int j = i+1; j < size; ++j)
288       {
289         pair = (int*)malloc(sizeof(int)*2);
290         pair[0] = i;
291         pair[1] = j;
292         partition[counter] = pair;
293         counter++;
294       }
295     }
296
297
298     for (int i = counter; i < partitionSize; ++i)
299     {
300       partition[i] = NULL;
301     }
302     return partition;
303   }
```

electromagnetismImp.c:

```
1   #include<stdlib.h>
2   #include<stdio.h>
3   #include<stdbool.h>
4   #include<math.h>
5   #include"electromagnetism.h"
6   #include"linearAlgebra.h"
7   #include "testing.h"
8   #include "omp.h"
9
10  int vecSize = 3;
11  double timeStep = 0.000001;
12  double electroKonst = 8.987551792E9;
13  double chargeToCoulombsRatio = 1.6022E-19;
14  double uToKgRatio = 1.66054E-27;
15  double avogadroConst = 6.02214076E23;
16  double scaling = 1E-7;
17
18  double getStep(){
19    return timeStep;
20  }
21
22  void setStep(double val){
23    timeStep = val;
24  }
25
26  /*
27    allocate memory for a particle struct
28  */
29  struct particle *createParticle(){
30    struct particle *p = (struct particle*)malloc(sizeof(struct particle));
31    p->position = (double*)malloc(vecSize*sizeof(p->position));
32    p->velocity = (double*)malloc(vecSize*sizeof(p->velocity));
33    p->acceleration = (double*)malloc(vecSize*sizeof(p->acceleration));
34    p->attributes = (struct particleInfo*)malloc(sizeof(struct particleInfo));
35    return p;
36  }
37  /*
38    free memory allocated for a particle struct
39  */
40  void freeParticle(struct particle *p){
41    free(p->position);
42    free(p->velocity);
```

```
43    free(p->acceleration);
44    free(p->attributes);
45    free(p);
46  }
47
48  /*
49    free memory allocated for a electricField struct
50  */
51  void freeElField(struct electricField *eF){
52    free(eF->fieldVector);
53    free(eF);
54  }
55
56  /*
57    free memory allocated for a magneticField struct
58  */
59  void freeMagField(struct magneticField *mF){
60    free(mF->fieldVector);
61    free(mF);
62  }
63
64  /*
65    free memory allocated for all particles in an array including the array itself
66  */
67  void freeArrayOfParticles(struct particle **particleArray, int arraySize){
68    for (int i = 0; i < arraySize; ++i)
69    {
70      freeParticle(particleArray[i]);
71    }
72    free(particleArray);
73  }
74
75  /*
76    allocate memory for a electricField struct
77  */
78  struct electricField *createElectricField(){
79    struct electricField *eF = (struct electricField*)malloc(sizeof(eF));
80    eF->fieldVector = (double*)malloc(sizeof(eF->fieldVector)*vecSize);
81  }
82
83  /*
84    allocate memory for a magneticField struct
85  */
86  struct magneticField *createMagneticField(){
87    struct magneticField *mF = (struct magneticField*)malloc(sizeof(mF));
88    mF->fieldVector = (double*)malloc(sizeof(mF->fieldVector)*vecSize);
89  }
90
91  /*
92    reads file containg information on a particle and returns an array containing all important values
93  */
94  double *getParticleAttributes(FILE *particleDoc){
95    double *attr = (double*)malloc(5*sizeof(double));
96    char output[15];
97    char *ptr;
98
99    // Extract mass from a file
100   fgets(output, 15, particleDoc);
101   fgets(output, 7, particleDoc);
102   fgets(output, 7, particleDoc);
103   attr[0] = strtod(output,&ptr);
104
105   // Extract charge from file
106   fgets(output, 9, particleDoc);
```

```
107     fgets(output, 7, particleDoc);
108     attr[1] = strtod(output,&ptr);
109
110     // Extract radius from file
111     fgets(output, 9, particleDoc);
112     fgets(output, 7, particleDoc);
113     attr[2] = strtod(output,&ptr);
114
115     // Extract epsilon from file
116     fgets(output, 10, particleDoc);
117     fgets(output, 7, particleDoc);
118     attr[3] = strtod(output,&ptr);
119
120     // Extract the sigma from the file
121     fgets(output, 8, particleDoc);
122     fgets(output, 7, particleDoc);
123     attr[4] = strtod(output,&ptr);
124
125     return attr;
126   }
127
128   /*
129     takes 4 arrays containing initial position, velcoity and acceleration and attributes and assigns them to a particle
130   */
131   void setParticleInfo(struct particle *p, double *pos, double *vel, double *acc, double * attr){
132     for (int i = 0; i < vecSize; ++i)
133     {
134       p->position[i] = pos[i];
135       p->velocity[i] = vel[i];
136       p->acceleration[i] = acc[i];
137     }
138     p->attributes->mass = attr[0];
139     p->attributes->charge = attr[1];
140     p->attributes->radius = attr[2];
141     p->attributes->epsilon = attr[3];
142     p->attributes->sigma = attr[4];
143
144   }
145
146   /*
147     sets the electric field to the given array 'norm'
148   */
149   void setElectricField(struct electricField *eF, double *norm){
150     for (int i = 0; i < vecSize; ++i)
151     {
152       eF->fieldVector[i] = norm[i];
153     }
154   }
155
156   /*
157     sets the magnetic field to the given array 'norm'
158   */
159   void setMagneticField(struct magneticField *mF, double *norm){
160     for (int i = 0; i < vecSize; ++i)
161     {
162       mF->fieldVector[i] = norm[i];
163     }
164   }
165
166   /*
167     print the memory addresses of fields of a particle struct
168   */
169   void printDevInfo(struct particle *p){
170     printf("Struct address:%p\n\n",p );
```

```
171     printf("Dynamics: \n");
172     printf("Position: %p\n",p->position);
173     printf("Velocity: %p\n",p->velocity);
174     printf("Acceleration: %p\n",p->acceleration);
175     printf("\nAttributes:\n");
176     printf("Struct: %p\n",p->attributes);
177     printf("Mass: %p\n",&(p->attributes->mass));
178     printf("Charge: %p\n",&(p->attributes->charge));
179     printf("Radius: %p\n",&(p->attributes->radius));
180   }
181
182   /*
183     print the fields of a particle struct
184   */
185   void printAttributes(struct particle *p){
186     printf("Position:");
187     printVector(p->position);
188     printf("Velocity:");
189     printVector(p->velocity);
190     printf("Acceleration:");
191     printVector(p->acceleration);
192     printf("\n");
193     printf("Attributes: %lf, %lf, %lf,
            %lf\n\n",p->attributes->mass,p->attributes->charge,p->attributes->radius,p->attributes->epsilon);
194   }
195
196   /*
197     print position into a given file for csv usage
198   */
199   void printPos(struct particle *p, FILE *particleDoc, int ID){
200     char buffer[300];
201     for (int i = 0; i < 2; ++i)
202     {
203       snprintf(buffer, 300, "%lf, ", p->position[i]);
204       fputs(buffer,particleDoc);
205     }
206     snprintf(buffer, 300, "%lf, %d\n", p->position[2], ID);
207     fputs(buffer,particleDoc);
208   }
209
210   /*
211     calculates the change in position and velocity of a particle and modifies the particles fields accordingly
212   */
213   void move(struct particle *particle,double timeStep){
214     double velProg[3];
215     double posProg[3];
216     double posAcc[3];
217
218
219     // calculation of the change in velocity
220     modifyVector(velProg, particle->acceleration);
221     scalarMultiplication(timeStep,velProg);
222
223     // calculation of the change in position
224
225     // vt
226     modifyVector(posProg, particle->velocity);
227     modifyVector(posAcc, particle->acceleration);
228
229     // (atˆ2)/2
230     scalarMultiplication(timeStep,posProg);
231     scalarMultiplication(0.5*pow(timeStep,2),posAcc);
232
233     // adding the position and velocity variation to their respective fields
```

```
234     vectorAddition(particle->position,posProg);
235     vectorAddition(particle->position,posAcc);
236
237     vectorAddition(particle->velocity,velProg);
238   }
239
240   /*
241     applies the acceleration cause from electric and magnetic fields
242   */
243   void applyFieldAcceleration(struct particle *p, double *eF, double *mF){
244     double massRatio = 1/uToKg(p->attributes->mass);
245     double *force = (double*)calloc(vecSize,sizeof(force));
246     double *eFo, *mFo;
247
248     // for the 3 directions the acceleration force of both fields are calculated and added to the acceleration of the particle
249
250     eFo = electricForce(p, eF);
251     mFo = magneticForce(p, mF);
252
253     vectorAddition(force, eFo);
254     vectorAddition(force, mFo);
255
256     scalarMultiplication(massRatio, force);
257     scalarMultiplication(scaling, force);
258
259     vectorAddition(p->acceleration, force);
260
261     free(eFo);
262     free(mFo);
263     free(force);
264   }
265
266   void applyInterAcceleration(struct particle *p1, struct particle **particleArray, int selfIdx, int particleArraySize){
267     double force[3];
268     double ljFo[3], elstFo[3];
269
270     if (particleArraySize == 0){
271       return;
272     }
273     // apply electrostatic force
274     if (p1->attributes->charge != 0)
275     {
276
277       for (int i = 0; i < particleArraySize; ++i)
278       {
279         if (i != selfIdx)
280         {
281           electroStaticForce(p1,particleArray[i],elstFo);
282           vectorAddition(force, elstFo);
283         }
284
285       }
286
287       scalarMultiplication(-1/uToKg(p1->attributes->mass), elstFo);
288
289       // p1
290       vectorAddition(p1->acceleration, elstFo);
291
292
293     }
294     else {
295
296       for (int i = 0; i < particleArraySize; ++i)
297       {
```

```
298           lennardJonesPotentialForce(p1, particleArray[i],ljFo);
299           vectorAddition(force, ljFo);
300         }
301
302         scalarMultiplication(-1/uToKg(p1->attributes->mass), ljFo);
303
304         vectorAddition(p1->acceleration, elstFo);
305       }
306     }
307
308     double *electricForce(struct particle *p, double *eF){
309       double charge;
310       double *elForce;
311
312       // initialization of values for the final computation
313       charge = chargeToCoulombs(p->attributes->charge);
314       elForce = (double*)calloc(vecSize,sizeof(elForce));
315
316       // qE
317       modifyVector(elForce, eF);
318       scalarMultiplication(charge,elForce);
319
320       return elForce;
321     }
322
323     double *magneticForce(struct particle *p, double *mF){
324       double charge;
325       double *magForce;
326       double crossP[3];
327
328       // initialization of values for the final computation
329       charge = chargeToCoulombs(p->attributes->charge);
330       magForce = (double*)calloc(vecSize,sizeof(magForce));
331
332       // v x B
333       crossProduct(p->velocity,mF,crossP);
334
335       // q(v x B)
336       modifyVector(magForce,crossP);
337       scalarMultiplication(charge,magForce);
338
339       return magForce;
340     }
341
342     /*
343       checks a priori wheter two particles collide
344     */
345     double checkPrioriCollision(struct particle *p1, struct particle *p2){
346       double colRadius = 1E-12*(p1->attributes->radius + p2->attributes->radius);
347
348       if (nanoToMeter(distance(p1->position,p2->position)) < colRadius)
349       {
350         return 0;
351       }
352
353       double **augementedMatrix = (double**)malloc(sizeof(double*)*3);
354       double *c1 = (double*)malloc(sizeof(double)*3);
355       double *c2 = (double*)malloc(sizeof(double)*3);
356       double *c3 = (double*)malloc(sizeof(double)*3);
357       double *solution;
358       double ratio;
359
360       // create augemented matrix corresponding to linear system composed of the instantaneous tragectory of two particles
361       modifyVector(c1, p1->velocity);
```

```
362     scalarMultiplication(timeStep, c1);
363
364     modifyVector(c2, p2->velocity);
365     scalarMultiplication(-timeStep, c2);
366
367     modifyVector(c3, p2->position);
368     vectorSubtraction(c3, p1->position);
369
370     augementedMatrix[0] = c1;
371     augementedMatrix[1] = c2;
372     augementedMatrix[2] = c3;
373
374     // solve the system to determine, whether the trajectory intersect
375
376     solution = solveAugMatrix(augementedMatrix,3,2);
377
378     if (solution[0] < 0 || solution[1] < 0 || (solution[0] == 0 && solution[1] == 0 && !compareArray(p1->position,p2->position,3)))
379     {
380       ratio = NAN;
381       freeMatrix(augementedMatrix,3);
382       free(solution);
383       return ratio;
384     }
385
386     if (solution[1] == 0)
387     {
388       solution[0] = augementedMatrix[2][0]/(augementedMatrix[0][0]+augementedMatrix[1][0]);
389       solution[1] = augementedMatrix[2][0]/(augementedMatrix[0][0]+augementedMatrix[1][0]);
390       ratio = evaluateCollision(solution);
391       freeMatrix(augementedMatrix,3);
392       free(solution);
393       return ratio;
394     }
395
396     ratio = evaluateCollision(solution);
397     freeMatrix(augementedMatrix,3);
398     return ratio;
399   }
400
401   double evaluateCollision(double *sol){
402     double distance = 0;
403     if (!checkIfAllNAN(sol,2) && isWithIn(sol[0],0,1) && isWithIn(sol[1],0,1) && abs(sol[0]-sol[1] < 0.001))
404     {
405       distance = sol[0];
406       free(sol);
407       return distance;
408     }
409     free(sol);
410     return NAN;
411   }
412
413   /*
414     calculates the forces caused by a collision and applies them on the respective particles
415   */
416   void calcCollision(struct particle *p1, struct particle *p2, double *direct){
417
418     double vCenter1[3];
419     double vCenter2[3];
420     double vNormal1[3];
421     double vNormal2[3];
422
423     if (abs(norm(p1->velocity) < 1E-10) || abs(norm(p2->velocity)) < 1E-10)
424     {
425       double relativeSpeed[3];
```

```
426       double normalSpeed[3];
427
428       // calculate the relative velocity
429       modifyVector(relativeSpeed,p1->velocity);
430       vectorSubtraction(relativeSpeed,p2->velocity);
431
432       // calculate the velocity in the direction of 'direct'
433       modifyVector(normalSpeed,direct);
434       scalarMultiplication(dotProduct(relativeSpeed,direct),normalSpeed);
435
436
437       // the respective normal (direct) forces are applied on the other particle
438       vectorAddition(p2->velocity,normalSpeed);
439       scalarMultiplication(-1,normalSpeed);
440       vectorAddition(p1->velocity,normalSpeed);
441     } else{
442       double angle1 = angleBetween(direct, p1->velocity);
443       scalarMultiplication(-1,direct);
444       double angle2 = angleBetween(direct, p2->velocity);
445
446       modifyVector(vCenter1,p1->velocity);
447       scalarMultiplication(cos(angle1),vCenter1);
448
449       modifyVector(vCenter2,p2->velocity);
450       scalarMultiplication(cos(angle2),vCenter2);
451
452       modifyVector(vNormal1, p1->velocity);
453       vectorSubtraction(vNormal1, vCenter1);
454
455       modifyVector(vNormal2, p2->velocity);
456       vectorSubtraction(vNormal2, vCenter2);
457
458       modifyVector(p1->velocity, vCenter2);
459       modifyVector(p2->velocity, vCenter1);
460
461       vectorSubtraction(p1->velocity, vNormal1);
462       vectorSubtraction(p2->velocity, vNormal2);
463     }
464
465     free(direct);
466
467   }
468
469   /*
470     executes necessary functions for the checking of collisions
471   */
472   bool handleCollision(struct particle *p1, struct particle *p2, int pID, int *handleArray){
473     double eval = checkPrioriCollision(p1,p2);
474     double *direct;
475     double a[3],b[3],c[3],d[3];
476     int check = 1;
477
478     if (!(handleArray[pID] == 0 && !isnan(eval)))
479     {
480       check = 0;
481     }
482
483     if (check == 0)
484     {
485       return check;
486     }
487     if (eval)
488     {
489       #pragma omp critical(lizz)
```

```
490         move(p1,eval*timeStep);
491         move(p2,eval*timeStep);
492     }
493
494     modifyVector(a,p1->position);
495     modifyVector(b,p2->position);
496
497     modifyVector(c,p1->velocity);
498     modifyVector(d,p2->velocity);
499
500     scalarMultiplication(0.001,c);
501     scalarMultiplication(0.001,d);
502
503     vectorSubtraction(a,c);
504     vectorSubtraction(b,d);
505     // vector between the two centers of the particles
506     direct = connectPoints(a,b);
507     normalize(direct);
508
509     #pragma omp critical(lizz)
510       calcCollision(p1, p2, direct);
511       move(p1,(1-eval)*timeStep);
512       move(p2,(1-eval)*timeStep);
513
514     return check;
515   }
516
517   /*
518     approximates the van der waals force between two particles
519   */
520   void lennardJonesPotentialForce(struct particle *p1, struct particle *p2, double *storage){
521     double *direct;
522     double distance;
523     double sigma;
524     double epsilon;
525     double lJForce;
526
527     // initialization of values for the final computation
528     sigma = p1->attributes->sigma;
529     epsilon = p1->attributes->epsilon;
530
531     // get distance between the two particles and the direction for the force
532     direct = connectPoints(p1->position,p2->position);
533     distance = norm(direct);
534     distance = meterToAngstrom(distance);
535     normalize(direct);
536
537     // calculate potential
538     lJForce = 48*epsilon*(pow(sigma,6)/pow(distance,6))*((pow(sigma,6)/pow(distance,7))-0.5*(pow(distance,-1)));
539
540     // apply the intensity of the force to the direction unit vector to obtain the acctual force vector
541     scalarMultiplication(lJForce,direct);
542
543     modifyVector(storage, direct);
544     free(direct);
545   }
546
547   void electroStaticForce(struct particle *p1, struct particle *p2,double *storage){
548     double *direct;
549     double distance;
550     double eSForce;
551
552     // get distance between the two particles and the direction for the force
553     direct = connectPoints(p1->position,p2->position);
```

```
554     distance = norm(direct);
555     normalize(direct);
556 
557     // force magnitude
558     eSForce =
            electroKonst*(chargeToCoulombs(p1->attributes->charge))*(chargeToCoulombs(p2->attributes->charge))/pow(distance,2);
559 
560     // final vector
561     scalarMultiplication(eSForce,direct);
562 
563     modifyVector(storage, direct);
564     free(direct);
565 }
566 
567 double chargeToCoulombs(double charge){
568     return chargeToCoulombsRatio*charge;
569 }
570 
571 double nanoToMeter(double value){
572     return value*1E-9;
573 }
574 
575 double uToKg(double weight){
576     return uToKgRatio*weight;
577 }
578 
579 double meterToAngstrom(double distance){
580     return 1E10*distance;
581 }
```

linearAlgebraImp.c:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <math.h>
5  #include "linearAlgebra.h"
6  #include "omp.h"
7 
8 
9  int vectorSize = 3;
10 
11 /*
12     creates and returns a new vector of size 3;
13 */
14 double* createVector(){
15     double* newVector = (double*)malloc(vectorSize*sizeof(newVector));
16     return newVector;
17 }
18 
19 void freeMatrix(double **matrix, int col){
20     for (int i = 0; i < col; ++i)
21     {
22         free(matrix[i]);
23     }
24     free(matrix);
25 }
26 
27 /*
28     creates and returns a new vector that connects two points pos1 and pos2 in 3D space
29 */
30 double* connectPoints(double* pos1, double* pos2){
31     double* newVector = (double*)malloc(vectorSize*sizeof(newVector));
32     for (int i = 0; i < vectorSize; ++i)
33     {
```

```
34        newVector[i] = pos2[i] – pos1[i];
35      }
36      return newVector;
37    }
38
39    /*
40      prompts the user to input values for a vector
41    */
42    void setVector(double *vector){
43      printf("Please input the vector values (x,y,z): ");
44      for (int i = 0; i < vectorSize; ++i)
45      {
46        scanf("%lf",vector+i);
47      }
48    }
49
50    /*
51      modifies the values of an existing vector 'oldVector' tom atch the values of a given vector
52    */
53    void modifyVector(double *oldVector, double *newVector){
54      for (int i = 0; i < vectorSize; ++i)
55      {
56        #pragma omp atomic read
57          oldVector[i] = newVector[i];
58      }
59    }
60
61    /*
62      prints the values of a given vector
63    */
64    void printVector(double *vector){
65      printf("(");
66      for (int i = 0; i < vectorSize–1; ++i)
67      {
68        printf("%lf,", vector[i] );
69      }
70      printf("%lf",vector[2]);
71      printf(")\n");
72    }
73
74    /*
75      calculates and retuns the Euclidean norm of a given vector
76    */
77    double norm(double* vector){
78      double norm = 0;
79      norm = sqrt(vector[0]*vector[0] + vector[1]*vector[1] + vector[2]*vector[2]);
80      return norm;
81    }
82
83    double distance(double* pos1, double* pos2){
84      double* vec = connectPoints(pos1,pos2);
85      double distance = norm(vec);
86      free(vec);
87      return distance;
88    }
89
90    /*
91      adds vector2 to vector1
92    */
93    void vectorAddition(double *vector1, double *vector2){
94      for (int i = 0; i < vectorSize; ++i)
95      {
96        vector1[i] += vector2[i];
97      }
```

```
98  }
99
100 /*
101   adds vector2 to vector1
102 */
103 void vectorAtomicAddition(double *vector1, double *vector2){
104   for (int i = 0; i < vectorSize; ++i)
105   {
106     #pragma omp atomic
107       vector1[i] += vector2[i];
108   }
109 }
110
111 /*
112   subtracts vector2 from vector 1
113 */
114 void vectorSubtraction(double *vector1, double *vector2){
115   for (int i = 0; i < vectorSize; ++i)
116   {
117     vector1[i] -= vector2[i];
118   }
119 }
120
121 /*
122   multiplies each element of a given vector by a scalar value
123 */
124 void scalarMultiplication(double scalar, double *vector){
125   int i;
126   for (i = 0; i < vectorSize; ++i)
127   {
128     vector[i] *= scalar;
129   }
130 }
131
132 /*
133   calculates and returns the dot product of two given vectors
134 */
135 double dotProduct(double *vector1, double *vector2){
136   double dP = 0;
137   int i;
138   #pragma omp parallel shared(vectorSize) private(i) reduction(+: dP)
139     for (int i; i < vectorSize; ++i)
140     {
141       dP += vector1[i]*vector2[i];
142     }
143   return dP;
144 }
145
146 /*
147   calculates the cross product of two given vectors and stores its values in the vector cP
148 */
149 void crossProduct(double *vector1, double *vector2, double* cP){
150   cP[0] = vector1[1]*vector2[2]-vector2[1]*vector1[2];
151   cP[1] = -vector1[0]*vector2[2]+vector2[0]*vector1[2];
152   cP[2] = vector1[0]*vector2[1]-vector2[0]*vector1[1];
153 }
154
155 /*
156   normalizes a given vector
157 */
158 void normalize(double *vector){
159   double vNorm = norm(vector);
160   if (vNorm == 0)
161   {
```

```
      return;
    }
    for (int i = 0; i < vectorSize; ++i)
    {
      vector[i] /= vNorm;
    }
  }

  /*
    gives the angle between two vectors
  */
  double angleBetween(double *v1, double * v2){
    return acos((dotProduct(v1,v2))/(norm(v1)*norm(v2)));
  }

  /*
    searches for unique solution of an equation
    assuming equation is linear
  */
  double *solveAugMatrix(double **augMatrix, int numEq, int numVar){
    double ratio, sub;
    double *solution = (double*)malloc(sizeof(solution)*numVar);
    double temp;
    int i,j,k;
    // transform matrix into reduced row echelon form
    // int i decides which column to operate on
    for (i = 0; i < numVar; ++i)
    {
      // makes sure that augementmatrix[i][i] is not 0
      if (augMatrix[i][i] == 0)
      {

        for (j = i+1; j < numEq; ++j)
        {

          if (augMatrix[i][j] != 0)
          {

            for (k = 0; k < numVar+1; ++k)
            {

              temp = augMatrix[k][i];
              augMatrix[k][i] = augMatrix[k][j];
              augMatrix[k][j] = temp;
            }

            break;
          }
        }
      }



      if (augMatrix[i][i] != 0)
      {
        // makes sure there is a leading one within the row
        ratio = 1.0/(augMatrix[i][i]);
        for (j = 0; j < numVar+1; ++j)
        {
          augMatrix[j][i] *= ratio;
        }

        // makes sure the values above and below the leading are turned to 0
        // j indicates which row is operated on
```

```
226         for (j = 0; j < numEq; ++j)
227         {
228           if (j != i)
229           {
230             sub = augMatrix[i][j];
231             // k indicates on which element of the row is operated on
232             for (k = 0; k < numEq; ++k)
233             {
234               augMatrix[k][j] -= sub*augMatrix[k][i];
235             }
236           }
237         }
238       }
239
240     }
241
242     // determine what the solution is
243
244     if (!(abs(augMatrix[numVar][numEq-1]) < 1E-10))
245     {
246       // no solution -> return an array containing only NAN
247       for (int i = 0; i < numVar; ++i)
248       {
249         solution[i] = NAN;
250       }
251     } else {
252       // the solution is contained within the column containing the equantion constants
253       for (int i = 0; i < numVar; ++i)
254       {
255         solution[i] = augMatrix[numVar][i];
256       }
257     }
258
259     return solution;
260 }
```

testingImp.c:

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<stdbool.h>
4  #include<math.h>
5  #include"testing.h"
6  #include"linearAlgebra.h"
7
8
9  bool compareArray(double *array1, double *array2, int size){
10   for (int i = 0; i < size; ++i)
11   {
12     if (!(abs(array1[i]-array2[i]<0.1)))
13     {
14       /*
15       printf("\n");
16       printf("Error: array not equal!\n");
17       printVector(array1);
18       printVector(array2);
19       */
20       printf("%d: %lf is not %lf",i,array1[i], array2[i]);
21       return false;
22     }
23   }
24
25   /*
26   printf("\n");
27   printf("Correct: array equal!\n");
```

```
28    printVector(array1);
29    printVector(array2);
30    */
31    return true;
32  }
33
34  bool compareValue(double v1, double v2, bool *check){
35    if (v1 != v2)
36    {
37      printf("Error: \n");
38      printf("%lf is not equal to %lf \n", v1, v2 );
39      *check = false;
40      return false;
41    }
42    return true;
43  }
44
45  bool checkIfAllNAN(double *arr, int size){
46    bool result = true;
47    for (int i = 0; i < size; ++i)
48    {
49      if (!isnan(arr[i]))
50      {
51        result = false;
52      }
53    }
54    return result;
55  }
56
57  bool isWithIn(double value, double min, double max){
58    if (value >= min && value < max)
59    {
60      return true;
61    }
62    return false;
63  }
```