MASTER'S THESIS

# Dynamic Particle Data Structures for Wigner Monte Carlo Simulations

Ausgeführt am
Institut für Mikroelektronik
der Technischen Universität Wien

unter Anleitung von

**Associate Prof. Dipl.-Ing. Dr.techn. Josef Weinbub**

durch

**Alexander Adel**

Matrikelnummer 01325110
Studienkennzahl 066 646

Wien, im März 2023

## Abstract

The over many decades lasting and still continuing miniaturization of semiconductor devices, combined with the subsequent increase of the integration density in microprocessors, led to minuscule device sizes in the range of a few nanometers. Quantum transport models are the only way to describe the plethora of quantum effects that are taking place in these systems. Among the models is the *signed-particle* WIGNER *model*, which utilizes an ensemble *Monte Carlo* concept to solve the WIGNER transport equation and thus offers an intuitive interpretation of quantum electron transport dynamics and provides a clear analogy to classical notions. The stochastic nature of the signed-particle model demands an extremely large number of numerical particles to reduce the variance of the resulting values. Additionally, the necessary particle generation and annihilation events, which are essential for practical utilization, modify the number of the simulated particles in every time step, further contributing to the computational challenge. Due to these challenges, the main objective of this thesis is an extensive quantitative performance analysis of dynamic particle data structures suitable for signed-particle WIGNER models, that store all particles of the ensemble and their assigned attributes. The reference simulation tool ViennaWD, which utilizes a signed-particle WIGNER model, is consulted to derive the necessary particle properties. Related *open source* projects, especially Monte Carlo particle simulators, are examined to obtain further possibilities for different implementations. The emphasis lies on the utilization of modern *supercomputer clusters* that employ multiple layers of parallelization techniques, such as the *Vienna Scientific Cluster* (VSC). A number of promising data structure implementations and corresponding test functions, which are inspired by the operations of the signed-particle model and necessary particle algorithms, are combined into a categorically and rigorously designed *benchmark framework*. This framework is written in the C$^{++}$ programming language. It is used to perform numerical experiments on the VSC and yields execution times which are then analyzed. The goal is to develop a software benchmark tool that is able to determine particle data structure implementations which offer the best performance for the considered task. Even though the runtime results at the end of this thesis feature a variety of data structure designs from which a hierarchy of best suited implementations can be derived, the general objective lies in the usability, reusability, maintainability, flexibility and expandability of the benchmark framework. New data structures can be defined and easily incorporated into the framework to obtain execution runtimes for all available test functions. These concepts support the possibility to use the application to yield further insights beyond the implementations featured in this thesis and reinforces the principles of modern software engineering for computational science and engineering.

## Kurzfassung

Die über viele Jahrzehnte andauernde und immer noch stattfindende Miniaturisierung von Halbleiterbauelementen, verbunden mit dem darauf folgenden Anstieg der Integrationsdichte in Mikroprozessoren, führte zu winzigen Bauelementgrößen in dem Bereich von ein paar Nanometern. Quantentransportmodelle sind die einzige Möglichkeit, die große Menge an Quanteneffekten, welche in diesen Systemen stattfinden, zu beschreiben. Unter diesen Modellen ist das *Signed-Particle*-Wigner-*Modell*, welches ein Ensemble-*Monte-Carlo*-Konzept verwendet, um die Wigner-Transportgleichung zu lösen und somit eine intuitive Interpretation von Quanten-Elektronentransportdynamiken und eine klare Analogie zu klassischen Auffassungen anbietet. Der stochastische Charakter des *Signed-Particle*-Modells verlangt eine extrem hohe Anzahl an numerischen Partikeln für die Reduktion der Varianz der resultierenden Werte. Zusätzlich verändern die notwendigen Generations- und Vernichtungs-Events der Partikel, welche essenziell für die praktische Anwendung sind, die Anzahl der simulierten Partikel in jedem Zeitschritt, womit die berechnungstechnische Herausforderung noch weiter erhöht wird. Aufgrund dieser Herausforderungen ist das Hauptziel dieser Arbeit eine ausführliche quantitative Performanceanalyse von dynamischen, für Signed-Particle-Wigner-Modelle geeigneten, Partikel-Datenstrukturen, welche alle Partikel des Ensembles und deren zugehörige Attribute verwalten. Als Referenz wird das Simulationsprogramm ViennaWD, welches ein Signed-Particle-Wigner-Modell anwendet, herangezogen, um die notwendigen Eigenschaften der Partikel zu bestimmen. Damit in Zusammenhang stehende *Open-Source*-Projekte, besonders Monte-Carlo-Partikel-Simulatoren, werden untersucht, um weitere Möglichkeiten für verschiedene Implementierungen zu erhalten. Der Schwerpunkt liegt auf der Verwendung von modernen *Supercomputer-Clustern*, welche mehrere Ebenen an Parallelisierungstechniken anwenden, so wie der *Vienna-Scientific-Cluster* (VSC). Eine Anzahl an vielversprechenden Datenstrukturimplementierungen und zugehörigen Testfunktionen, welche von den Operationen des Signed-Particle-Modells und notwendigen Partikel-Algorithmen inspiriert sind, werden zu einem kategorisch und rigoros konzipierten *Benchmark-Framework* kombiniert. Dieses Framework ist in der Programmiersprache C$^{++}$ geschrieben. Es wird verwendet, um numerische Experimente auf dem VSC auszuführen und Laufzeiten zu erhalten, welche dann analysiert werden. Das Ziel ist ein Software-Benchmark-Programm zu erstellen, welches die Bestimmung von Implementierungen von Partikel-Datenstrukturen ermöglicht, welche die beste Performance für die betrachtete Aufgabe bereitstellen. Obwohl die Laufzeitresultate am Ende dieser Arbeit eine große Anzahl an Datenstrukturdesigns beinhalten, von welchen eine Rangordnung der am besten passenden Implementierungen abgeleitet werden kann, liegt das Hauptziel in der Anwendbarkeit, Wiederverwendbarkeit, Instandhaltungsfähigkeit, Flexibilität und Erweiterbarkeit des Benchmark-Frameworks. Neue Datenstrukturen können definiert und bequem in das Framework eingefügt werden, um Laufzeiten für alle vorhandenen Testfunktionen zu ermitteln. Diese Konzepte unterstützen die Möglichkeit, die Applikation zu verwenden, um weitere Erkenntnisse zu erhalten, welche über die Implementierungen hinausgehen, die in dieser Arbeit vorgestellt werden und bekräftigen die Prinzipien von modernem Software-Engineering für *Computational Science and Engineering*-Anwendungen.

## Acknowledgement

First and foremost I want to thank **Prof. Josef Weinbub** for the supervision of this thesis. His guidance, suggestions and feedback throughout the preparation, creation and compilation of this thesis were essential for every chapter, section and subsection of this document. He secured me access to the newest version of the Vienna Scientific Cluster supercomputer, where I was able to perform a large number of numerical experiments with a magnitude of computational power that is normally far out of reach for a master student such as myself. This experience was truly exceptional and I am very grateful for this opportunity. Additionally I want to thank **Mihail Nedjalkov**, **Mauro Ballicchia**, **Clemens Etl** and **Philipp Haslhofer** for our weekly group meetings where I was able to obtain a deeper insight into the nature of the academic working field. The discussed topics reached from the preparation of scientific papers and the visitation of research conferences around the world to the experience of presentation hearings at the seminars organized by the Institute of Microelectronics. These meetings expanded my knowledge and increased my appreciation for the effort of reseach in general. Last but not least I want to mention the support from my father **Shahram**, my mother **Elisabeth** and my sister **Lisa-Mariam** that helped me to pursue my studies on the Technical University of Vienna. Without their financial and spiritual assistance I would have not been able to undertake the challenges of my chosen academic studies and therefore finish this very interesting phase of my life.

# Contents

# 1 Introduction and Overview

The objective of this thesis is an extensive quantitative performance analysis of dynamic particle data structures for signed-particle Wigner simulations [1, 2]. The impact of different data structure designs on the performance of large scale particle transport simulations is investigated. The problem is considered from the perspective of the modern computational scientist, which assembles knowledge from mathematics, numerics, physics and computer science to develop high quality computational software. This chapter gives an overview of the ideas and principles of computational electronics to establish a foundation for the main topics of this thesis. Modeling considerations, simulation aspects and solution techniques of the Wigner transport equation in the context of computational electronics are summarized. A short description of the Wigner signed-particle model is given to introduce the reader to the characteristic features of this approach. Parallel application programming interfaces that are utilized in the benchmark application are presented and their connection to the corresponding hardware computing units is described. Finally, the research task, the general objectives and the computational challenges of the thesis are summarized.

## 1.1 Transport Simulations in Computational Electronics

This section gives a short overview of computational electronics and positions it relative to the general field of computational science and engineering. The quantum Wigner transport equation, which governs the physical phenomena of interest, is introduced. The section is completed by mentioning stochastic techniques for the solution of the Wigner transport equation.

**1.1.1 Computational Science and Engineering (CSE).** Computational electronics is a subcategory of the field of computational science and engineering. Stevenson and Panoff [3] define this discipline as *„the interdisciplinary involvement in the identification and elimination of unwarranted assumptions and approximations in scientific models and the complete integration of computation into these models so as to constitute a whole new scientific technique on a par with hypothesis and experimentation“*. Most of todays researchers support the notion that CSE, while inherently an interdisciplinary field, represents a freestanding scientific discipline whose contents can be seen as self-contained. The concentration of known superior methods in one single scientific field can help to deminish extensive and unnecessary duplication of algorithm development efforts. Additionally it helps to reduce inconsistent and conflicting notation for similar objects and concepts in the various – otherwise often independently from each other – operating disciplines [4]. A key goal of CSE is to equip scientists of specific fields with computational tools that enable the possibility to explore their research topics in a more efficient way. They should not have to resort to non-physical approximations to reduce the model to a tractable and closed form [3]. The field is deeply interconnected with the principles of high performance computing (HPC). Cutting-edge HPC technology enables breakthroughs in CSE research and modern CSE applications are the main drivers for the evolution of HPC supercomputer and cluster systems. HPC and CSE are deeply intertwined and form a symbiotic relationship [5].

Higher performance and therefore faster computation times improve the research process of scientists in many ways. It enables accurate scientific modeling within much smaller time scales, speeds up the scientific feedback loop of evolving and revising the employed models and provides the possibility of wider exploration of the relevant parameter space [6]. Computational modeling, simulation and visualization are inevitable in situations where real life studies are impossible or highly dangerous. They allow us to acquire insights into problems which are too complex or difficult to study analytically, or too expensive, big or small to access experimentally [7]. An extensive list of research fields where CSE is applied can be found in [5, 7, 8].

**1.1.2 MODELING AND SIMULATION IN COMPUTATIONAL ELECTRONICS.** The main task of computational electronics is the development and use of simulation tools with the level of sophistication necessary to capture the essential physics, while at the same time minimizing the computational burden so that they can support the future development of electronic devices [9]. These tools are often summarized under the term technology computer-aided design (TCAD), which refers to the use of computer simulations by scientists and engineers in academia and industry to design, develop and optimize semiconductor processing technologies and device designs. The effective use of TCAD tools leads to minimized experimental time loss and reduction of the number of trial-and-error iterations, as well as reduced cost and resource consumption [10]. The characterization of semiconductor devices is part of device simulation, where the goal is to obtain devices with high performance, low power consumption, low cost and high reliability. Effects of slight process variations on physical and electronic device characteristics can be analyzed without the execution of usually hundreds of processing steps and the loss of months of processing time typical for semiconductor manufacturing [11]. The over many decades lasting and still continuing miniaturization of semiconductor devices, combined with the subsequent increase of integration density in microprocessors, led to minuscule device sizes in the range of a few nanometers. Todays processors easily contain many billion transistors whose ultra-small feature sizes require complicated and time-consuming manufacturing processes [9]. The scale of these devices increasingly demand adequate quantum transport models to capture quantum transport effects in the highly confined systems of, e.g., FinFETs, Spin-FETs, Tunnel-FETs, nanowires and nanosheets based GAA FETs [12]. The astonishing advances towards next generation technologies in the field of quantum electronics and subfields – such as electron quantum optics, quantum dots and quantum cascade devices – can only continue with the help of quantum transport simulations [13].

**1.1.3 WIGNER TRANSPORT EQUATION (WTE).** The WIGNER transport equation can be seen as a kind of quantum analog of the semi-classical BOLTZMANN transport equation. In its core it allows to describe the evolution of a WIGNER function $f_\mathrm{w}(\mathbf{r},\mathbf{k},t)$ [14] over time. The WIGNER function itself is defined over the phase space, same as the distribution function in the BOLTZMANN case, and describes the number of particles per unit volume at time $t$. The phase space $\{\mathbf{r},\mathbf{k}\}$ contains all possible combinations of the position $\mathbf{r}$ and the wavevector $\mathbf{k}$ that can be assigned to a particle [15]. This phase space formulation (instead of wave functions and operators) offers a more intuitive interpretation of quantum phenomena and provides a clear analogy to classical notions. However, the WIGNER function is not a true probability density function since it may attain negative values, which are a manifestation of the uncertainty relation in the phase space [15]. Regardless of the above statement, the critical property

$$\iint \mathrm{d}\mathbf{r}\mathrm{d}\mathbf{k}\, f_\mathrm{w}(\mathbf{r},\mathbf{k},t) = 1 \tag{1.1.1}$$

of a probability distribution is still retained, which means that physical averages still can be calculated using the same expressions as in the BOLTZMANN case, which classifies the WIGNER function as a so-called quasi-distribution function [15]. The WIGNER function is obtained by applying the WIGNER transform to the density matrix

$$\rho(\mathbf{r},\mathbf{r}',t) = \psi_t^*(\mathbf{r}')\,\psi_t(\mathbf{r}) \tag{1.1.2}$$

which yields

$$f_\mathrm{w}(\mathbf{r},\mathbf{k},t) = \frac{1}{(2\pi)^3}\int \mathrm{d}\mathbf{r}'\,\mathrm{e}^{-\mathrm{i}\mathbf{k}\cdot\mathbf{r}'}\psi\left(\mathbf{r}-\frac{\mathbf{r}'}{2}\right)\psi^*\left(\mathbf{r}+\frac{\mathbf{r}'}{2}\right). \tag{1.1.3}$$

Similarly, a Fourier transform of the Liouville-Von Neumann equation

$$\mathrm{i}\hbar\,\partial_t\rho = \left[\mathcal{H},\rho\right] \tag{1.1.4}$$

which describes the evolution of the density matrix and where $\left[\mathcal{H},\rho\right] = \mathcal{H}\rho - \rho\mathcal{H}$ denotes the commutator bracket, $\hbar$ the reduced Planck constant and $\mathcal{H}$ the Hamiltonian operator, yields the evolution equation for the associated Wigner function, the WTE (shown here for the electrostatic case and in the absence of scattering)

$$\frac{\partial}{\partial t}f_\mathrm{w}(\mathbf{r},\mathbf{k},t) + \frac{\hbar\mathbf{k}}{2m^*}\frac{\partial}{\partial\mathbf{r}}f_\mathrm{w}(\mathbf{r},\mathbf{k},t) = \int \mathrm{d}\mathbf{k}'\,V_\mathrm{w}(\mathbf{r},\mathbf{k}'-\mathbf{k},t)\,f_\mathrm{w}(\mathbf{r},\mathbf{k}',t) \tag{1.1.5}$$

where

$$V_\mathrm{w}(\mathbf{r},\mathbf{k}'-\mathbf{k},t) = -\frac{1}{\mathrm{i}\hbar\,(2\pi)^3}\int \mathrm{d}\mathbf{s}\,\mathrm{e}^{\mathrm{i}\mathbf{s}\cdot(\mathbf{k}'-\mathbf{k})}\left\{V\left(\mathbf{r}+\frac{\mathbf{s}}{2}\right) - V\left(\mathbf{r}-\frac{\mathbf{s}}{2}\right)\right\} \tag{1.1.6}$$

denotes the Wigner potential. Simulation domains require finite dimensions and therefore impose bounds on the integration of the variables. A finite value of the isotropic coherence length $|\mathbf{L}| = L$ yields a discretization of the momentum space $\mathbf{k}$. The semi-discrete Wigner function, equation and potential can be obtained by applying a discrete Fourier transform, which yields

$$f_\mathrm{w}(\mathbf{r},\mathbf{q}\Delta k,t) = \frac{1}{L}\sum_{\mathbf{q}}\mathrm{e}^{-\mathrm{i}\mathbf{q}\Delta k\cdot\mathbf{s}}\rho\left(\mathbf{r}+\mathbf{s},\mathbf{r}-\mathbf{s},t\right), \tag{1.1.7}$$

$$\left[\frac{\partial}{\partial t} + \frac{\hbar\mathbf{q}\Delta k}{m^*}\nabla_\mathbf{r}\right]f_\mathrm{w}(\mathbf{r},\mathbf{q},t) = \sum_{\mathbf{q}}V_\mathrm{w}(\mathbf{r},\mathbf{q}-\mathbf{q}')\,f_\mathrm{w}(\mathbf{r},\mathbf{q}',t) \tag{1.1.8}$$

and

$$V_\mathrm{w}(\mathbf{r},\mathbf{q}) = \frac{1}{\mathrm{i}\hbar L}\int_{-L/2}^{+L/2}\mathrm{d}\mathbf{s}\,\mathrm{e}^{-\mathrm{i}2\mathbf{q}\Delta k\cdot\mathbf{s}}\left\{V\left(\mathbf{r}+\mathbf{s}\right) - V\left(\mathbf{r}-\mathbf{s}\right)\right\} \tag{1.1.9}$$

where $\mathbf{q}$ is an integer multi-index and $\Delta k = \pi/L$, which denotes the resolution of the discretized wavevector [15].

**1.1.4 Advantages and Applications of the WTE.** The Wigner function is defined over the phase space, which allows the semi-classical scattering models from the Boltzmann transport equation to be incorporated into the formalism. It can be shown that for both phonon and impurity scattering the semi-classical models can be obtained as a limiting case of the full quantum models, which opens the possibility to form a hierarchy of scattering models with different levels of accuracy, from which the best suited model, depending on the computational problem, can be chosen [15]. The combination of the Wigner equation with the Boltzmann scattering mechanisms results in the Wigner-Boltzmann equation, which unifies the two theories and ensures a seamless transition between purely quantum (ballistic) and classical (diffusive) transport. This means that the Wigner formalism can be used for a semi-classical description of the extended contact regions and simultaneously for the quantum description of the active region of a device [16]. Additionally, the

WIGNER function finds wide application in a large number of research fields outside of computational electronics, such as quantum physics, quantum optics and quantum information processing [2, 17]. Another use case are applications of computational chemistry, where the formalism can be extended to the so-called WIGNER density functional theory. Single- and many-body problems such as atoms, molecules and systems of two identical Fermions can be investigated [18].

**1.1.5 STOCHASTIC SOLUTION TECHNIQUES FOR THE WTE.** Already in the semi-classical BOLTZMANN case, the high-dimensionality of the BOLTZMANN equation complicates deterministic solutions. The curse of dimensionality leads to simulations that require enormous memory consumption and execution times, therefore stochastic approaches such as the Monte Carlo method are utilized to solve the BOLTZMANN equation. The Monte Carlo algorithm consists of the simulation of particle drift motions, where the free flight times are generated randomly, and also of randomly selected scattering mechanisms, which change the final energy and momentum of the particle. These particle ensemble trajectories can be sampled at various points in time throughout the simulation to statistically estimate the desired physical quantities [9]. If rare events have to be simulated or the distribution function is needed only in a small phase space domain, the so-called Backward Monte Carlo method (BMC) is used, where the simulation followes the particle trajectories in reverse direction back in time [16]. The basis of stochastic solution techniques for the WIGNER transport equation is also the Monte Carlo method, which has been inspired by the great success of the application to the BOLTZMANN transport equation [17]. The association of trajectories to a single or an ensemble of particles is used here as well, either in the way of WIGNER trajectories, which are defined by a quantum force formalism, or by WIGNER paths, where the action of the WIGNER potential operator is interpreted as scattering. Another approach is to map all the information of the quantum state of the system onto the amplitude of DIRAC excitations in the phase-space, which all summed up represent the WIGNER function. These amplitudes are called affinities, therefore this method is known as the affinity model [19]. The affinities are updated by the WIGNER potential during the particle evolution and can be represented by positive or negative values, which act as weighting factors in the reconstruction of the Wigner function and consequently in the computation of all physical averages. The signed-particle model is a modified version of the affinity model, based on the alternative interpretation of the WIGNER potential as a generator of signed particles [1]. The signed-particle method only considers integer affinities with the values $+1$ and $-1$, in all other aspects the evolution of the particle is field-less and classical. If two particles with opposite sign meet in the same discretized cell of the phase space, they annihilate each other, since they have an equivalent probabilistic future, but make an opposite contribution in the process of averaging [15].

## 1.2 WIGNER SIGNED-PARTICLE SOLUTION APPROACH

The signed-particle model is – as already stated above – a variation of the affinity model, where only integer affinities with the values $+1$ and $-1$ are considered. The generation and annihilation events, which occur frequently during the simulation, lead to peculiarities regarding the implementation and parallelization of this algorithm. This section describes considerations regarding the distributed memory implementation and an overview of the structure of the algorithm. The summary of these characteristics will become important when they are compared to the simplified test functions of the benchmark framework in Section 3.4. This section follows, unless otherwise stated, the description of ELLINGHAUS [15].

**1.2.1 DISTRIBUTED MEMORY CONSIDERATIONS.** Semi-classical Monte Carlo codes are straight forward to parallelize, since the particles of an ensemble are independent of each other and, therefore, the subensembles on each parallel process/thread can be calculated in a self-consistent manner. The parallelization is more difficult for Wigner Monte Carlo codes, because here the critical annihilation step must be performed in a synchronized manner to uphold statistics. This leads to synchronized communications during the execution. In this section the parallelization structure of the ViennaWD [20] reference implementation is shortly discussed, which uses a MPI-based domain decomposition approach (see Section 1.3.2) on a distributed memory architecture (see Definition 5). The global simulation domain is split into uniformly sized subdomains, as well as the entire particle ensemble into subensembles. Every MPI process (see Definition 7) handles a subdomain and the subensemble that is currently located in this subdomain. The phase space is decomposed in the spatial domain, but not in the momentum domain. The reason behind this decision lies in the nature of the scattering events: While in a generation event all new particles spawn in the same position as the already existing particle, the wave vector of a scattered particle would most certainly lie in a part of the **k**-space which would be represented on another process and would, therefore, increase the communication demand between the processes.

**1.2.2 The Solution Process.** The basic steps of the solution process are the initialization of the particles, the time loop, which consists of evolution, growth prediction, annihilation, and particle transfer steps – which are executed alternately until the total simulation time is reached – and finally the post processing step. The structure of the algorithm as a flowchart is shown in Figure 1.1. A histogram records the position, wavevector and sign of each particle in the ensemble at regular time intervals $\Delta t$. This procedure approximates the distribution function $f_\mathrm{w}(\mathbf{r}, \mathbf{k}, t)$.

- **Initialization.** First the inputs describing the geometry, potential profile and parameters are loaded by the master process (see Definition 8). The master process initializes an ensemble of $N$ particles representing the initial condition of the evolution problem by setting the corresponding position and momentum values. Then the master process distributes the particles to the worker processes according to the domain decomposition, together with the potential profile and further global parameters. Each process then initializes localized versions of the required data structures with the received initial values specific to its subdomain. An often used initial condition is the Gaussian minimum uncertainty wavepacket

$$f_\mathrm{w}(\mathbf{r}, \mathbf{q}) = \mathcal{N} \exp\left[-\frac{(\mathbf{r}-\mathbf{r}_0)^2}{\sigma^2}\right] \exp\left[-(\mathbf{q}\Delta k - \mathbf{k}_0)^2 \sigma^2\right] \tag{1.2.1}$$

  where $\mathbf{r}_0$ denotes the mean position, $\mathbf{k}_0$ the mean wave vector, $\sigma$ the standard spatial deviation and $\mathcal{N}$ a normalization constant. After this step the time loop starts which tracks the trajectories of all particles in the ensemble over the simulation time.

- **Evolution.** Each process performs the evolution of its ensemble of particles for a single time step for itself, without any communication involved. During the drift phase the particles are following a Newtonian trajectory according to the laws of classical mechanics, where the particles do not accelerate due to forces and the wave vector remains constant since there appears no explicit force term in the WTE. The position is calculated by

$$\delta\mathbf{r} = \frac{\hbar(\mathbf{q}\Delta k)}{m^*} \min\left\{\tau, \delta t\right\} \tag{1.2.2}$$

  which means the particle drifts either until the current time step ends ($\delta t$ denotes the remaining time in the time step $\Delta t$) or until the next scattering event happens, which is represented by the free-flight time $\tau$. The value of $\tau$ is determined by the generation of an uniformly distributed

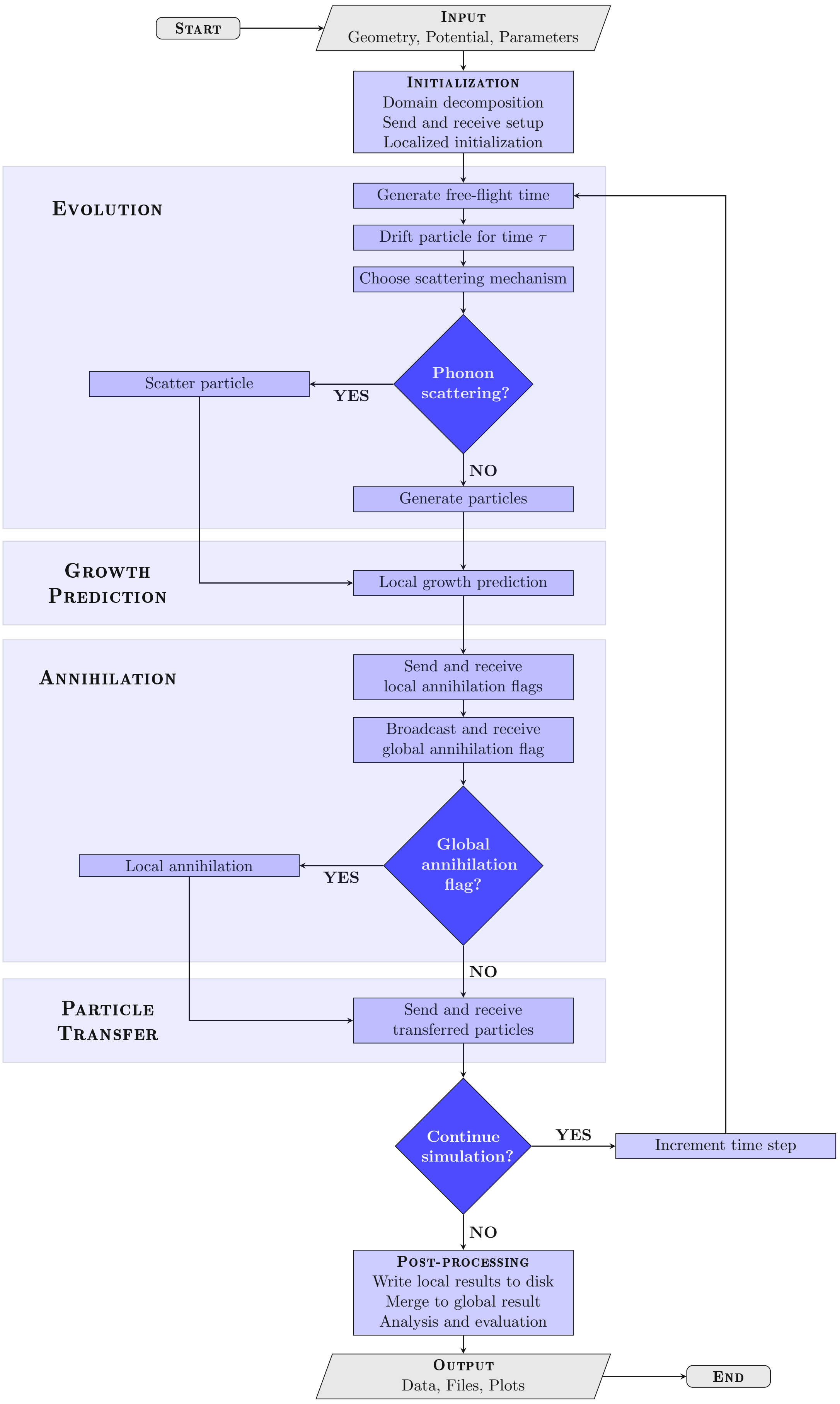

**Figure 1.1** Flowchart of ViennaWD, based on Ellinghaus [15].

random number $r$. If the free-flight drift is finished, a scattering event occurs, which can either be phonon scattering or a particle generation event. The selection is again determined by the generation of a random number $r$ from a normalized scattering table. If a generation event is selected, two additional particles with signs $\pm 1$ and wave vectors $\mathbf{k} \pm \ell$ are created, where the offset $\ell$ is determined by the WIGNER potential. If a scattering event (e.g., phonons) is selected, the wave vector of the particle is modified according to the selected scattering mechanism. The processes of drift and scattering are repeated in an iterative fashion for all particles in the ensemble until the end of the time step $\Delta t$ is reached.

➤ **GROWTH PREDICTION.** The process of particle generation leads to an exponential increase in the number of particles, simulations easily become computationally infeasible. Consequently, an annihilation step has to be executed to reduce the number of particles. However, this annihilation procedure is only performed when the number of particles exceeds a specific maximum in the next time step, which reduces the computational burden and prevents undesired numerical effects. To determine if this maximum will be exceeded in a subsequent time step, each process performs a growth prediction for its subensemble of particles after the evolution step. It is advisable to overestimate the particle increase, therefore, the maximum value of the generation rate $\gamma$ for all particles is used, which yields an upper bound on the particle growth and the predicted number of particles in the next time step as

$$N_{t+\Delta t} = N_t \left(1 + \max_i \{\gamma(\mathbf{r}_i)\} \Delta t\right). \tag{1.2.3}$$

➤ **ANNIHILATION.** All processes have to perform their local annihilation step for the particles in the subdomain at the same time step, otherwise the global statistics of the WIGNER function would be falsified. This means if even one single process requires an annihilation step according to its local growth prediction, all other processes have to perform one too. To ensure this synchronized behaviour the result of the growth prediction of every process is communicated to the master process in the form of an annihilation flag. The master process collects all flags and checks if at least one of them is true. If this is the case, a positive global annihilation flag is broadcasted and all processes perform an annihilation step. Otherwise a negative flag is communicated, therefore, no annihilation takes place. The actual annihilation is executed on the basis of phase space cells. The phase space is divided into cells, where each cell is associated to a specific value of the wave vector, since the semi-discrete WIGNER equation (1.1.8) is used. If two particles with opposite signs are located in the same cell, they annihilate each other and are removed from the ensemble. This can be done because all the particles within a cell are deemed to be identical and indistinguishable, therefore, two particles with opposite sign would make the same contribution to the WIGNER function, just with an inverted magnitude. After the annihilation, the remaining particles are regenerated and the time loop continues.

➤ **PARTICLE TRANSFER.** After the annihilation step each process checks if particles are located in the overlapping boundaries of the different subdomains. If this is the case, the particles are collected and sent to the adjacent processes. Particles are also received from other processes. A synchronization barrier is used to ensure all transfers are complete before the next time-step commences. The reason for performing the transfer after the annihilation is that after an annihilation step the size of the particle ensemble will be significantly smaller, consequently the number of particles to be transferred will have been reduced, which is beneficial for the communication burden.

➤ **POST-PROCESSING.** The above described steps are repeated until the total simulation time has been reached. To avoid a central communication bottle neck at the end of the simulation, there is no global reduction step issued by the master process to collect the resulting data. Instead, the simulation results of each subdomain are written to disks locally by each process. After the merging of the simulation results, handled by external scripts, different post-processing steps such as analysis, examination and evaluation can take place.

## 1.3 PARALLEL COMPUTING DEFINITIONS

One of the computational challenges of stochastic solution techniques is the enormous number of numerical particles in the simulated particle ensemble, which guarantees that the variance of the resulting physical quantities of interest is as small as possible (see Section 1.1.5). This number lies in the here considered case in the range of $10^6$ to $10^8$ particles. In combination with the necessary operations of the WIGNER signed-particle model (described in Section 1.2), it demands a parallel solution scheme, as previously hinted, to realize practically relevant simulation runtimes. Therefore, parallelization techniques have to be employed to boost the performance of the application. This section establishes a number of important definitions as a reference for later chapters and describes the connections between the computer architecture terms *cluster*, *node*, *socket* and *core* and the concepts of *processes* and *threads* that are essential for the utilized parallel APIs. The way how these components are assigned to each other (sometimes called *pinning* or *affinity* [21]) is crucial for understanding the different configurations (see Section 4.1.2) of the benchmark results in Chapter 4. The section first mentions the hardware aspects and definitions, followed by the parallel APIs that are relevant for this thesis. At last, a few remarks regarding the execution time as a general measurement of performance are given.

**1.3.1 HARDWARE.** Large-scale supercomputers designed for high-performance computing tasks are composed of a hierarchy of elements, which in turn can be classified by a number of technical terms. The definitions of these terms, as far as they are used in this thesis, are now presented in a top-down manner. The complete system, the supercomputer, is often also called a *cluster*.

**DEFINITION 1** (Cluster). *A system that consists of a large number of connected computing units is called a* cluster*. The computing units are controlled and scheduled by specific software and connected to each other through fast local area networks.*

The individual computing units of a cluster are called *nodes*.

**DEFINITION 2** (Node). *A* (compute) node *is the building block of a cluster. It usually possess its own memory block. Nodes contain at least one CPU and potentially a number of additional accelerators (e.g., GPUs). If a combination of CPUs and accelerators are present on a node, it is called a* hybrid *node.*

The sweetspot of compute node configurations with respect to cost vs. performance is often a two-socket setup, that is, a system offering two CPUs [21]. The CPUs are connected via a coherent link and each CPU has its own locality domain (NUMA effects, see Definition 6).

**DEFINITION 3** (CPU). *A* CPU (central processing unit) *is an integrated circuit that reads and executes program instructions. While in a traditional single-core design the terms CPU and* core *can be used synonymously, modern multicore CPUs contain several cores which execute code concurrently. They share resources like memory interfaces or caches to varying degrees, dependent on the chip design.*

**DEFINITION 4** (GPU). *A* (GP)GPU (general-purpose graphics processing unit) *is an accelerator card* *that is designed to perform certain classes of parallel computations particularly efficient. It typically contains an order of magnitude more parallel computing units (compared to a CPU) and is typically connected to the node via the peripheral component interconnect express bus, which allows communication between the CPU and the GPU* [22].

The hardware components above can also be classified by the type of their memory system. For this thesis two types are important, the *distributed* and the *shared memory system*.

**Definition 5** (Distributed memory system)**.** *A* distributed memory system *is a group of computing units where each unit is connected to exclusive local memory, which means that only the specific unit has direct access to it and all other units are not able to fetch data from this memory block. As there is no remote memory access on these systems, data transfer has to be executed cooperatively by sending messages back and forth between the units* [21].

**Definition 6** (Shared memory system, UMA, ccNUMA)**.** *In a* shared memory system*, all computing units work on a common, shared physical address space. There are two varieties possible.* Uniform Memory Access (UMA) *systems provide the same latency and bandwidth for all computing units and all memory locations. UMA systems are from a hardware perspective not scaleable, since the bandwidth decreases when the number of sockets increase. The memory of* cache-coherent Nonuniform Memory Access (ccNUMA) *machines is logically shared, but physically distributed. This leads to varying memory access performance, depending on which computing unit accesses a particular part of the memory. Memory of ccNUMA systems is partitioned in so-called locality domains (sometimes called memory domains) that locally resemble UMA building blocks and are linked via a coherent interconnect* [21].

Since the system of all nodes combined represents a distributed memory machine, the cluster can be classified as a so-called *hierarchical hybrid system* that allows for different layers of parallelism. While the VSC-4 cluster [23] employs nodes with CPUs only, the VSC-5 cluster [24] also contains hybrid nodes with both CPUs and GPUs (see also Section 4.1.1). These characteristics are represented by the names of the build options of the benchmark framework in Section 3.2.5.

### 1.3.2 Software.

Modern computing hardware allows the developer to manage multiple computing units in parallel with the help of specific application programming interfaces (APIs), provided by software libraries that can be included in the source code of the application to utilize the offered functionality. Some of them are based on international standards which ensure (to a large degree) the same behaviour of the library functions, independent of the concrete – often open source – implementation. Others are developed and maintained by corporate vendors, where the code is classified as closed source and not available to the public. The following APIs are relevant for the benchmark framework described in Chapter 3.

- **Message Passing Interface (MPI).** If a plethora of distributed computing units which are partly connected via a network are utilized to perform computations in parallel, they need a way to communicate with each other. The preparation, post processing and updating of the data between time steps of the here considered stochastic solution method necessitates methods to transfer the data between the computing units. If a distributed memory system is used, the computing units cannot access the same memory, therefore, the data has to be explicitly sent and received between the accessable memory of the individual parallel units. The Message Passing Interface (MPI), an international standard [25], is a communication layer for so-called *processes* and is provided by different software libraries.

  **Definition 7** (Process)**.** *Executing computer programs consist of instances called* processes *and can contain at least one* thread (see Definition 9)*. Processes manage the scheduling of* *operations and the allocation of processor resources* [22]. *Processes don't have direct access to the memory block of another process, calling out for explicit communication to realize data transfer, as previously hinted.*

Each process is assigned to a distinct index inside the MPI communicator, the *rank*. Depending on the problem at hand, one or more processes are considered for each node. For instance, compare a pure MPI approach, where typically as many processes are created as there are cores available on the node's CPU(s), with a hybrid approach, where, for instance, one MPI process is spawned on each node and the remaining cores are utilized with a shared-memory parallelization approach. Regardless, a straightforward basic approach to initialize a MPI program and distribute the workload among the available processes is the *master-worker model*.

**DEFINITION 8** (Master-worker model)**.** *Usually one specific process out of all spawned processes has a unique task in parallel applications. This process is called the* master, *the other processes are called* workers. *The master, e.g., manages the input and output, sends instructions and receives results from the workers and performs reduction operations at the end of a simulation. The workers on the other hand execute the desired operations in parallel during the simulation* [22].

The master-worker model is also used in the ViennaWD reference implementation (see Section 1.2.2) (and the ParticleStackBenchmark application, see the later discussion in Section 3.4.8). As previously indicated, a hybrid parallelization approach is sometimes used, e.g., using MPI for inter-node communication and a shared-memory model (such as OpenMP) for intra-node parallelization.

➤ **OPEN MULTI-PROCESSING (OPENMP).** As previously discussed, each process can have at least one thread. When more than one thread is being used, possibilities open up for thread-based parallelization by making use of the shared memory system. In this case the Open Multi-Processing (OpenMP) library can be utilized, which is – similar to the MPI library – based on a standardized specification [26] that ensures consistent behaviour. OpenMP spawns so-called *threads* on the CPU, where the library assignes each thread to a core.

**DEFINITION 9** (Thread)**.** *A* thread *of execution presents another level of parallelism control within a process and can be seen as the smallest sequence of programmed instructions that can be managed independently by a scheduler* [22]. *Usually one thread is executed per core.* Simultaneous multithreading *(*SMT*, sometimes also called* hyperthreading*) utilizes multiple architectural states in the core and allows for the execution of multiple instruction streams in parallel and, therefore, allows for multiple threads to be executed on a single physical core (typically 2 threads per core)* [21].

Considering shared memory programming, with OpenMP the fundamental programming model is the *fork-join model*.

**DEFINITION 10** (Fork-join model)**.** *If a region of a sequential algorithm, executed by a thread, can be parallelized, a so called* parallel region *is generated. In the* fork *phase, a number of threads are created. These threads now perform the work concurrently until all threads are finished. Then in the* join *phase, the results of those concurrent operators are accumulated into a single resulting operator. The process is then executing in sequential order until a new parallel region can be generated* [22].

In contrast to the master-worker model of MPI processes, the fork-join model of OpenMP threads does not necessarily select a specific thread for managing tasks. All threads inside the parallel region are working on the same task concurrently. All OpenMP instructions are realized by preprocessor directives, which are often single line commands, indicated by the `#pragma omp` directive. A large number of different operations are available, for example one possibility to achieve data parallelism is through SIMD (Single Instruction Multiple Data) instructions, which are part of instruction set extensions of modern cache-based processors. They issue identical operations on a large number of arguments of the same type to perform concurrent executions of arithmetic operations in special wide registers [21].

➤ **Compute Unified Device Architecture (CUDA).** Another type of processors that give the developer the possibility to utilize parallelism techniques on the software level is the graphics processing unit (see also Definition 4). Historically these processors were used for efficient manipulation of computer graphics and image processing algorithms. Although they are still used today for these tasks, recent improvements in the hardware architecture allow to use GPUs in all sorts of applications where high performance is in demand. These tasks are often summarized under the term general-purpose computing on graphics processing units (GPGPU) [27]. The advantage of using a GPU instead of a CPU for specific, suitable parallel problems is the fact that GPUs possess a much larger number of computing units than CPUs, which of course leads to a beneficial effect regarding the performance if the parallel problem can be optimally fitted to the GPU. If not only CPUs, but also other processor architectures such as the GPU are available on a compute node, we need libraries that offer the functionality to communicate between the CPU (in this context called the *host*) and GPU (in this context called the *device*) and to develop source code that can be executed on the GPU. The native toolkit for NVIDIA acceleration cards is the Compute Unified Device Architecture (CUDA), a C$^{++}$ language extension and maintained by NVIDIA itself [28]. The library offers functionality for the allocation of memory on the GPU and methods to send and to receive data between the host and the device. Functions that should run on the device possess a specific syntax and are called *kernels*. The existence of CUDA has simplified the development of scientific software significantly and is one of the most important reasons why the GPU architecture has found its way into the world of general purpose applications.

➤ **Heterogeneous-Computing Interface for Portability (HIP).** Besides NVIDIA, the currently most prolific vendor of GPU acceleration cards, other companies provide similar products. One of them is Advanced Micro Devices (AMD), which maintaines the Radeon Open Compute (ROCm) framework, a software stack for GPU programming. It provides an alternative GPU library to CUDA, which includes a tool to transform CUDA code files to general portable source code files, which can be compiled on both the NVCC compiler by NVIDIA and the HCC compiler by AMD. This tool is called Heterogeneous-Computing Interface for Portability (HIP) [29]. The syntax for both CUDA and HIP are almost identical (where mostly the `cuda` term has to be substituted by `hip` in the source code), which is part of the intention to convince software developers to switch to the new alternative.

**1.3.3 Measurement of Performance.** Since this thesis compares the performance of different data structures, a few remarks regarding the measurement of the performance should be made. Two fundamental quantities for performance are the *floating-point operations per second* [Flops/Second], which measures the number of add and multiply operations on floating-point data per time unit, and the *bandwidth* [Bytes/Second], which measures the amount of data that is transferred between the caches and main memory per time unit [21]. For general benchmarking, the straightforward *execution time* [Seconds/Program] (sometimes also called *elapsed time* or *wall clock time*) is used, which measures the time between the start and completion of a specific task. The execution time is inversely proportional to the performance. This time adds (in contrast to the *CPU time*) also the contributions from disk accesses, memory accesses, input/output activities and operating system overhead to the amount of time that is needed to complete the task [30]. The execution time can be affected by three key factors, namely the number of instructions executed by the program (called *instruction count*), the average number of clock cycles per instruction (sometimes abbreviated as CPI) and the number of seconds per clock cycle (called *clock cycle time*). If these three parameters are combined

$$\text{Execution time} = \frac{\text{Instructions}}{\text{Program}} \cdot \frac{\text{Clock cycles}}{\text{Instruction}} \cdot \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} \tag{1.3.1}$$

we obtain the already established unit of seconds per program. The only complete and reliable measure of computer performance is time. When comparing the performance, one must look at all three components, which combine to form the execution time [30]. There are different software libraries available that offer functions for the measurement of the elapsed time by a wall clock, for example the `std::chrono::system_clock` class from the `<chrono>` header in the C++ standard library, the `omp_get_wtime()` function from the OpenMP library and the `MPI_Wtime()` function from the MPI library. The last option was utilized for the ParticleStackBenchmark application (see Section 3.4.7).

## 1.4 Thesis Overview and Outline

The computational scientist views a computational problem from different perspectives. The scientist has to find answers for diverse questions and has to make decisions based on given characteristics and limitations. This introductory chapter discussed the engineering applications of computational electronics in general and the WIGNER signed-particle model in particular, as well as the description of the affinity between the utilized hardware components and the parallel application programming interfaces. With this preliminary information in mind, the general objectives of this thesis can be treated.

**1.4.1 General Objectives.** As previously mentioned, particle WIGNER simulations are computationally highly challenging. Particularly, the optimal choice of the data structure for the significant particle stack sizes (e.g., $> 10^7$) is unknown. The primary goals of this thesis are thus:

➤ **Selection of Promising Data Structures.** The enormous number of simulated numerical particles and the necessary dynamic memory management demand data structures with appropriate memory layouts and specific access methods. The relation between the data structures (also called *particle stacks*) on the one hand and the possible parallelization techniques, the communication and memory access patterns and the algorithmic time complexities on the other hand should be analyzed in detail to find a selection of promising data structures that can then be benchmarked.

➤ **Development of Flexible Benchmark Framework.** Different implementations of potential data structures and corresponding test functions, which are inspired by the operations of the WIGNER signed-particle model, should be combined into a benchmark framework. The goal is to develop a benchmark tool that is able to determine data structure implementations which offer the best performance for the desired computational task. The objective lies in the usability, reusability, maintainability, flexibility and expandability of the benchmark framework. New data structures can be defined and easily incorporated into the framework to obtain execution runtimes for all available test functions. This feature facilitates the so-called inversion of control property, where the user customizes specific framework components and the framework itself undertakes the actual execution of the application [31].

➤ **Identification of Optimal Data Structures.** The benchmark framework should be used to perform numerical experiments and yield execution times which are compared and examined. The runtime results should feature a variety of diverse data structure designs from which a hierarchy of best suited particle stack implementations can be derived. Optimally the detailed investigations should yield one specific data structure that performes superior than the others under the vast number of circumstances that are tested by the framework.

**1.4.2 Outline of the Thesis.** This thesis is composed of five chapters, where the first chapter presented the reseach topics of interest by giving an overview of the important aspects and referencing the existing literature. The four following chapters describe the undertaken research, as well as the obtained results and insights. Chapter 2 analyzes the design of data structures in the context of particle transport simulations. Theoretical aspects such as algorithmic time complexity are investigated and used for the determination of promising particle stack containers. Also data structure implementations of related open source Monte Carlo particle simulators are analyzed and compared to the theoretical principles. Chapter 3 describes the software structure of the developed benchmark framework, which is written in the C++ programming language. The utilization of modern programming paradigms in combination with software engineering techniques to yield flexible and expandable software is illustrated. All components of the benchmark application and their interactions and dependencies are described in detail. Chapter 4 presents the surrounding conditions regarding benchmark execution on clusters and the obtained execution runtimes, augmented with discussions and interpretations of the output data. Chapter 5 summarizes the content and the findings of the thesis. Future extensions are discussed and an outlook for further investigation possibilities is given. The final conclusion gives a statement about the insights that were obtained from the analysis of the benchmark runtimes.

# 2 Data Structure Design

The research behind data structures – combined with algorithms – is one of the corner stones of the field of computer science in general. The organization of data in memory and the corresponding access methods are extremly important with regards to performance optimization. This chapter investigates the selection of the data structures that are contenders for the implementation of Wigner signed-particle simulators. First the inner components of the data structures – in the following denoted as *containers* – have to be selected. This is done with the help of the theory of algorithmic time complexity. Then the arrangement between these containers has to be examined. For this task other approaches used in available open source simulators are considered and evaluated. Lastly the chosen data structures are discussed in more detail. These data structures are implemented in the ParticleStackBenchmark framework, which will be described in Chapter 3.

## 2.1 Algorithmic Complexity Measures

The first question that has to be answered is the choice of the containers that will hold all particle information. These containers will then be combined in different memory layouts to yield data structure concepts that can be used inside the benchmark application. Strictly speaking, the containers themselfs are also data structures. Therefore we can utilize the theory behind algorithmic time complexity to compare the efficiency of the methods of different containers. This section first defines the idea behind time complexity, emphasises the methods that are important for the algorithm at hand and then compares a number of data structures with the desired methods with the help of the established theory. This procedure will yield the best suited containers for our endeavor.

**2.1.1 Algorithmic Time Complexity.** The idea behind algorithmic time complexity is to find a clear notation for the efficiency of algorithms in a general sense. The notation should be dependent on the amount of data that is processed during the execution of the algorithm, since this quantity is proportional to the executed operations. In our case the algorithms of interest are represented by methods of data structures that modify the data members inside them. First we give a definition for the utilized notation.

**Definition 11** (Asymptotic notation). *Let $A(n)$ and $f(n)$ both be real and positive valued functions of the size $n$ (which represents the input in units of bits or elements), where $A(n)$ denotes the time complexity of a given algorithm and $f(n)$ describes a specific time complexity class. The* asymptotic notation *$A(n) = \mathcal{O}\{f(n)\}$ represents the condition that $A(n)$ is at most a positive constant multiple of $f(n)$ if $n \to \infty$. This is the case if and only if there exists a positive real number $M$ and positive integer $n_0$ such that $A(n) \leq Mf(n)$ for all integers $n \geq n_0$* [32].

Although the time complexity represents a continuous measurement tool, it is advantageous to define a number of complexity classes that are distinct enough from one another so that the identification and assignment of a time complexity class to a specific algorithm yields a practical estimation of the efficiency. We will need the following time complexity classes in this section.

**Definition 12** (Time complexity classification). *The time complexity of order* $\mathcal{O}\{1\}$ *is called* constant, *the order* $\mathcal{O}\{\log(n)\}$ *is called* logarithmic *and the order* $\mathcal{O}\{n\}$ *is called* linear [33]. *This sequence of complexity classes is sorted in increasing order, which means that* $\mathcal{O}\{1\}$ *is the best achievable complexity,* $\mathcal{O}\{\log(n)\}$ *is desirable and* $\mathcal{O}\{n\}$ *is the worst possible (from the here presented) complexities.*

These are just a small selection of the large classification system that is available, but since only these orders appear in Table 2.1, the others are omitted. These definitions give us a tool to determine in a fast and efficient way which data structures are performing the best and which should be avoided for an actual implementation.

**2.1.2 Data Structure Interface.** Different data structures are appropriate for different applications. Here we distinguish between two types of data structures, namely the *sequence* and the *set* data structures, where both of them have a specific interface [34]. The term interface denotes in this context the supported methods to modify the contained data.

- **Sequence Data Structures.** These data structures organize their data in a sequential way, where the data can be, but does not have to be, contiguously stored inside the memory. The idea behind the sequential structure is the assignment of a specific index to every element, which can be used as an enumeration. This functionality is often implemented with the help of a specific index operator and guarantees the possibility of random access to and subsequent modification of all elements of the data structure. Examples for these data structures are arrays, lists and specific tree structures. Important methods of this type are the `setAt()`, `getAt()`, `insertAt()` and `deleteAt()` functions.

- **Set Data Structures.** These data structures are convenient for the sorting and finding of specific data of interest. Often the elements of the data structures are unique, which means that every element has a unique value, but this does not necessarily have to be the case. The organization of the data is not based on a sequence, but on more sophisticated methods, such as key-value pairs. Examples for these data structures are sorted arrays, hash tables and specific tree structures. Important methods of this type are the `sort()` and `find()` functions, often with specific use cases such as `findMin()`, `findMax()`, `findPrev()` and `findNext()`.

The description of the Wigner signed-particle method in Section 1.2 shows us which methods are the most useful for a data structure that should be used for the implementation of a particle stack. In every time step of the time loop, all particles have to be modified, for example the position and momentum values. Additionally scattering events lead to the modification of specific elements in the particle stack, which is best supported by random access functions such as `setAt()` and `getAt()`. Generation events create new particles, which can be added by the `insertAt()` function and annihilation events erase specific particles, which again need random access functions such as `deleteAt()`. On the other hand functionality such as `find()` and `sort()` are not essential in our considered application case, since the particles cannot really be sorted in a fashion that would be beneficial for the solution process. Even scenarios where for example all particles with positive annihilation flags would be sorted to the end of an array for faster deletion are not really beneficial, since the sorting algorithm would have to iterate over the elements in one way or another as part of the method, and this iteration alone would suffice for the deletion of the particles. The considerations in this section lead to the conclusion that sequence data structures are better suited for the Wigner signed-particle method. Therefore, only data structures of this type are investigated and considered as contenders for particle containers below.

### 2.1.3 Data Structure Comparison.

Now we use the time complexity measure to compare different data structures with a sequential interface to find containers that are useful for the Wigner signed-particle solution approach. The results can be found in Table 2.1.

Table 2.1 Time complexity of methods of interest for different data structures with a *sequential* interface. The variable `n` determines the index of the element in the data structure and `E` represents the element data. The number of elements of a data structure is denoted by $n$ and the height of a tree data structure by $h$. Complexities with the exponent $\mathcal{O}\{\cdot\}^{A}$ mean *amortized* time costs. The time complexities are taken from the information in [34, 35, 36].

| **Data Structure** | **Random Access** | | **First and Last Element** | |
|---|---|---|---|---|
| | `setAt(n,E)` `getAt(n)` | `insertAt(n,E)` `deleteAt(n)` | `insertFirst(E)` `deleteFirst()` | `insertLast(E)` `deleteLast()` |
| Dynamic Array | $\mathcal{O}\{1\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{1\}^{A}$ |
| Linked List | $\mathcal{O}\{n\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{1\}$ | $\mathcal{O}\{n\}$ |
| Binary Tree | $\mathcal{O}\{h\}$ | $\mathcal{O}\{h\}$ | $\mathcal{O}\{h\}$ | $\mathcal{O}\{h\}$ |
| AVL Tree | $\mathcal{O}\{\log(n)\}$ | $\mathcal{O}\{\log(n)\}$ | $\mathcal{O}\{\log(n)\}$ | $\mathcal{O}\{\log(n)\}$ |
| `std::vector` | $\mathcal{O}\{1\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{1\}^{A}$ |
| `std::deque` | $\mathcal{O}\{1\}$ | $\mathcal{O}\{n\}$ | $\mathcal{O}\{1\}$ | $\mathcal{O}\{1\}$ |

Compared are the random access methods `setAt()`, `getAt()`, `insertAt()` and `deleteAt()`, which can be used to modify specific particles inside the particle stack. This kind of methods are the most important for the algorithm. Additionally methods that modify the first and the last element of the containers are also compared, namely `insertFirst()`, `insertLast()`, `deleteFirst()` and `deleteLast()`. From these eight methods four are of especially high interest for us. The methods `setAt()` and `getAt()` are used every time step for the evolution and scattering modules, `deleteAt()` is used for the deletion of particles inside the annihilation module and `insertLast()` is used when new particles are created in a generation event and they need to be attached to the particle stack. The specific functions are colorized in blue in Table 2.1 and therefore only the efficiency of these four methods will be discussed in the following.

### 2.1.4 Interpretation and Discussion.

The first data structure, namely the dynamic array, shows constant time complexity for both set and get functions and also for the insertion method. Only the deletion method is linear in complexity. The linked list has linear complexity in all for us interesting methods, which leads to inferior efficiency in contrast to the dynamic array. The binary tree, which can be interpreted as a more complex linked list, shows time complexity for all methods that is dependend on the height $h$ of the tree. This can be in detrimental situations a value close to the number of elements $n$, which leads to linear complexity. Even tree structures that try to optimize the complexity by transforming the tree regularly to preserve a height of $h = \mathcal{O}\{\log(n)\}$ – such as the AVL tree [33] – lead to logarithmic time complexities for all methods [34]. Although the deletion method is now faster than for the dynamic array (logarithmic versus linear complexity), one has to keep in mind that the deletion method is called a lot less then the other functions. The `setAt()` and `getAt()` functions are called in every time step because of the evolution module and the `insertLast()` method is used every time a generation event happens, which is highly probable to happen also every time step. The annihilation events on the other hand are only happening if the number of particles is exceeding a specific number, which does not has to happen every time step. Therefore, even though the deletion method is more efficient in the case of the AVL tree, the dynamic array has overall the highest efficiency of the methods that are called the most frequent. This leads to the conclusion that the dynamic array – although a rather simple data structure – still represents the best choice for the particle container.

**2.1.5 Concrete Implementations.** The last two rows of Table 2.1 show two sequential containers of the C$^{++}$ standard library. The first container is a concrete implementation of the dynamic array, called `std::vector`, and will be used in the benchmark application. It possesses the same time complexities as the theoretical dynamic array structure [35]. The second container, which is also part of the options of the benchmark framework, is the `std::deque`, which can be described as follows: „*As opposed to* `std::vector`*, the elements of a deque are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one*" [36]. The information in Table 2.1 shows that the time complexities for the important methods are equal to the `std::vector`. Only the insertion and deletion of the first element is faster, but since this is not one of the four frequently used functions, this difference is not relevant for our case.

## 2.2 Related Open Source Material

After the selection of the best suited containers for the particle stack we now have to investigate the question of how to arrange the containers in the most efficient way. First we should determine the number and type of particle properties that are necessary for a Wigner signed-particle simulation. For this task we take inspiration from the ViennaWD reference implementation and additionally reconstruct the memory access patterns of the particle properties inside the application for further insight. With this information in mind we analyse a number of other open source simulators to see what concepts they use to organize all particle properties in different container combinations.

**2.2.1 ViennaWD Particle Attributes.** The ViennaWD reference assignes – if every entry in the higher dimensional attributes is counted independently – in total 15 properties to a single numerical particle. Some of them represent physical features, others are numerical values necessary for the operations of the algorithm. All attributes that each particle in the ensemble carries are now described, together with the variable name and data type that were used to represent the specific property in memory. The list of 15 attributes are organized in two different property groups to emphasise the utilities of and the dependencies between the properties.

- **Phase Space Attributes.** The particles are modeled inside the phase-space $\{\mathbf{r}, \mathbf{k}\}$, therefore the basic attributes are the two-dimensional position vector of positive real numbers (`float position[2]`) and the up to three-dimensional momentum vector of integer multipliers of the discretized momentum values (`short int momentum[3]`). The algorithm groups all particles into phase space cells, which can be accomplished with a discretized position variable that represents the space cell closest to the particle (`short int i_x, j_y`). An ensemble sorting algorithm for memory demand reduction utilizes an additional index variable that is calculated from the discretized position and momentum values (`int i5d`).

- **Evolution Mechanism Attributes.** The position calculation inside the evolution module (see Equation (1.2.2)) requires the free-flight time $\tau$ until the next scattering event happens (`float flightTime`) and the remaining time $\delta t$ (`float detT`) in the current time step, both positive real numbers. The energy level necessary for the scattering table look-up is stored in a real number (`float energy`) and a valley parameter (`short int valley`) assigns the particle to one of the three available $L$, $X$ or $\Gamma$ valleys of the band structure. Additionally, the number associated to the WIGNER potential (`short int wp_num`) assigned to the particle is used for self-force cancellation mechanisms. Each particle is marked by an activation flag (`char active`) to indicate whether the particle is still active in the stack or not. If two new particles are generated, they possess an integer sign (`short int sign`) with the values $a = \pm 1$.

### 2.2.2 ViennaWD Particle Stack.

The particle attributes described above are implemented as member variables of a classic C-style `struct`, which can be seen in the following code fragment.

```c
typedef struct {
    char active;
    short int sign;
    short int wp_num;
    short int i_x, j_y, valley;
    short int momentum[3];
    float position[2];
    float flightTime, delT;
    float energy;
    int i5d;
} particle_t;
```

Each `struct` represents one individual numerical particle. Multiple numerical particles can be combined to model a wave packet. Each wave packet is also implemented as a `struct`, where one of the member variables is a pointer to an `particle_t` array.

```c
typedef struct {
    particle_t *stack;
    /* ... */
} wavepacket_t;
```

The concept of organizing all available attributes first in a seperate structure, that is then placed continuously in memory in the form of an array is called *Array Of Structs*, often abbreviated as AOS (see Section 2.3). This is the first possibility to arrange the containers and constitutes one of the available data structures in the benchmark framework.

### 2.2.3 Open Source Monte Carlo Particle Simulators.

To find further possible data structure concepts for the particle stack, a closer look into the source code of available particle transport simulators with a similar solution approach to the WIGNER signed-particle method is taken. In particular Monte Carlo particle simulators are of high interest, as these compare the most to the fundamental solution approach of the here considered particle WIGNER solution approach. Although there is a large number of closed source simulators available, such as BOSS [37], FLUKA [38], MCNP [39], PHITS [40], Serpent [41] and TRIPOLI [42], they are not useful for our endeavor because we cannot analyze the concrete implementation of the particle stack. Table 2.2 presents a collection of open source Monte Carlo particle simulators. References to the fundamental papers, the website and the source code (often available through an online repository) are given. The field of application, the programming language and the parallelization techniques are also specified for further information. Finally the data structure concept of the implemented particle stack is denoted in the last column to compare and analyze further possible memory layouts of the particle containers.

**Table 2.2** Overview over open source Monte Carlo particle simulators.

| Name | Paper | Website | Code | Field of Application | Language | Parallelization | Particle Stack |
|---|---|---|---|---|---|---|---|
| BRICK-CFCMC | [43] | [44] | [45] | Phase and Reaction Equilibria | Fortran | None | Struct Of Arrays (SOA) |
| Cassandra | [46] | [47] | [48] | Atomistic Thermodynamic Properties | Fortran | OpenMP | Array Of Structs (AOS) |
| CP2K | [49] | [50] | [51] | Electronic Structure Calculations | Fortran | MPI, OpenMP, CUDA | Array Of Structs (AOS) |
| DLMONTE | [52] | [53] | [54] | Force Field Calculations | Fortran | MPI | Array Of Structs (AOS) |
| EGSnrc | [55] | [56] | [57] | Particle Transport Processes | Fortran | None | Array Of Structs (AOS) |
| ESPResSo | [58] | [59] | [60] | Soft Matter Research | C++ | MPI, CUDA | Array Of Structs (AOS) |
| Etomica | [61] | [62] | [63] | Molecular Simulation | Java | None | Array Of Structs (AOS) |
| FEASST | [64] | [65] | [66] | Particle-Based Molecular Simulations | C++ | OpenMP | Array Of Structs (AOS) |
| GATE | [67] | [68] | [69] | Emission Tomography | C++ | None | Array Of Structs (AOS) |
| GEANT4 | [70] | [71] | [72] | Particle Transport Processes | C++ | MPI | Array Of Structs (AOS) |
| GGEMS | [73] | [74] | [75] | Emission Tomography | C++ | CUDA | Struct Of Arrays (SOA) |
| GOMC | [76] | [77] | [78] | Phase Equilibria of Fluids | C++ | OpenMP, CUDA | Array Of Structs (AOS) |
| HOOMD-blue | [79] | [80] | [81] | Hard Particle Simulations | C++ | MPI, CUDA | Struct Of Arrays (SOA) |
| LAMMPS | [82] | [83] | [84] | Particle-Based Materials Modeling | C++ | MPI, OpenMP, CUDA | Array Of Structs (AOS) |
| MolFlow+ | [85] | [86] | [87] | Ultra-High Vacuum Radiation | C++ | None | Array Of Structs (AOS) |
| ms2 | [88] | [89] | [90] | Atomistic Thermodynamic Properties | Fortran | MPI, OpenMP | Array Of Structs (AOS) |
| mVMC | [91] | [92] | [93] | Interacting Fermion Systems | C++ | MPI, OpenMP | Struct Of Arrays (SOA) |
| OpenMC | [94] | [95] | [96] | Particle Transport Processes | C++ | MPI, CUDA | AOS (CPU) / SOA (GPU) |
| OpenMM | [97] | [98] | [99] | Molecular Dynamics Simulation | C++ | CUDA | Struct Of Arrays (SOA) |
| QMCPACK | [100] | [101] | [102] | Electronic Structure Calculations | C++ | MPI, OpenMP, CUDA | Struct Of Arrays (SOA) |
| RASPA | [103] | [104] | [105] | Adsorption and Diffusion in Materials | C++ | None | Array Of Structs (AOS) |
| SCONE | [106] | [107] | [108] | Neutron Transport Calculations | Fortran | None | Array Of Structs (AOS) |
| SPPARKS | [109] | [110] | [111] | Materials Science Applications | C++ | MPI | Struct Of Arrays (SOA) |
| Tinker | [112] | [113] | [114] | Molecular Mechanics Simulations | Fortran | MPI, OpenMP, CUDA | Struct Of Arrays (SOA) |
| WARP | [115] | N/A | [116] | Neutron Transport Calculations | C++ | CUDA | Struct Of Arrays (SOA) |

The source code of every simulator was examined and analyzed to yield the desired information. Table 2.2 contains a diverse collection of applications, languages and parallelization techniques, nevertheless we recognize a specific pattern of only two different types for the utilization of data structure concepts for the particle stack. The majority of the presented simulators (16 projects colored in blue) use the same arrangement of the particle attribute containers, namely the Array Of Structs (AOS) concept. This means that they, similar to the ViennaWD reference implementation, define a seperate structure for every particle. These structures are then concatenated to form an array in the memory. The other concept, which is utilized by fewer simulators (10 projects colored in green), is the so-called Struct Of Arrays (SOA) concept that reverses the idea of AOS (see Section 2.3). In this case, not all particle attributes are combined into one single struct, but a seperate array for every attribute is allocated in memory. This means that for example an array is allocated for the first component of the position vector, the same is done for the second component, and so on. In the case of the ViennaWD reference implementation, where in total 15 attributes are implemented, also 15 arrays would have to be allocated. These 15 arrays (or pointers to the first element of them) would then be used as member variables of a single `struct` that contains all particles of the ensemble. The SOA is the second option for a data structure in the benchmark application. AOS and SOA seem to be the standard for Monte Carlo particle simulator implementations and will, therefore, be implemented, benchmarked and analyzed in detail in Chapters 3 and 4.

## 2.3 Derived Data Structure Concepts

The analysis of data structure design in this chapter has led us to the conclusion that dynamic arrays are the best suited containers to store particle attributes for algorithms where these properties are modified in every time step. Additionally we know from the examination of source code of the ViennaWD implementation and other open source Monte Carlo particle simulators, that both the Array Of Structs (AOS) and the Struct Of Arrays (SOA) concepts are, although being old concepts, still the go-to approaches for particle stack designs. This section summarizes the ideas behind these data structures to give a clear reference for the following chapters, where the implementation of the benchmark application is discussed in detail. Additionally a third concept is introduced, that bases on the idea that all attributes possess the same data type and can therefore be stored inside one single array.

**2.3.1 Array Of Structs (AOS).** This data structure is the classical implementation of a particle stack. It follows the mindset of humans in general to store data that belongs conceptually together also locally close together in the memory. Therefore the attributes of a single particle are combined to a `struct` that arranges the attributes one after another. The following code snippet gives a simplified example for an implementation in C++.

```cpp
struct Particle
{
    char  active;
    int   momentum;
    float position;
};

struct ArrayOfStructs
{
    std::vector<Particle> particleArray(100);
};
```

In this section we assume a pseudo particle that only contains three attributes with different data types, an activation flag, a discrete momentum and a continuous position variable. If we want to store 100 particles in our pseudo particle stack, we would generate a dynamic array (in this example `std::vector`) with 100 `Particle` elements, which themselfs are structs. This data structure is therefore an Array Of Structs (AOS).

**2.3.2 Struct Of Arrays (SOA).** If on the other hand we want to generate a single `struct` that contains not only one particle, but all particles of the ensemble, the following concept is implemented.

```
struct StructOfArrays
{
    std::vector<char>  activeArray(100);
    std::vector<int>   momentumArray(100);
    std::vector<float> positionArray(100);
};
```

Since our pseudo particle only possesses three attributes, only three distinct dynamic property arrays have to be created. We want to store 100 particles, therefore every dynamic array has also the length 100 to have enough memory to store all attributes of all particles.

**2.3.3 Array Of Floats (AOF).** Additionally to the already defined data structures we introduce a third possibility, where we assume that all particle attributes are implemented by the same data type, in this case `float`. Then the opportunity arises for an alternative memory layout, where only one single dynamic array is sufficient to store all particle properties.

```
struct ArrayOfFloats
{
    std::vector<float> attributeArray(300);
};
```

One has to keep in mind to allocate the space not only for 100 elements, but for 300 elements, since every single one of the 100 particles contains three distinct attributes. This data structure concept is also available in the benchmark application and part of the runtime measurements. It should be mentioned that certain issues can arise when `char` and `int` variables are stored as `float`. The conversion between the different data types could lead to imprecise values, especially when the `char` flag is used as a simplified boolean variable that differs between zero and other values. Also particle attributes that should only be realized as integer values could now potentially contain improper values, which are (due to rounding errors) close, but not exactly equal, to the desired `int` values.

**2.3.4 Container Variation.** The examples of the pseudo particle stacks above only contain `std::vector` objects as containers, but in Section 2.1 we mentioned that not only dynamic arrays should be available as containers for particle attributes, but also other classes that feature an interface with an overwritten index operator, for example the `std::deque` class from the C$^{++}$ standard library. To ensure the flexibility of the particle stacks to utilize a variety of different containers, template parameters and other C$^{++}$ features are applied in the data structure class definitions. For more information on the implementation of the benchmark application see Chapter 3.

# 3 Benchmark Framework

After the establishment of all necessary facts and information in the chapters above it is now time to present the ParticleStackBenchmark application. This benchmark framework is based on fundamental ideas of software engineering and tries to give the user as much freedom as possible when the software is employed for different tasks. This is accomplished by a multitude of input parameters and other options that are described in this chapter. Also the concepts of flexibility and expandability lead to features that allow the user to develop custom modules that increase the functionality of the framework and which help to improve the quality of the benchmark execution sequences in general. Therefore, although this application has been developed with a concrete context in mind (particle Wigner simulations), the fundamental framework and its design can be applied to other contexts as well. The source code of the framework is available on TUgitLab [117].

## 3.1 Software Specification

This section gives a short specification of the ParticleStackBenchmark application that was developed and used to yield the results of this thesis. After the definition of the main objective and the description of the most important requirements, a few aspects regarding the design of the software are mentioned. The section closes with a list of the utilized implementation methods and tools for all modules of the application, including the pre-processing of the input parameters, the main execution of the framework and the post-processing of the output data. Figure 3.1 shows all parameters and components of the framework, together with references to the corresponding sections and algorithms.

**3.1.1 General Description and Objective.** The objective of the ParticleStackBenchmark application is to provide the means to determine the best suited particle stack implementations in terms of performance for a specific context, here the Wigner signed-particle method. The application measures the runtime of different particle stack implementations regarding the execution of specific test functions which are motivated by the context in mind. The software user can choose between different input parameters to adjust the application output to the particular research needs. The raw output files of the application are not adequate for the determination of the performance hierarchy. Therefore, the extraction of the runtime values from the output files is automated by additional visualization tools, which support the analysis and the comparison of the results.

**3.1.2 Practically Inspired Test Functions and Data Structures.** The executed test functions should perform operations that either can be used to deduce reference values for performance ranges and are easy to check for correctness, or that are similar to the operations that are performed in the algorithm of interest. Test functions of the first kind are reference functions for correctness and should give a feeling for the performance measure in general. The second kind should try to emulate the frequency and specific patterns of memory access of the actual algorithm to yield precise insights. The application incorporates both kinds of test functions, which are discussed below (see

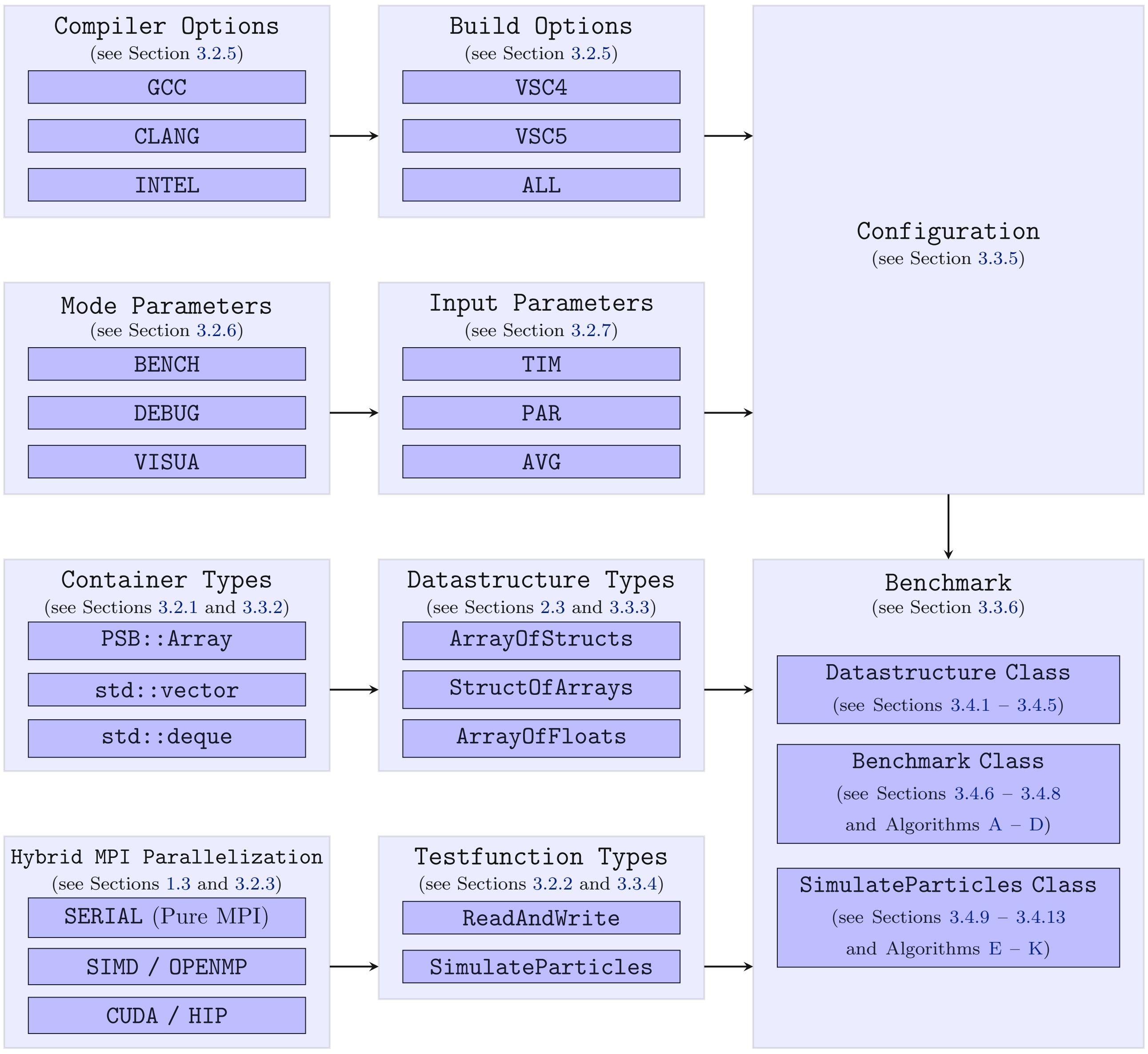

**FIGURE 3.1** Components of the ParticleStackBenchmark application.

Section 3.2.2). Also the data structures for the particle stack should be inspired by the investigated algorithm and adjusted to the utilized hardware. The exploration of different data structures above (see Section 2.1) suggests particle stacks that have the ability to perform random access operations on all simulated particles and their member variables. Therefore, only data structures with this property are implemented and tested.

**3.1.3 Requirements for Cluster Execution.** The application should be able to be executed directly on high-performance computing hardware, specifically on supercomputer clusters. This fact leads to the following requirements regarding the user interface and the software design.

- **Command Line Input.** Almost all modern supercomputers utilize a member of the family of open-source operating systems based on the Linux kernel. All these so-called distributions provide a terminal with a command line interface (CLI) which is used for user interaction, in contrast to a graphical user interface (GUI) which is commonly found on personal workstations. Therefore, all interactions between the user and the cluster need to be performed via command line instructions.

- **Text File Output.** The output of the application is written into human-readable text files, following a comma-separated-values (CSV) format. This allows for command-line level inspection of the output (already on remote clusters) and straightforward, external post-processing due to the structured format.

- **Specific Parallelization Techniques.** The particular hardware of the cluster determines the possibility of the utilization of different parallelization techniques. Therefore, the application incorporates all techniques that are available, but also ignores methods that are not supported by the cluster.

Additionally, it should be noted that all components of the application have to be compatible with the software modules that are available on the cluster. This also includes the used build tools, compilers and libraries. Therefore, it is advantageous to design the application around and on top of established open source standards and up-to-date library versions.

**3.1.4 Software Engineering Aspects.** A software specification does not only include the general objective of the application. It also has to define the software engineering principles that are applied to design a software structure that is not only useful for a few specific cases, but for a wide range of different tasks. The remainder of this section discusses the characteristics and perspectives on which the development of the application was based on.

**3.1.5 Correctness Investigation Tools.** Even though the output of interest are the runtime values, it should be ensured that the results of the measured test functions are also correct, otherwise the execution times lose their meaning and are worthless for further processing. Therefore, different tools for the testing of the test function results are implemented. Deterministic test functions can be checked by a comparison of the output values with correct pre-calculated values, and if the magnitudes of the variables do not coincide, the application terminates. However, this method is not feasible for test functions with a stochastic component. One possibility for these test functions is the visualization of the calculations, which can be verified by visual inspection of the user to conclude plausible results. Both methods are implemented and are discussed below (see Sections 3.2.2 and 3.2.9).

**3.1.6 Framework Characteristics.** The ParticleStackBenchmark application is designed with flexibility and expandability in mind, which are desirable for software frameworks in general. These characteristics ensure that scientific software is of greater usefulness for both the specific investigation at hand and for the whole research community at large. The organization of the application provides usability and reusability by utilizing a component-based structure, where the individual modules can be combined to custom benchmark execution files (for an example see Section 3.3.9). Abstract interface guidelines for data structures (see Section 3.4.3) and test functions allow the design and implementation of custom particle stacks and even complete new test functions (see Section 3.4.14), which increases the flexibility and expandability of the application.

**3.1.7 Software User Perspective.** Not every software user has the same education, experience or goals while utilizing software frameworks. The software engineer has to take this fact into account when the design of the application should incorporate the framework characteristics described above. Weinbub [31] identifies three different software user groups, the *end user*, who has no interest in the technical details of the tool and views it as a black box with specific input and output parameters, the *advanced user*, who is inclined to modify the utilized software for specific tasks by changing mostly easily accessible software components, and the *developer*, who generates scientific software and views applications from both the low-level and the high-level design perspectives. These three groups are sorted in increasing order of assigned software engineering capabilities and skills. The different perspectives of these software users constitute the content of the following sections in this chapter, which describes the benchmark application in three different levels of precision (Section 3.2 for the end user, Section 3.3 for the advanced user and Section 3.4 for the developer perspective).

**3.1.8 Implementation Methods and Tools.** The implementation of the ParticleStackBenchmark application is based on different open source software tools and programming languages that should be mentioned in this section. Without the help of these sophisticated tools it would have been impossible to develop a benchmark application with such a wide variety of options for the user.

**3.1.9 Implementation of the Benchmark Module.** The application is written in the C++ programming language [118], which offers a number of advantages in the context of the concrete implementation. The framework is implemented as a so-called header-only library, which means that – if all necessary header files are available – the central header file can also be included into other programs to add further functionality to them. All classes and functions are available under the namespace `PSB` (ParticleStackBenchmark). The following characteristics of C++ support different implemented features of the application, which are discussed in detail in the remainder of this chapter.

- **Preprocessor.** Even though the C++ preprocessor is nowadays mostly seen as a relict of the past, it still has a specific functionality that is very useful for the implementation of the benchmark application. The definition of preprocessor variables in combination of the conditionally compilation of parts of the source code is used to implement different parallelization techniques in the same files, classes and methods. If libraries are for some reason not available, a preprocessor variable can be undefined and the dependent part of the source code is ignored (an example can be seen in Section 3.4.7).

- **Standard and Parallel Libraries.** The application makes heavy use of the C++ standard library up to the C++17 standard. Many of the functionalities, often based on the programming paradigms discussed below, are incorporated in these library classes and methods. C++ also supports all well-established parallelization libraries such as implementations of MPI, OpenMP, CUDA and HIP.

- **Object-oriented Programming.** This programming paradigm allows the organization of source code into seperable modules, which are called classes, where data (class members) and functionality (class methods) can be grouped together. Different classes can also rely on one another by simple or complex dependencies. Object-oriented features such as inheritance and polymorphism are utilized to support the expandability of the application (see for example Section 3.4.4).

- **GENERIC PROGRAMMING.** Almost all classes and methods of the application use the power of template programming to shorten and structure the source code. It also provides more flexibility for custom benchmark execution files by establishing a concise interface of the specific benchmark methods (see Section 3.3).

- **FUNCTIONAL PROGRAMMING.** Specific function objects are used to deduce the data types of the class members of the particle stack inside the benchmark methods (see for example Section 3.4.5). This allows further compactness of the source code and allows to design custom data structures without much overhead.

**3.1.10 IMPLEMENTATION OF THE ADDITIONAL MODULES.** Before and after the execution of the main benchmark module there are pre- and post-processing steps necessary to complete the application workflow and to yield meaningful output results that match with the given input parameters. The most important auxiliary modules are listed in the following.

- **BUILD PROCESS WITH CMake.** The utilization of CMake [119] should specifically be mentioned in the context of this application, since the well-arranged implementation of the different build options (see Section 3.2.5) are only possible by the defined and undefined CMake variables which are transferred to the C$^{++}$ preprocessor. This is also the case for all used parallelization libraries.

- **CLUSTER EXECUTION WITH Slurm.** The job scheduler Slurm [120] allows with a few simple, but powerful commands the generation of job submit scripts which help to execute the application on supercomputer clusters with the correct options and parameters. The features of Slurm were essential for the creation of the results of this thesis, especially for the management of the process and thread configuration (see Section 4.1.2).

- **OUTPUT VISUALIZATION WITH Python.** The visualisation of the output text files is accomplished with the help of Python [121] scripts that take the text files as input and automatically generate bar chart diagrams for runtime files (see Figure 3.3) and animations of the particle simulation for visualization files (see Figure 3.2). The diagrams allow a comparison of the yielded runtimes and the animations serve as a plausibility check for the dynamic test functions with a stochastic component.

## 3.2 THE END USER PERSPECTIVE

The end user is only interested in the execution of an application to yield information that can be used to advance the investigation at hand. The specific technical methods and tools inside the application are not important. Therefore, this section gives an overview of all input options and output file types that are available for the ParticleStackBenchmark application. The data structures with their containers and the test functions with their parallelization methods, which can be combined in every possible way to yield benchmark results for a specific test case, are presented. After the description of the features, the user is guided through the execution steps of the application, where simultaneously all relevant options are discussed and illustrated with simple examples.

**3.2.1 Container Implementations.** The three already discussed data structures AOS, SOA and AOF (see Section 2.3) are available in the application. The above mentioned containers inside the data structures can be specified via a template parameter (see Section 3.3.3). In theory every container that supports the interface of the dynamic sequence containers of the C++ standard library [122] with an overwritten index operator can be used in this fashion. There are currently the following dynamic container types supported (see also Section 2.1.5):

- **`PSB::Array`.** This container uses in its core a classic C-style array, which allocates memory on the heap. The dynamic features are added by further reallocation member functions. Therefore, this container can be seen as a wrapper class. The namespace `PSB` indicates functionality of the ParticleStackBenchmark application.
- **`std::vector`.** The dynamic array implementation of the C++ standard library.
- **`std::deque`.** The doubly ended queue implementation of the C++ standard library.

**3.2.2 Test Function Implementations.** The above data structures use the test functions for the benchmarking process inside a time loop, which itself is executed multiple times to calculate an average value of the runtime (see Section 3.4.7). These test functions are available:

- **`ReadAndWrite`.** This test function reads the properties of each particle from the data structure, increments it by one and writes it back into the same space in memory (see also Section 3.4.11). The number of different properties that should be modified can be selected by a seperate template parameter. The correctness of this deterministic test function is checked by specific `assert()` function calls that compare the result values with pre-calculated values.
- **`SimulateParticles`.** Here the particles are moving inside a simulation area according to classic Newtonian mechanics which results in a drift trajectory. Additional simple scattering, generation and annihilation events try to test the performance of the dynamic data structures in use (see also Section 3.4.12). The output files can be processed into an animation of the particle evolution which can be used for visual inspection to ensure the plausibility of the calculations.

The test functions can be executed in serial or they can use different parallelization techniques. The test functions can be run on architectures which use a distributed memory system (see Section 1.3.2). This means the different processes do not share the memory on which the operations are calculated. Therefore, they have to use a framework that helps communicate the data between them. In this application the MPI library is used to fulfill these requirements.

**3.2.3 Parallelization.** On the basic parallelization level the test functions are executed on multiple MPI processes in serial. But inside each MPI process another parallelization method can be used to further improve the performance. These parallelization methods are sometimes also called *implementations* in this thesis. The following implementations for all testfunctions are currenty available (see also Section 1.3):

- **SERIAL.** This implementation does not use any parallelization technique, its just uses one single thread on one core of the CPU per employed compute node of the cluster. This case represents the reference value for all other implementations regarding the speedup and performance.
- **SIMD.** Here the calculations are augmented by vectorization operations if they are supported by the used CPU architecture. These allow to use specific instructions on multiple data objects at once.

- **OPENMP.** In this implementation multiple threads are used by the OpenMP library in parallel to perform the required tasks. The calculations are parallelized, but are still running only on the CPU.
- **CUDA.** Now the operations are not only running on the CPU, but also the performance of the GPU can be taken into account. The CUDA programming toolkit is an extension of C++ and is supported by most GPU architectures. The calculations itself can experience an excellent speedup if executed on the GPU, but the data migration between the CPU and the GPU can cause performance drops.
- **HIP.** This implementation also runs on the GPU (similar to the CUDA case), but here the HIP programming toolkit is used, which is distributed by AMD (where CUDA is maintained by NVIDIA).

**3.2.4 APPLICATION WORKFLOW.** Additionally to all possible benchmark combinations discussed above there are a variety of options that can be chosen by the user to specialize the execution of the application even further. All available build options, mode and input parameters are listed below, followed by a short description of the different output files and the post processing step.

**3.2.5 BUILD OPTIONS.** This software project uses CMake as a building tool to support the user. The `CMakeLists.txt` file defines different variables which are connected to preprocessor constants that are given to the compiler as input parameters. These constants define which parts of the source code should be compiled. There are three different C++ compilers that can be selected for the building process:

- **GCC.** The GCC compiler [123] can be selected by defining the CMake flag `GCC`.
- **CLANG.** The CLANG compiler [124] can be selected by defining the CMake flag `CLANG`.
- **INTEL.** The INTEL compiler [125] can be selected by defining the CMake flag `INTEL`.

Furthermore, there are CMake flags for different implementations. The reason behind these is the fact that not every computer owns the required architecture or utilizes all the necessary libraries which are needed for all available implementations. The SERIAL implementation is always compiled, all other methods can be selected with the flags `USE_SIMD`, `USE_OPENMP`, `USE_CUDA` and `USE_HIP`. While it is possible to combine these four flags in every way possible, it is easier to use these predefined flags in the compilation process:

- **VSC4.** Parallelization only on the CPU, which means the SERIAL case and additionally the `USE_SIMD` and `USE_OPENMP` flags, can be selected by defining the CMake flag `VSC4`.
- **VSC5.** All three implementations from `VSC4` and additional parallelization on the GPU with `USE_CUDA` can be selected by defining the CMake flag `VSC5`.
- **ALL.** All five implementations, this means the `VSC5` flag and additionally `USE_HIP`, can be selected by defining the CMake flag `ALL`.

The names of the `VSC4` and `VSC5` flags are based on the VSC-4 [23] and VSC-5 [24] clusters (see also Sections 1.3.1 and 4.1.1). If the user wants to compile the benchmark application with, for example, the GCC compiler only with implementations working on the CPU, the following command is used

```
cmake -DGCC=ON -DVSC4=ON .. && make
```

As another example, if the user wants to compile the benchmark application with the INTEL compiler and with all implementations available, the following command is used

```
cmake -DINTEL=ON -DALL=ON .. && make
```

### 3.2.6 Mode Parameters.

There are different modes available for the `ParticleStackBenchmark` executable. These are given to the file by command line parameters. Currently there are three modes implemented:

- **BENCH.** This is the benchmark mode. It measures the runtime of the execution of the test functions with the corresponding data structures. Currently all nine data structure combinations execute the three test functions `readAndWrite(1)`, `readAndWrite(15)` and `simulateParticles()` with 1000 timesteps, 1000 particles and 3 execution runs for the average value. This mode can be selected by the parameter `BENCH`.
- **DEBUG.** This is the debugging mode. It checks if the resulting values from the calculations are correct. If this is not the case for at least one value, the programm is exited by an `assert()` function call. Here all nine data structure combinations execute the test functions `readAndWrite(1)` and `readAndWrite(15)` with 10 timesteps, 10 particles and 1 execution run for the average value. This mode can be selected by the parameter `DEBUG`.
- **VISUA.** This is the visualization mode. Since the `simulateParticles()` test function contains random generated values, it is not possible to check the results with an `assert()` function call as in the debugging mode. Therefore, an output file is generated which can be processed into an animation of the simulation for visual expection and sanity checking. Here only one data structure combination is executing the test function `simulateParticles()` with 500 timesteps, 100 particles and 1 execution run for the average value. This mode can be selected by the parameter `VISUA`.

This means for example if the user wants to execute the benchmark application with only 1 MPI process in BENCH mode, the following command is used

```
mpirun -np 1 ParticleStackBenchmark -BENCH
```

As another example, if the user wants to execute the benchmark application with 8 MPI processes in VISUA mode, the following command is used

```
mpirun -np 8 ParticleStackBenchmark -VISUA
```

There are further input parameters available, which are discussed next.

### 3.2.7 Input Parameters.

As can be seen above, every mode is executed with default values. If the user wants to change these values, the following parameters from the command line can be used:

- **TIM.** This parameter defines the number of time steps that are used inside the time loop that is executing the selected test function. This value can be changed by the command `-TIM=X`, where `X` is an integer between 1 and 9.
- **PAR.** This parameter defines the number of particles that are used as starting value for the execution. In the case of the `readAndWrite()` test function this number is fixed, but for the

`simulateParticles()` test function it can potentially be modified every time step. This value can be changed by the command `-PAR=X`.

➤ **AVG.** This parameter defines the number of execution runs of the time loop to calculate the average runtime. After all execution runs, the runtime is divided by this number to yield the average. This value can be changed by the command `-AVG=X`.

Integer values in the interval between 1 and 9 are valid for all three input parameters. One has to be careful in the case of the `TIM` and `PAR` parameters (but not the `AVG` parameter), where this value is the exponent in scientific notation. This means if one uses the commands

```
-TIM=3 -PAR=4 -AVG=2
```

the number of time steps equals $10^3 = 1000$, the number of particles equals $10^4 = 10\,000$ and the number of average runs equals 2. All three input parameters can be used in any order and any given number of them can also be omitted. If the parameter is omitted, the default value of the chosen mode (see Section 3.2.6) is used instead. For example if the user wants to execute the benchmark application with only 1 MPI process in BENCH mode, but with $10^5 = 100\,000$ time steps instead of the default $10^3 = 1000$ the following command is used

```
mpirun -np 1 ParticleStackBenchmark -BENCH -TIM=5
```

The number of particles and the number of average runs are then still the default values $10^3 = 1000$ and 3. As another example, if the user wants to execute the benchmark application with 8 MPI processes in VISUA mode, but with $10^4 = 10\,000$ time steps, $10^4 = 10\,000$ particles and 8 average runs, the following command is used

```
mpirun -np 8 ParticleStackBenchmark -VISUA -TIM=4 -PAR=4 -AVG=8
```

As described above, the input parameters are optional, but at least one of the three mode parameters `BENCH`, `DEBUG` or `VISUA` has to be specified. If a command line call is not valid, an error and usage message is displayed on the command line. All used input parameters are also written into the output files for later inspection. These output files are discussed next.

**3.2.8 OUTPUT FILES.** If the `ParticleStackBenchmark` application is executed, the progress is printed to the command line for visual inspection. This is helpful if the user wants to estimate how far the benchmarking process is away from finishing. An example progress output for a `VISUA` mode execution is given below:

```
user@workstation:path$ mpirun -np 1 ParticleStackBenchmark -VISUA

USING VISUA MODE.

1 MPI RANK(S) WITH 16 OPENMP THREAD(S) EACH AVAILABLE.

[ 20%] (1/5) simulateParticles() AOF std::deque SERIAL 0.078187 finished.
[ 40%] (2/5) simulateParticles() AOF std::deque SIMD   0.048343 finished.
[ 60%] (3/5) simulateParticles() AOF std::deque OPENMP 0.077729 finished.
[ 80%] (4/5) simulateParticles() AOF std::deque CUDA   0.655052 finished.
[100%] (5/5) simulateParticles() AOF std::deque HIP    0.396987 finished.
```

```

COMPLETE RUNTIME = 1.3 SECONDS
```

The progress in percent, the absolute number of executed test functions, the data structure combination, the test function implementation and the runtime is displayed on the command line. Additionally the executed mode, the available number of MPI processes and OpenMP threads and the complete runtime of the application can also be inspected. All three modes also generate an output file in `.txt` format. All modes write the input parameters into the output file, which are the mode itself, the number of MPI processes, the number of OpenMP threads, the number of time steps, the number of particles and the number of average runs. Additionally, for the visualization of the simulation, the length of the simulation area, the minimum distance for the annihilation step and the length of a single time step are also included. These are the header lines which all output files include. The `BENCH` mode output file also lists the executed test function, the data structure combination, the test function implementation and the runtime, similar to the progress output. The `DEBUG` mode output file lists the result value of all properties of all particles inside the data structures for reference and for correctness. The `VISUA` mode output file lists for every time step the complete number of particles, the position in two dimensions and the generation index for every particle. The names of the output files are automatically generated and use the input parameters as reference. The output files for the `BENCH` and `VISUA` modes can be visualized, which is discussed next.

**3.2.9 Post Processing.** It is possible to study the output files in text format. But to get an overview of the results it is practical to visualize the output data. For this task a Python script called `PLOT.PY` is ready for usage. The script automatically recognizes if a `BENCH` or a `VISUA` output file is given as input. If the output file `FILE.txt` should be visualized, the following command is used

```
python3 PLOT.PY FILE.txt
```

If a `DEBUG` output file is given, an error message appears on the command line. An example for a `VISUA` output file visualization screenshot can be seen in Figure 3.2. The animations have the format `.mp4`. The left plot shows a screenshot at the time step $t = 499$, where currently $N = 684$ particles are present in the simulation region. These values are shown in the top middle. The colors of the particles themselfs indicate the number of generation events that were necessary to construct them. The plot on the right displays the same information, but as a density plot, where the colorbar on the far right indicates the density in a specific space region. The resolution of the density plot is changeable inside the script. An example for a `BENCH` output file visualization can be seen in Figure 3.3 (detailed discussions on benchmarks are given in Chapter 4). The runtimes are sorted from the slowest on the left to the fastest execution on the right. The runtime is plotted on the $y$-axis in logarithmic scale. The $x$-axis displays the data structure combination and the implementation. The name of the test function can be seen in the top right of every plot. The colors of the bars indicate the used data structure. The numbers over the bars display the speedup of a specific implementation in relation to the SERIAL reference value. This means that the SERIAL implementations always display the value **1**. The values inside the bars are the inverse values to the ones on the top.

## 3.3 The Advanced User Perspective

The advanced user has the interest to investigate the application that is utilized and requests the possibility to modify specific components of the software to customize the input parameters and,

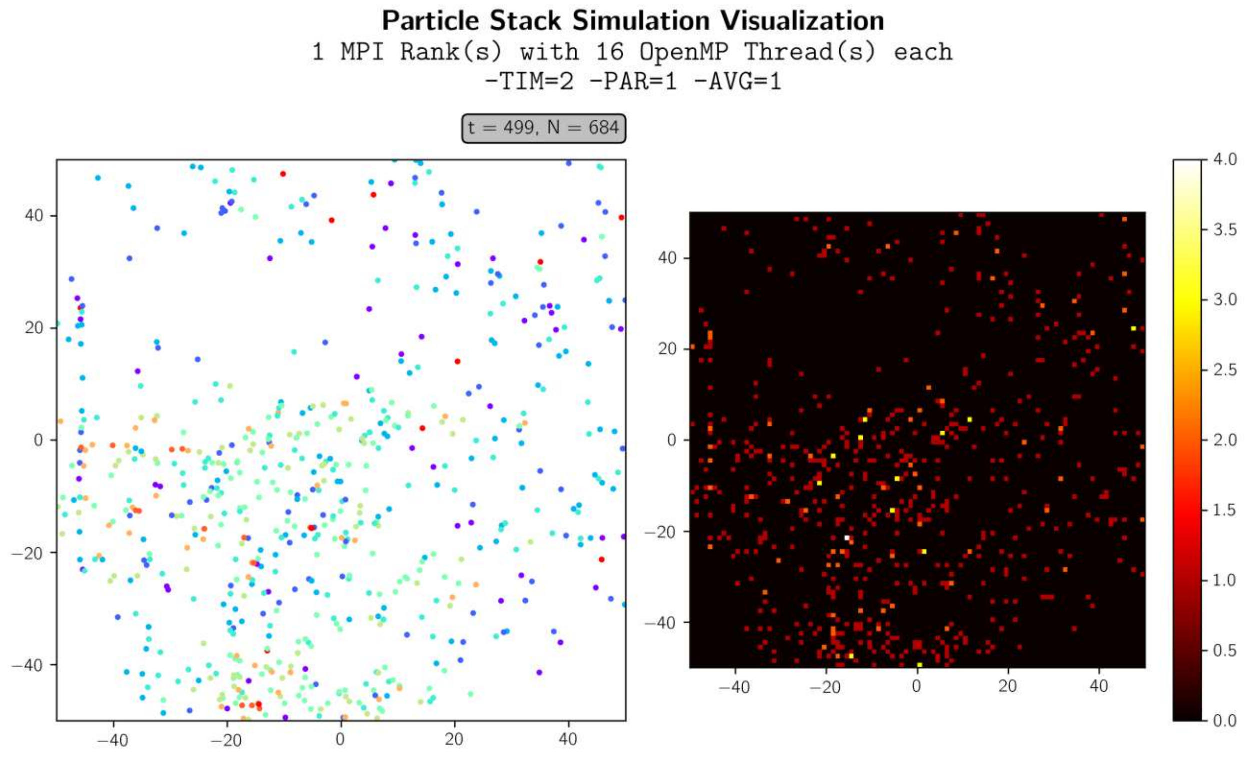

**Figure 3.2** Screenshot of the animation of the particle evolution calculated by the `simulateParticles()` test function. If the VISUA mode is selected, an output file is generated, which can be processed by the Python script `PLOT.PY` to generate the animation. A description of the screenshot is given in Section 3.2.9.

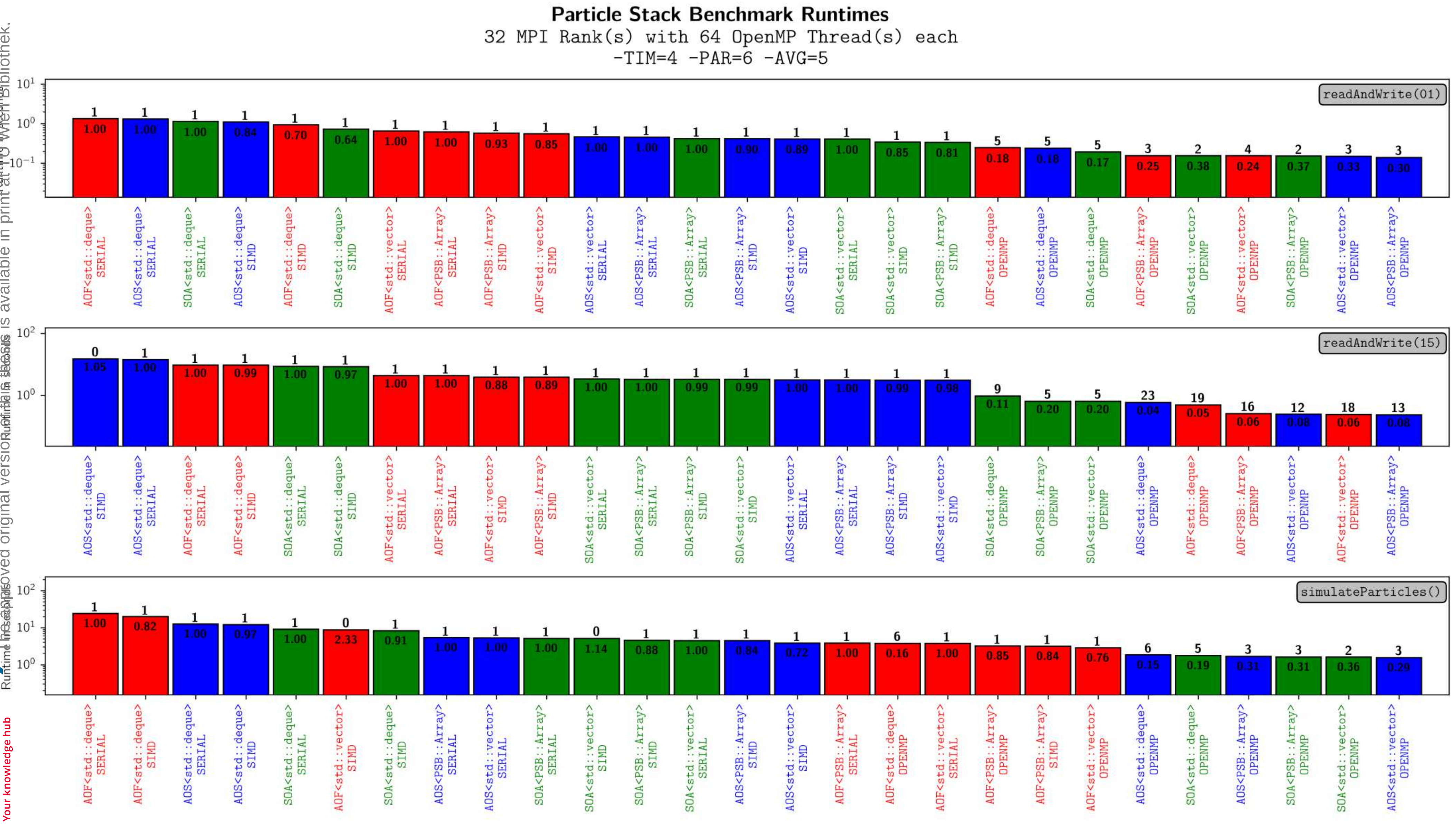

**FIGURE 3.3** Visualization of a BENCH mode output file. The configuration of the processes and threads and the input parameters can be seen at the top. A description of the content is given in Section 3.2.9.

therefore, the execution and the output data. The ParticleStackBenchmark application supports the feature of custom benchmark sequence generation with the help of the class system that is located inside the core of the implementation. This feature is established by first discussing all the necessary classes and their dependencies via heavy use of template parameters and then describing the required commands to assemble the execution file.

**3.3.1 CLASS SYSTEM DEPENDENCIES.** Behind the different input options discussed above lies a system of classes which use themselfs as parameters in an interlaced structure. This system is described in the following and serves as an overview for the possibility of custom benchmark execution file generation. In the end all classes are incorporated into the `Benchmark` class as template parameters or constructor arguments. Additionally to the class definition all class interfaces are formulated in a pseudo-notation to construct an abstract scheme for the reader. The basic notation pattern is represented by Class⟨Template Parameter⟩.

**3.3.2 CONTAINER CLASSES.** All containers offer the same class interface, where a template parameter defines the data type of the container elements. As representative example only the class definition

```
template <typename Type>
class Array
{ /* ... */ };
```

of the wrapper class for a classic C array with additional member functions for dynamic memory allocation is shown here, the definitions for `std::vector` and `std::deque` are similar and can be found in the header files of the standard library. The pseudo-notation for the container classes is $C = C\langle t\rangle$, where $C$ stands for the class itself and $t$ denotes the type parameter. The type parameter is omitted in the following discussion.

**3.3.3 DATA STRUCTURE CLASSES.** All data structure classes are derived from an abstract class called `Datastructure`, which only contains virtual member functions which should be overwritten by implementations which are derived from this class (see Section 3.4.3). Also the specific data types for the particle properties (see Section 2.2.1) should be specified as template parameters by the derived classes. This can be seen in the class definition

```
template <template <typename> typename ContainerType>
class ArrayOfStructs
    : public Datastructure<
        char, // activeType
        int, // i5dType
        short int, // momentumType
        short int, // signType
        short int, // wp_numType
        short int, // i_xType
        short int, // j_yType
        short int, // valleyType
        float, // positionType
        float, // flightTimeType
        float, // delTType
        float> // energyType
{ /* ... */ };
```

for the `ArrayOfStructs` class, the other particle stack class definitions possess the same kind of interface. Every data structure takes a container type as template parameter as input, where $C$ denotes the container class and $D$ the data structure class, which leads to the pseudo-notation $D\langle C \rangle$.

**3.3.4 Test Function Classes.** Both the static test function `ReadAndWrite()` and the dynamic test function `SimulateParticles()` possess the same class interfaces regarding the data types and template parameters. As representative example only the first test function

```
template <long numberOfVariables>
class ReadAndWrite
{ /* ... */ };
```

is shown here. The pseudo-notation $T\langle n \rangle$ represents this interface, where $T$ denotes the test function class and $n$ stands for the `numberOfVariables` template parameter. It denotes the number of variables that are read, incremented and written back into memory. The range of $n$ lies between 1 and 15, the maximal number of implemented particle properties (see Section 2.2.1). The `SimulateParticles()` test function also uses a template parameter of data type `long`, but in this case it represents the number of maximal particles before the annihilation mechanism is started. Therefore, the parameter can theoretically accept all values greater than 1. The individual parallelized versions of the test function are implemented as nested classes inside the test function class, in the case of `ReadAndWrite()` for the SERIAL implementation as

```
template <long numberOfVariables>
template <typename DataStructureType>
class ReadAndWrite<numberOfVariables>::SERIAL
{ /* ... */ };
```

where the other implementations SIMD, OPENMP, CUDA and HIP possess the same interface and are omitted here.

**3.3.5 Configuration Class.** This class manages the input parameters, the output file generation, the MPI communication and further functions which set up the benchmark environment. It organizes the MPI processes and the corresponding command line output. Furthermore, it transfers all input variables such as the number of time steps, the number of particles and the number of benchmark executions for the averaging process to the executed test function classes. The mode parameters are processed and the corresponding flags are set. The class also writes the results of the benchmarks together with progress information into the output stream for command line inspection. The class has to be instantiated before the benchmarks are executed, where the constructor initializes the MPI methods and the destructor finalizes them. The class definition

```
class Configuration
{ /* ... */ };
```

possesses no template parameters, therefore it has no dependency on any other class. It rather constitutes one of the input parameters for every benchmark instantiation. It is represented by the pseudo-notation Con.

**3.3.6 Benchmark Class.** This class manages the MPI communication between the different MPI processes, runs and benchmarks the selected test function and generates the output files which are used for further post-processing steps by other applications or scripts. A type deduction mechanism (see Section 3.4.5) allows to benchmark data structures with different data types as input. It also contains the methods for correctness checking. The class definition

```cpp
template <typename DataStructureType, typename TestFunctionType>
class Benchmark
{ /* ... */ };
```

shows that the data structure and the test function classes of interest are taken as template parameter input. The `Configuration` class object is given as a function argument. This can be represented by the pseudo-notation $\mathbf{B}\big\langle D\langle C\rangle, T\langle n\rangle\big\rangle(\text{Con})$ and serves as the main interface for custom benchmark execution, which is discussed next.

**3.3.7 Custom Benchmark Execution.** If the user is interested in specific benchmark sequences that are not part of the three default modes described above (see Section 3.2.6), then there is the possibility to generate custom benchmark execution files. The class structure of the application enables the potential to use simple commands as building blocks for individually generated benchmark sequences. After the presentation of these building blocks a short example is given to illustrate this useful feature.

**3.3.8 Execution File Building Blocks.** To generate a custom benchmark file, the header file for the ParticleStackBenchmark library has to be included into the source file, which is accomplished with

```cpp
#include "ParticleStackBenchmark.hpp"
```

Then the `Configuration` class object has to be created. The constructor

```cpp
Configuration(int argc, char** argv) { /* ... */ }
```

takes the argument counter and vector of the executable file as input, therefore the instantiation is obtained with the command

```cpp
PSB::Configuration c(argc, argv);
```

After these two simple preliminary steps the user is free to construct a custom benchmark sequence, where every combination of the above mentioned options can be incorporated. The constructor of the `Benchmark` class

```cpp
Benchmark(Configuration& configuration) : configuration{configuration} { /* ... */ }
```

uses the `Configuration` object Con as argument. Additionally the data structure class $D$ with the corresponding container template parameter $C$ and the test function class $T$ with the corresponding number of particles template parameter $n$ are given, which can for example be done by

```cpp
PSB::Benchmark<PSB::ArrayOfStructs<std::vector>, PSB::ReadAndWrite<15>> b(c);
```

where the pseudo-notation $\mathbf{B}\big\langle D\langle C\rangle, T\langle n\rangle\big\rangle(\text{Con})$ can be recognized. In this example the data structure is chosen as the `ArrayOfStructs` class with `std::vector` container, while the test function `ReadAndWrite()` is selected, where all 15 particle properties are modified.

**3.3.9 Execution File Example.** The instruction steps above are now illustrated by a short example. The following source file includes the header file, creates the necessary `Configuration` class and defines the benchmarks that should be performed. It should be noted that the constructor and the destructor of the `Configuration` class initialize and finalize the MPI library, therefore it should always be instantiated in a scope that encompasses the parallelization methods that utilize MPI functions.

```
 1  // Include for all ParticleStackBenchmark classes
 2  #include "ParticleStackBenchmark.hpp"
 3
 4  // Main function for custom benchmark execution
 5  int main(int argc, char** argv)
 6  {
 7      // Creating Configuration object
 8      PSB::Configuration c(argc, argv);
 9
10      // Benchmark sequence for all modes
11      PSB::Benchmark<PSB::ArrayOfStructs<PSB::Array>, PSB::ReadAndWrite<1>> b1(c);
12
13      // Benchmark sequence for BENCH mode
14      if (c.inputManagement.benchFlag)
15      {
16          PSB::Benchmark<PSB::StructOfArrays<std::vector>, PSB::ReadAndWrite<15>> b2(c);
17          PSB::Benchmark<PSB::ArrayOfFloats<std::deque>, PSB::SimulateParticles<100>> b3(c);
18      }
19  }
```

While the first benchmark (`b1` in line 11) is executed for all three modes, the other two (`b2` in line 16 and `b3` in line 17) are only performed when the BENCH mode is activated by the corresponding command line parameter. If benchmark sequences for the other two modes should be defined, the flag `debugFlag` for the DEBUG mode or the flag `visuaFlag` for the VISUA mode have to be utilized in a simple conditional statement similar to line 14, instead of the `benchFlag` for the BENCH mode as shown in the example.

## 3.4 The Developer Perspective

The last kind of software user in our hierarchy is the developer, the user that does not only utilize software as a black box, but also tries to understand the ideas behind the concrete implementation. The developer wants to change and improve the scientific software that is available, and often generates complete new software projects with an own objective in mind. This section discusses all classes of the application in detail, describes all methods and mechanisms and gives examples and new ideas for further extensions.

**3.4.1 Implementation of the `Datastructure` Class.** The `Datastructure` class represents the core of the whole benchmark application. All desired features determine the characteristics of the other classes in the framework. In the following, the general features of the utilized particle stacks are discussed, the concrete implementation of the abstract class from which all data structures are derived is presented and a few examples for custom data structure designs are given. Also specific mechanisms for data type and member function deduction are discussed to illustrate the connections between the Datastructure class and the benchmark execution classes of the application.

### 3.4.2 General Features.

The particle stack has the task to contain all attributes of the simulated particle ensemble, which includes all physical quantities of interest, such as the position, the momentum and the energy of the particles. To maximize the flexibility of the benchmark application, it should be possible to adjust the particle properties of the particle stack to the specific simulation at hand. The idea of this customization can be represented by three measures that should be incorporated into the structure of the data structure class to allow for sophisticated comparison of these designs.

- **Memory Layout of Particle Properties.** Modern hardware architectures, especially the nowadays established complex multilevel memory layouts, demand the possibility of more intricate allocation of memory for the different particle properties. The user should for example have the freedom to combine multiple properties into one single container or to split one single property into a number of containers. Also the possibility of the usage of more than one kind of container should be supported. All in all the memory layout should as much as possible depend only on the choice of the user of the framework.

- **Data Types of Particle Properties.** The data types of the properties depend either on the mathematical model of the physical process or on the level of abstraction of the calculation that is utilized for the simulation. The user should face no limitation regarding the choice of the concrete data type for each attribute. It could be a fundamental data type or a custom defined struct or class.

- **Number of Particle Properties.** Additionally to the data types it should also be possible to modify the absolute number of properties of each particle to adapt the particle stack to the physical model that lies behind the simulation. While the other two measures are realized in the current version of the framework, this feature is currently fixed to 15 particle properties (deduced from the ViennaWD reference implementation in Section 2.2.1) due to the statically typed characteristics of the C++ programming language and the inherent restrictions of the type deduction mechanisms at compile time (see also Section 5.1.1).

### 3.4.3 Abstract Datastructure Class.

All particle stack implementations are derived from the abstract Datastructure class that defines the interface that is necessary to be compatible with the other classes of the application. For further reference, the complete definition of the class is discussed in the following. As already mentioned above, every particle possesses 15 properties that represent either specific physical or numerical quantities. Every property can be modeled by a data type of choice. These options are incorporated into the class definition as template parameters

```cpp
template <typename activeType,
          typename i5dType,
          typename momentumType,
          typename signType,
          typename wp_numType,
          typename i_xType,
          typename j_yType,
          typename valleyType,
          typename positionType,
          typename flightTimeType,
          typename delTType,
          typename energyType>
class Datastructure
{ /* ... */ };
```

which are assigned when the derived class is defined, as can for example be seen in Section 3.3.3 for the `ArrayOfStructs` class. Besides the obligatory constructor and destructor methods

```
public:
    Datastructure() = default;
    virtual ~Datastructure() {};
```

the abstract class contains pure virtual member functions for the access to the particle properties

```
    virtual activeType&     active(long n)      = 0;
    virtual i5dType&        i5d(long n)         = 0;
    virtual momentumType&   momentum_0(long n) = 0;
    virtual momentumType&   momentum_1(long n) = 0;
    virtual momentumType&   momentum_2(long n) = 0;
    virtual signType&       sign(long n)        = 0;
    virtual wp_numType&     wp_num(long n)      = 0;
    virtual i_xType&        i_x(long n)         = 0;
    virtual j_yType&        j_y(long n)         = 0;
    virtual valleyType&     valley(long n)      = 0;
    virtual positionType&   position_0(long n) = 0;
    virtual positionType&   position_1(long n) = 0;
    virtual flightTimeType& flightTime(long n) = 0;
    virtual delTType&       delT(long n)        = 0;
    virtual energyType&     energy(long n)      = 0;
```

that constitute the basic class interface. For every property exists a seperate method that has to be implemented in the derived particle stack. All member functions take an index integer as input parameter, which specifies the specific particle of interest. This uniform input interface simplifies the implementation of the data type deduction in the other classes, as it is described below (see Section 3.4.5). All methods return a reference to the attribute, which allows for the method to be used on both sides of the assignment operator, or in other words, both as $\ell$-value and $r$-value. Additionally there are another three member functions included:

```
    virtual std::string     print()             = 0;
    virtual void            resize(long n)      = 0;
    virtual void            erase(long n)       = 0;
```

The `print()` function returns all particle properties of all particles in the data structure for the generation of the debug file in the DEBUG mode, and the other two functions are necessary for the operations in dynamic test functions where the number of particles can change every time step.

**3.4.4 Custom Data Structure Design.** The definition of the abstract `Datastructure` class allows for a custom memory layout and the free choice of the data types for the particle properties. This design permits the user to generate custom data structures in an efficient way. After the selection of the memory layout all virtual member functions have to be implemented, which makes the data structure compatible with the benchmark execution and the test functions. To illustrate this feature, the significant source code of the three already implemented data structures are presented and discussed in the following. The user can use these examples as incentive for further data structure designs. For brevity only the first of the 15 properties (the `active` flag) is shown here, the other access functions are implemented in a similar way.

➤ **ARRAY OF STRUCTS (AOS).** First we look at the implementation of the ArrayOfStructs class.

```
// Class member variable
ContainerType<Particle> particleArray;
/* ... */
// Inside class constructor
particleArray = ContainerType<Particle>(numberOfParticles, Particle());
/* ... */
// Class member function
char& active(long n) { return particleArray[n].active; }
/* ... */
// Inside resize() method
particleArray.resize(n, Particle());
/* ... */
// Inside erase() method
particleArray.erase(particleArray.begin() + n);
```

The Particle class is a simple nested class inside the ArrayOfStructs data structure that contains the particle properties similar to a classical struct. This example shows that even custom nested classes inside the particle stack can be used for the memory layout. As the name suggests, these Particle objects are arranged inside the container of choice and initialized by the constructor. The access function for the active flag, as well as the resize() and erase() methods, are adjusted to the memory layout.

➤ **STRUCT OF ARRAYS (SOA).** Similar to above is the implementation for the second data structure. This time a seperate container is used for every particle property. Only the first is shown here, in total there are 15 containers implemented. The member functions are again adapted to the concrete implementation.

```
// Class member variable
ContainerType<char> activeArray;
/* ... */
// Inside class constructor
activeArray = ContainerType<char>(numberOfParticles, 0);
/* ... */
// Class member function
char& active(long n) { return activeArray[n]; }
/* ... */
// Inside resize() method
activeArray.resize(n, 0);
/* ... */
// Inside erase() method
activeArray.erase(activeArray.begin() + n);
```

➤ **ARRAY OF FLOATS (AOF).** Now the same data type for all properties is used, which is float in this case. This can also be seen on the return type of the access function. All attributes are now stored inside one single array, which contains now numberOfParticles times numberOfVariables elements. Again only one single line in every member function has to be changed to adjust the abstract data structure to the new particle stack design.

```
// Class member variable
ContainerType<float> variableArray;
```

```
/* ... */
// Inside class constructor
variableArray = ContainerType<float>(numberOfParticles * numberOfVariables, 0.0);
/* ... */
// Class member function
float& active(long n) { return variableArray[n * numberOfVariables + 0]; }
/* ... */
// Inside resize() method
variableArray.resize(n * numberOfVariables, 0.0);
/* ... */
// Inside erase() method
for (long i = 0; i < numberOfVariables; i++)
   variableArray.erase(variableArray.begin() + (n * numberOfVariables));
```

### 3.4.5 Data Type and Member Function Deduction.

The assignment of the data types of the properties and the concrete implementation of the access member functions in the derived data structure classes is only one half of the implementation to ensure that the interface is compatible with the other classes of the framework. The other half is the deduction of the correct data types and access functions in the methods of the other classes. This section emphasises the techniques that were used to implement this feature. The discussed source code is part of the `Benchmark` and `SimulateParticles` classes, but is described here to demonstrate the connection to the `Datastructure` class definition.

➤ **Data Type Deduction.** In both the `Benchmark` and the `SimulateParticles` classes the data type of the particle properties is derived from the information that is given by the data structure template parameter, here named `DataStructureType`.

```
using activeType = typename std::remove_reference_t<
   decltype(std::declval<DataStructureType>().active(long{}))>;
```

The `std::declval` method (defined in header `<utility>`) *„converts any type to a reference type, making it possible to use member functions in* `decltype` *expressions without the need to go through constructors“* [126], which can be seen in the code above. Since all return types of the access functions are reference types, the `std::remove_reference_t` method (defined in header `<type_traits>`) is used before the data type is saved as a type alias for simplicity. Now the data type of every property is known to the class and can be used accordingly.

➤ **Member Function Deduction for MPI Communication.** The `Benchmark` class uses the MPI library to distribute the particles between the different worker processes (see below in Section 3.4.8). This is performed in the `MPIMEM()` method (see also the later discussion on Algorithm D), which takes `std::function` objects (defined in header `<functional>`) as input parameters.

```
template <typename Type>
void MPIMEM(std::string mode,
   std::function<Type& (long)> completeParticleProperty,
   std::function<Type& (long)> particleProperty)
{ /* ... */ }
```

The first function object is the access function of the particle stack in the master process, the second the access function of the particle stack in the specific worker process. The access functions are called via the method `std::bind`, which *„generates a forwarding call wrapper for the function“* [127], as can be seen here

```
MPIMEM<activeType>(mode,
    std::bind(&DataStructureType::active, &completeDataStructure, _1),
    std::bind(&DataStructureType::active, &dataStructure, _1));
```

*„Calling this wrapper is equivalent to invoking the function with some of its arguments bound to* ***args***“ (arguments of `std::bind`) [127], which are represented by the `std::placeholders::_1` object. This mechanism allows the `MPIMEM()` method to call the correct access functions when assembling the arrays for the communication between the processes.

➤ **MEMBER FUNCTION DEDUCTION FOR CUDA COMMUNICATION.** Similar to the MPI communication in the `Benchmark` class the `SimulateParticles` class utilizes the communication supported by the CUDA library to send and receive data between the CPU and the GPU. The `CUDAMEM()` method (Algorithm K in Section 3.4.13) again needs to bind the correct access function for every particle property to the specific function object. Therefore, the method

```
template <typename Type>
void CUDAMEM(std::string mode, Type** cuda_array,
    std::function<Type& (long)> particleProperty)
{ /* ... */ }
```

is called by

```
CUDAMEM<activeType>(mode, &cuda_activeArray,
    std::bind(&DataStructureType::active, &dataStructure, _1));
```

for the `active` flag. The other particle properties are called in a similar way. The communication in the case of HIP is performed by the function `HIPMEM()`, which possesses the same structure as the here presented `CUDAMEM()` method.

**3.4.6 IMPLEMENTATION OF THE `Benchmark` CLASS.** This class manages the MPI communication between the different MPI processes, runs and benchmarks the selected test function and generates the output files which are used for further post processing steps by other applications or scripts. The class takes, as already mentioned in Section 3.3.6, two template parameters as input. The `DataStructureType` selects which data structure class should be used and `TestFunctionType` selects the benchmarked test function class.

**3.4.7 BENCHMARK ALGORITHM.** All benchmarkTestfunction() methods (Algorithm A) are called inside the `Benchmark` class constructor. This means that the instantiation of the class is also automatically performing the benchmarks. The constructor executes all benchmarks for the chosen test function, depending on the preprocessor definitions that select the used implementations. The OpenMP implementation for example is called by

```
#ifdef USE_OPENMP
    benchmarkTestfunction<typename TestFunctionType::OPENMP<DataStructureType>>();
#endif
```

where the corresponding preprocessor variable definition of USE_OPENMP is visible. The other implementations are called in a similar way. This function uses the MPI_Wtime() function to measure the runtime of the selected test function, which is executed inside a time loop. To yield a mean value of the runtimes, the resulting value is divided by the number of average runs before it is stored. After the execution, the progress is printed on the command line. If the BENCH flag is set, the names of the test function, the data structure and the implementation, together with the runtime, are written to the benchmark output file. The function runTestfunction() (Algorithm B) between the MPI_Wtime()

**Algorithm A** Function benchmarkTestfunction() which benchmarks the selected test function.

```
start ← MPI_Wtime()                                    ▷ Synchronized with MPI_Barrier()
runTestfunction()                                      ▷ Algorithm B
end ← MPI_Wtime()                                      ▷ Synchronized with MPI_Barrier()
runtime ← (end − start) / numberOfAverageRuns          ▷ Calculate averaged runtime
IF mpiRank ∈ {0} THEN                                  ▷ If master MPI process
    writeOutputFile(runtime)                           ▷ Write runtime to text file
    printProgress(runtime)                             ▷ Print runtime to command line
END IF
```

function calls first utilizes the setMemberVariables() function that sets the correct number of particles for all MPI processes. The complete number of particles (set by the PAR input parameter) is divided by the number of MPI processes. If there is a non-zero remainder of the division present, then these remaining particles are added to the last MPI process with the highest rank index. At last the setMpiParticleLengths() function is called to broadcast the calculated number of particles to the other processes. Then it initializes the global data structure class (completeDataStructure) with all particles on the master process, a smaller data structure class (dataStructure) with a chunk of the particles on the master and all worker processes and the test function with the selected implementation type. Then the data is sent from the master to the worker processes, where the testfunction is initialized, executed in the time loop and post processed. If the VISUA flag is set, then the visualization output file is written in every time step. After the execution the resulting data is sent back from the workers to the master process. If the DEBUG flag is set, the results are checked for correctness and written to a debug output file. The names of the test function, the data structure and the implementation are returned for the process printing function and the benchmark output file.

**Algorithm B** Function runTestfunction() which runs the selected test function inside a time loop.

```
setMemberVariables()                         ▷ Set the correct number of particles for all MPI processes
IF mpiRank ∈ {0} THEN                        ▷ If master MPI process
    DataStructureType completeDataStructure  ▷ Initialize global data structure object
END IF
DataStructureType dataStructure              ▷ Initialize local data structure object
ImplementationType testFunction              ▷ Initialize test function object
MPIALL(SEND)                                 ▷ completeDataStructure → dataStructure, Algorithm C
testFunction.initialization()                ▷ Initialization of testFunction, see Section 3.4.12
FOR $a \in \{0, \dots,$ numberOfAverageRuns$\}$ DO   ▷ numberOfAverageRuns set by AVG parameter
    FOR $t \in \{0, \dots,$ numberOfTimeSteps$\}$ DO ▷ numberOfTimeSteps set by TIM parameter
        testFunction.run()                   ▷ Execution of testFunction, see Section 3.4.12
    END FOR
END FOR
testFunction.postProcessing()                ▷ Post-processing of testFunction, see Section 3.4.12
MPIALL(RECV)                                 ▷ completeDataStructure ← dataStructure, Algorithm C
```

**3.4.8 MPI Memory Management.** The MPIALL(MODE) (Algorithm C) function manages the communication between the different MPI processes. First the setMpiParticleLengths() function is called which broadcasts the number of particles from every MPI process to all other processes and saves them in the `mpiParticleLengths` array. Additionally the number of all particles in the application is determined by summing all elements in the `mpiParticleLengths` array. This means that the lengths of the particle arrays on each process are communicated to all other processes and especially to the master process to guarantee no segmentation faults if the data is transferred back to the global data structure. Then the MPIMEM(MODE) functions are called for all particle properties to send or receive the selected data to the other processes. The mode parameter determines the MPI communication direction. If mode equals `"_SEND"`, the data is sent from the master process to the worker processes. If mode equals `"_RECV"`, the data is sent from the worker processes back to the master process.

**ALGORITHM C** Function MPIALL(MODE) which manages the MPI memory requirements for the test function execution.

```
setMpiParticleLengths()                    ▷ Broadcasts the number of particles to all other MPI processes
IF mpiRank ∈ {0} THEN                      ▷ If master MPI process
    completeDataStructure.resize()         ▷ Resize the global data structure to the new length
END IF
MPIMEM(MODE)                               ▷ First particle property, Algorithm D
/* ... */                                  ▷ Call MPIMEM(MODE) for all particle properties in between
MPIMEM(MODE)                               ▷ Last particle property, Algorithm D
```

The MPIMEM(MODE) function (Algorithm D) manages the extraction and assignment of the particle properties for the MPI communication. It collects the particle property data from the given data structure class in an allocated array and sends or receives this data via the MPI_Send() and MPI_Recv() functions. If the data is sent, the complete array from the master process is divided into smaller chunks for the different worker processes. If the data is received, the smaller worker arrays are combined together in the complete array on the master process. This functionality is achieved with the help of the index and length variables which are defined inside the method. The `std::function<Type& (long)>` object completeParticleProperty($n$) is bound to the function that manages the particle property for the master process and particleProperty($n$) is bound to the function that manages the particle property for the worker processes, as described in Section 3.4.5. When functions from the MPI library, such as

```
MPI_Send(mpi_array, length, mpi_get_type<Type>(), rank, 0, MPI_COMM_WORLD);
```

are called, the deduced data type for the particle property is only available as a fundamental data type in standard C$^{++}$ syntax, but the MPI library uses special data types that are defined inside the library. Therefore, the function `mpi_get_type<Type>()` is called which receives a fundamental data type as a template parameter and searches in an `if-else`-construct for the equivalent MPI data type which can be used in the functions from the MPI library. This mechanism is achieved with the help of the `std::is_same` method (defined in header `<type_traits>`). For example the conditional statement for the fundamental `char` data type is implemented as follows

```
MPI_Datatype mpi_type = MPI_DATATYPE_NULL;
/* ... */
if (std::is_same<T, char>::value) { mpi_type = MPI_CHAR; }
/* ... */
return mpi_type;
```

**Algorithm D** Function MPIMEM(MODE) which manages the MPI memory requirements for the individual particle properties.

```
index ← 0                                                        ▷ Initialize index variable
FOR rank ∈ {0, ..., numberOfMpiRanks} DO
    length ← mpiParticleLengths[rank]          ▷ Initialize length variable for specific MPI process
    IF MODE ∈ {SEND} THEN                           ▷ Master MPI process → Worker MPI process
        IF mpiRank ∈ {0} THEN                                        ▷ If master MPI process
            FOR $n$ ∈ {index, ..., index + length} DO
                mpi_array[$n$ − index] ← completeParticleProperty($n$)  ▷ Extract particle properties
            END FOR
            IF mpiRank ∉ {0} THEN                                    ▷ If worker MPI process
                MPI_Send(mpi_array, rank)             ▷ Master sends MPI array to worker process
            END IF
            index ← index + length                   ▷ Increment index variable for next rank
        END IF
        IF mpiRank ∈ {rank} THEN
            IF mpiRank ∉ {0} THEN                                    ▷ If worker MPI process
                MPI_Recv(mpi_array, 0)          ▷ Worker receives MPI array from master process
            END IF
            FOR $n$ ∈ {0, ..., length} DO
                particleProperty($n$) ← mpi_array[$n$]              ▷ Obtain particle properties
            END FOR
        END IF
    ELSE IF MODE ∈ {RECV} THEN                      ▷ Master MPI process ← Worker MPI process
        /* ... */                                   ▷ Reverse operations as in SEND mode
    END IF
END FOR
```

**3.4.9 Implementation of the SimulateParticles Class.** The Benchmark class calls the three execution methods initialization(), run() and postProcessing() inside the time loop of the runTestfunction() (see Algorithm B) method. The remainder of this section describes the structure behind these functions by the example of the SimulateParticles() test function. Emphasis lies on the relation between the original Wigner signed-particle algorithm and the extreme simplifications that are incorporated into the SimulateParticles() test function. Also the possibility of custom test function design is discussed at the end (see Section 3.4.14) to illustrate another feature to maximize the flexibility of the application.

**3.4.10 Relation to the Wigner Signed-Particle Method.** One of the objectives discussed in Section 3.1.2 is the implementation of practically inspired test functions that try to reproduce the memory access of the physical algorithm of interest, in our case the Wigner signed-particle method described in Section 1.2. The simulateParticles() test function incorporates all important modules of this solution approach, namely the initialization, the evolution, the growth prediction and the annihilation modules (see Section 3.4.12). Some mechanisms, for example the broadcast of the annihilation flag between the MPI processes, is relatively similar to the ViennaWD reference implementation, while other modules are extremly simplified, often to such a degree, that the calculations are not representing any physical process anymore. However, physical meaning is – as mentioned above – not the objective of this test function. As long as the read and write operations to the member variables of the particle stack resemble the memory access pattern in the original algorithm to some extent (in terms of order and frequency), the test function can be used to determine the performance of the data structure on which it is applied. Improvements of the seperate modules to converge to operations that resemble real physical processes is one possibility for future extensions of the framework (see also Section 5.1.2).

**3.4.11 RELATION TO THE ReadAndWrite CLASS.** The `ReadAndWrite()` test function possesses the same structure as this test function in terms of parallelization implementations and execution functions, namely the three modules initialization(), run() and postProcessing(), which are called by the `Benchmark` class inside the runTestfunction() method (Algorithm B). The initialization() and postProcessing() functions of `ReadAndWrite()` are only used for CUDAALL() calls (see Section 3.4.13) for data transfer when the GPU is involved in the calculation and the run() function only contains a `for`-loop over all particle properties, where they are read, incremented and written back into the same place in memory. Therefore, it seems sufficient to only discuss the `SimulateParticles()` test function in the following, since almost all implementation details of the `readAndWrite()` test function can be derived from `simulateParticles()` by simplifying specific implementation modules.

**3.4.12 IMPLEMENTATION ON THE CPU.** All three modules are called by every MPI process and run partially in parallel. The implementation called SERIAL is a pure MPI parallelization where the test function is executed serial on every process. The SIMD and OPENMP implementations are a hybrid technique, where every MPI process utilizes the OpenMP library for the parallization of `for`-loops by employing vectorization instructions (SIMD) or multiple threads on the CPU (OPENMP) with the help of `#pragma omp` compiler directives discussed in Section 1.3.2. The `simulateParticles()` function (Algorithm E) and the corresponding methods are discussed in the following. The comments next to the pseudo-code denote if the module is parallelizable or if it is a critical section, which means that it always has to be executed in serial to prevent segmentation faults and race conditions.

**ALGORITHM E** Test function `SimulateParticles()` executed on the CPU only.

```
initialization()                      ▷ Parallelizable section on the CPU
evolution()                           ▷ Parallelizable section on the CPU, Algorithm F
generateParticles()                   ▷ Critical section
growthPrediction()                    ▷ Communication between MPI processes, Algorithm G
IF annihilationFlag ∈ {TRUE} THEN
    annihilation()                    ▷ Parallelizable section on the CPU, , Algorithm H
    eraseParticles()                  ▷ Critical section
    annihilationFlag ← FALSE          ▷ Reset annihilationFlag
END IF
```

➤ **INITIALIZATION.** This module initializes the particle properties for all particles inside the used data structure. This function generates random values and assignes them to the position and momentum variables of all particles, where the position values are located inside a rectangular two-dimensional simulation domain and the momentum values lie in the range between `-1` and `+1`. In contrast to the ViennaWD implementation that uses Gaussian minimum uncertainty wavepackets described by Equation (1.2.1) as an initial condition for the particles, these random values are distributed uniformly with the help of the `std::uniform_real_distribution` class [128] and are generated by the general-purpose pseudo random number generator module `std::mersenne_twister_engine` [129], both defined in the header `<random>`. The random values for the free-flight time and the scatter mechanism probabilities in the evolution module are generated in the same way, only within other ranges.

➤ **EVOLUTION.** This function (Algorithm F) manages the evolution step of the particles. Inside a for-loop the following functions are applied to every particle in the ensemble. The function sampleFreeFlightTime($n$) first checks if the free-flight time parameter is zero. If this is the case, this means that the particle was part of a scattering event in the last time step and a new random value for the next free-flight time is generated and assigned to it. If this is not the case, then the free-flight time is decremented by one to indicate the time period that has expired during the last time step. The ViennaWD implementation differentiates between the remaining time $\delta t$ in the time step $\Delta t$ and the free-flight time $\tau$ (see Equation (1.2.2)). This module simplifies the mechanism by setting the duration of the free-flight time $\tau$ to an integer-multiple of the time step length $\Delta t$ and omits the remaining time $\delta t$ completely. This means that scattering events can only happen at the end of every time step and particles can drift on a trajectory over the duration of multiple time steps.

**Algorithm F** Function evolution() which executes the drift and scattering mechanisms.

```
FOR $n \in \{0, \ldots, \text{numberOfParticles}\}$ DO
    sampleFreeFlightTime($n$)
    IF flightTime[$n$] $\notin \{0\}$ THEN
        drift($n$)
    ELSE IF flightTime[$n$] $\in \{0\}$ THEN
        scattering($n$)
    END IF
END FOR
```

If the free-flight time is not zero, then the particle drifts away according to Newtonian mechanics. The function drift($n$) manages the drift of the particles. This function updates the position values of all particles by adding the momentum value multiplied with the time step length $\Delta t$. This simulates a Newtonian trajectory where the particles do not accelerate and the wave vector remains constant, similar to the ViennaWD implementation (see Section 1.2.2). If the particle reaches the boundary of the simulation area, it is reflected back into the region. If on the other hand the flight time equals zero, a scattering event takes place. The function scattering($n$) manages the scattering events of the particles. This function defines the scattering probability and samples a random number which chooses the next event. If the probability is smaller than the defined value, a phonon scattering event is chosen, otherwise a generation event is executed. The phonon scattering function modifies the momentum values of the particle. In this case it is just multiplying the momentum vector with the value -1, which should imply a simple hypothetical scattering event. This operation only resembles the memory access operation between the scattering module and the particle stack and is not meant to imply a physical process. The SimulateParticles() test function utilizes – in contrast to the ViennaWD implementation – the activation flag not only in the case of an annihilation event (where it is set to a specific value), but also for the marking of particles that are part of a generation event (where it is set to another value as in the annihilation event). In the case of a generation event, this module sets the active flag to another value, but the real generation (which means the addition of new particles to the particle stack) takes place later in the generateParticles() module, because the generation mechanism itself is a critical section and should not be parallelized.

➤ **Generate Particles.** After the first for-loop over all particles in the evolution step a second for-loop occurs where the new particles are generated. This function manages the generation event of the new particles. It resizes the data structure which holds the particles and assignes new initial values to the two new generated particles. The position values are the same as for the old particle, but the momentum values are taken from the old particle and modified by multiplying them with a number smaller than one. Both the free-flight time and the `active` flag are set to zero and the generation number is incremented by one. The so-called generation number denotes the number of generation events that had to be executed to spawn the specific particle. This is only relevant in the case of the visualization of the particle evolution as part of the VISUA mode and determines the color of the particle in the resulting animation. It is, therefore, not present in the ViennaWD implementation. The value of the generation number is stored inside the `i5d` variable, since this index is not used by the ParticleStackBenchmark application. The signs of the new particles are chosen between two opposite values, where for each generation event always two particles with opposite signs are generated. This is a critical section that has to be executed in serial, because the length of the particle stack is modified. Otherwise this step can lead to race conditions or segmentation faults. After the generation of the new particles the activation flag of the original particle is set back to the default value.

- **GROWTH PREDICTION.** This module (Algorithm G) manages the growth prediction for every MPI process. The function checks if the number of particles inside the MPI process has reached the maximal defined value. In contrast to the ViennaWD implementation, this module does not perform a growth prediction based on the maximum value of the generation rate $\gamma$ (see Equation (1.2.3)) to determine the maximal number of particles. This number can be given as the template parameter `maximumOfParticles` to the test function (see Section 3.3.4). In the benchmark measurements of this thesis (see the results in Chapter 4) this value was specifically set to the global number of particles of all MPI processes combined, divided by the number of MPI processes, to enforce annihilation steps in the execution as soon as possible, since the particles are uniformely distributed over all MPI processes.

**ALGORITHM G** Function growthPrediction() which determines if an annihilation step has to be performed in this time step.

```
IF numberOfParticles > maximumNumberOfParticles THEN
    annihilationFlag ← TRUE                                          ▷ Set annihilationFlag
END IF
FOR rank ∈ {0, ..., numberOfMpiRanks} DO
    annihilationFlagArray[rank] ← annihilationFlag        ▷ Store annihilationFlag in array
    MPI_Bcast(annihilationFlagArray[rank], rank)              ▷ Broadcast annihilationFlag
    IF annihilationFlagArray[rank] ∈ {TRUE} THEN
        annihilationFlag ← TRUE                                      ▷ Set annihilationFlag
    END IF
END FOR
```

If the maximum number of particles is reached, then the annihilation flag is set to TRUE. This annihilation flag is then broadcast to all other MPI processes and stored in an array. If at least one of the MPI processes set the annihilation flag to TRUE, then all processes perform an annihilation step. If all annihilation flags are FALSE, then the annihilation step is ignored for this time step.

- **ANNIHILATION.** This function (Algorithm H) manages and performes the annihilation step for the particles. A nested loop over all particles checks three conditions, first if the particles do not have the same index, second if the distance between them is smaller than a defined value $\varepsilon$ and third if the signs of both particles are different. The distance is calculated from the positions of the particles by the Pythagoras theorem. If all these three conditions are true, the `active` flags of both particles are set to the corresponding value for the annihilation flag. Similar to the evolution module this step only sets the activation flags, the actual deletion of the particles is performed in the eraseParticles() method due to the prevention of race conditions.

**Algorithm H** Function annihilation() which determines the particles that have to be erased.

```
FOR $n \in \{0, \dots, \texttt{numberOfParticles}\}$ DO
    FOR $m \in \{0, \dots, \texttt{numberOfParticles}\}$ DO
        IF $n \notin \{m\}$ AND distance $< \varepsilon$ AND sign$(n) \notin \{$sign$(m)\}$ THEN
            setAnnihilationFlag($n$, $m$)
        END IF
    END FOR
END FOR
```

The annihilation step in the ParticleStackBenchmark application is different from the ViennaWD implementation in the sense that here only the position space is taken into account for the determination if the two particles are in the same space cell or not. The ViennaWD implementation additionally is checking if the values of the momentum vector are also close enough to each other, since the Wigner signed-particle method is simulated in the phase space $\{\mathbf{r}, \mathbf{k}\}$ and has to include both quantities.

➤ **Delete Particles.** This function loops over all particles and checks if the `active` flag is set to the specific value of the annihilation event. If this is the case, the particle is deleted from the data structure and the number of the particles in the particle stack is reduced to the new number of particles. The `for`-loop begins at the last element of the particle stack and counts back to the first element to prevent false elimination of particles due to erroneous index assignments if the loop would run in the normal direction. This is, similar to the generateParticles() module, also a critical section that has to be executed in serial, because the length of the particle stack is modified.

Finally, it has to be mentioned that the domain decomposition of the position space (or the simulation area) of the ViennaWD implementation is not incorporated into the ParticleStackBenchmark application. Here, the parallelization is operating only on the number of particles. This means that the particles are distributed before the time loop starts and merged at the end of the execution. In between these two points of time (which includes the entire time loop) the particles are always assigned to the same MPI process, independent of where the particles are currently situated in the simulation space. Therefore, the `SimulateParticles()` test function does not possess an implementation of the particle transfer module of the Wigner signed-particle method, since the particles are not changing the MPI process.

**3.4.13 Implementation on the CPU and the GPU.** If the GPU is incorporated as a parallelization device, the data has to be sent and received between the host (the CPU) and the device (the GPU), similar to the communication between the different MPI processes (see Section 3.4.8). This type of parallelization is implemented in the application via the CUDA and the HIP libraries. If every node on a cluster possesses a CPU and a GPU, one MPI process per node can be spawned, which in turn can now access the GPU on the same node. This kind of hardware is in mind for this implementation (the reason behind this is discussed in Sections 4.1.5 and 5.1.4). The source code is divided into the same modules as for the CPU only case (Algorithm E), but some additional functions are

now necessary, most prominent the functions CUDAALL(MODE) (Algorithm J) and CUDAMEM(MODE) (Algorithm K), which perform in essence the same tasks as the MPIALL(MODE) (Algorithm C) and MPIMEM(MODE) (Algorithm D) functions in the `Benchmark` class. Also the generateAndSendRandomNumbers() function is now part of the test function, and the evolution() and annihilation() modules, which can be parallelized, are realized as so-called kernels, which are suitable for execution on the GPU device.
The hybrid CPU and GPU implementation (Algorithm I) is now discussed in more detail. First the generateAndSendRandomNumbers() is executed. Then the data is sent to the GPU, where the evolution step is performed. Then the data is sent back to the CPU. On the CPU the generation step is performed in serial, since it is a critical section. The growthPrediction() function determines if the annihilation flag is set. If this is the case, the data is again sent to the GPU, where the annihilation step is performed. Then the data is again sent back to the CPU. At last the CPU performes the deletion step, again in serial to prevent race conditions.

**ALGORITHM I** Test function `SimulateParticles()` executed on the CPU and the GPU. The additional functions which are necessary for the GPU communication are colored in blue.

```
generateAndSendRandomNumbers()                      ▷ Generate random numbers on CPU for GPU
CUDAALL(SEND)                                       ▷ CPU → GPU, Algorithm J
evolutionCUDAKernel()                               ▷ Parallelizable section on the GPU, Algorithm F
CUDAALL(RECV)                                       ▷ CPU ← GPU, Algorithm J
generateParticles()                                 ▷ Critical section
growthPrediction()                                  ▷ Communication between processes, Algorithm G
IF annihilationFlag ∈ {TRUE} THEN
    CUDAALL(SEND)                                   ▷ CPU → GPU, Algorithm J
    annihilationCUDAKernel()                        ▷ Parallelizable section on the GPU, Algorithm H
    CUDAALL(RECV)                                   ▷ CPU ← GPU, Algorithm J
    eraseParticles()                                ▷ Critical section
    annihilationFlag ← FALSE                        ▷ Reset annihilationFlag
END IF
```

Besides the modules of the CPU implementation, there are now a few additional functions for the GPU version available, which are now described in more detail. While the cudaMemcpy() functions are represented in the pseudo-code in a non-formal fashion, the cudaMalloc() functions for memory allocation on the GPU are omitted for brevity.

➤ **GENERATE AND SEND RANDOM NUMBERS.** This module manages the random sampling for the free-flight time values and the scattering probabilities. It generates random numbers for these quantities for all particles in advance on the CPU and stores them in seperate arrays. These arrays are then transferred to the memory of the GPU with the help of the cudaMalloc() and cudaMemcpy() functions, where they will be used inside the evolution step module. The random values could also be generated on the GPU itself (which would reduce the communication overhead between the CPU and GPU). The idea behind this specific implementation is to use the same random number generator as in the CPU only case to improve the possibility of comparison between both implementations.

➤ **CUDAALL.** This function (Algorithm J) manages the CUDA memory requirements for the test function execution and the communication between the CPU and the GPU. First the lengths of the particle arrays on each MPI process are communicated to all other MPI processes via the setMpiParticleLengths() function. If the data is sent to the GPU, the number of particles of the particle stack has to be transferred to the GPU as well to be able to know the correct size of the memory block on the device. If the data is received from the GPU, the new number of particles after the calculation on the GPU has to be obtained because the annihilation step on the GPU reduces this number. Then the CUDAMEM(MODE) functions are called for all particle properties to send or receive the selected data on the CPU or GPU. The mode parameter determines the CUDA communication direction. If mode equals `"_SEND"`, the data is sent from the CPU to the GPU. If mode equals `"_RECV"`, the data is sent from the GPU back to the CPU. The calls of the CUDAMEM(MODE) functions, where the correct access functions of the particle attributes have to be bound to `std::function` objects, are discussed in Section 3.4.5.

**ALGORITHM J** Function CUDAALL(MODE) which manages the data transfer between the CPU and the GPU for all particle attributes.

```
setMpiParticleLengths()                          ▷ Broadcasts the number of particles to all other MPI processes
IF MODE $\in$ {SEND} THEN                                                              ▷ CPU $\to$ GPU
    cudaMemcpy(numberOfParticles, HostToDevice)                        ▷ Send number of particles to GPU
ELSE IF MODE $\in$ {RECV} THEN                                                         ▷ CPU $\leftarrow$ GPU
    cudaMemcpy(numberOfParticles, DeviceToHost)                   ▷ Receive number of particles from GPU
END IF
CUDAMEM(MODE)                                                     ▷ First particle property, Algorithm K
/* ... */                                 ▷ Call CUDAMEM(MODE) for all particle properties in between
CUDAMEM(MODE)                                                      ▷ Last particle property, Algorithm K
```

➤ **CUDAMEM.** This method (Algorithm K) manages the CUDA memory requirements for the individual particle properties and therefore the communication between the CPU and the GPU. If the data has to be sent to the GPU, the particle property data from the given data structure class is collected in an allocated CPU array, which is then transferred to an allocated array on the GPU. If the data has to be received, the reverse mechanism takes place. The particleProperty($n$) calls are the corresponding access functions to the member variables of the particle stack, which are given as input parameters (see Section 3.4.5).

**ALGORITHM K** Function CUDAMEM(MODE) which manages the data transfer between the CPU and the GPU for a specific particle attribute.

```
IF MODE $\in$ {SEND} THEN                                                              ▷ CPU $\to$ GPU
    FOR $n \in$ {0, ..., numberOfParticles} DO
        array[$n$] $\leftarrow$ particleProperty($n$)                              ▷ Extract particle properties
    END FOR
    cudaMemcpy(array, HostToDevice)                                      ▷ Send particle attributes to GPU
ELSE IF MODE $\in$ {RECV} THEN                                                         ▷ CPU $\leftarrow$ GPU
    cudaMemcpy(array, DeviceToHost)                                 ▷ Receive particle attributes from GPU
    FOR $n \in$ {0, ..., numberOfParticles} DO
        particleProperty($n$) $\leftarrow$ array[$n$]                               ▷ Obtain particle properties
    END FOR
END IF
```

**3.4.14 CUSTOM TEST FUNCTION DEVELOPMENT.** Similar to the design of custom data structure classes (see Section 3.4.4) it is also possible to develop custom test function classes. The emphasis lies on the modularized structure of the test function classes that allow for expandability. There are two cases possible (see also Section 5.1.2):

- **CUSTOM `SimulateParticle()` MODULES.** The `simulateParticle()` test function is divided into multiple modules which work mostly independent from one another. This structure can be used to enhance the functionality of single or multiple modules. For example more sophisticated calculations, which describe the physical phenomena more accurate, could be implemented. One idea could be to improve the phonon scattering mechanism, which currently is just a simple modification of the momentum vector by multiplication with a constant value.

- **COMPLETE NEW TEST FUNCTIONS.** Another possibility is the development of complete new test functions. All particle stack classes are derived from the abstract `Datastructure` class. This is not the case for the test functions, since there is currently no abstract `Testfunction` class implemented. But similar to the interface with the access functions in the case of the `Datastructure` class, there is also an interface for test functions available, namely three necessary components have to be incorporated into a new test function class to be compatible with the other classes of the application. First there has to be a template parameter with the data type `long` (used for example for `numberOfVariables` or `maximumOfParticles`), then there have to be between one and five nested implementation classes available (SERIAL, SIMD, OPENMP, CUDA, HIP) and finally all implemented nested classes have to contain the three execution methods initialization(), run() and postProcessing() for the time loop execution.

# 4 Results

The applicability of the ParticleStackBenchmark framework – which was described in the last chapter – will now be demonstrated by using it directly as a sophisticated measurement tool for the determination of the best suitable data structure for the directly implemented test functions. First the general setup – which includes the hardware specification, software library versions, MPI process and OpenMP thread configuration and input parameters – is discussed in detail to ensure reproducibility of the presented results. The output of the benchmark measurements are organized in a structured fashion, which utilizes compact table representations and visualization methods, such as diagrams, to simplify the classification and analysis of the large number of output values. These representations are the basis for the subsequent analysis of the results, which concludes in considerations and recommendations regarding the best suitable data structure for implementations of the Wigner signed-particle method. An additional section investigates the upsides and downsides of the two implemented GPU parallelization techniques, since only one of them is currently available on the cluster on which the majority of the benchmark measurements were performed.

## 4.1 General Setup and Output

This section describes the general setup and the resulting output of the performed benchmark measurements. The specification of the hardware and software on which the application was executed is discussed and structured in an organized fashion to ensure reproducibility of the presented results. All configuration and input parameters are listed and explained for the same reason. Lastly, the results of the measurements, which contain a large number of values, are presented and described. An interpretation and discussion of the output is given in Section 4.2.

**4.1.1 Hardware and Software Specification.** All benchmark application executions were performed on the VSC-5 system, which is part of the Vienna Scientific Cluster infrastructure [130]. To guarantee the reproducibility of the presented results, all necessary information of the system, including the utilized CPU and GPU hardware and the operating system, compiler and software library versions are assembled in Table 4.1. The specification details in this section are based on the VSC-5 system documentation [24] and the TOP500 list [131]. The VSC-5 cluster features 770 compute nodes, where 710 of them are pure CPU nodes and the remaining 60 are hybrid CPU/GPU nodes. Each of the hybrid nodes of the VSC-5 cluster is equipped with two AMD EPYC 7713 CPUs and two NVIDIA A100 GPU cards, combined with 512 GB of memory for the CPUs and additionally 40 GB of memory for each GPU. Each CPU has 64 cores available and is operating at a base clock frequency of 2 GHz. Out of the 60 hybrid CPU/GPU nodes 16 nodes were utilized for the benchmark measurements. VSC-5 employs the operating system Alma Linux [132], which is a distribution based on the Linux kernel. The benchmarks utilize three different compilers, namely the g++ compiler from the GNU Compiler Collection (GCC) [123], the CLANG C$^{++}$ compiler [124] and the INTEL C$^{++}$ compiler [125]. For all benchmarks the compiler flags `-std=c++17` (using the C$^{++}$17 standard) and `-O3` (the

TABLE 4.1 VSC-5 Specification, based on [24] and [131].

| AMD CPU | Cores per CPU | Nvidia GPU | Cores per GPU |
|---|---|---|---|
| EPYC 7713 | 64 | A100 | 6912 |
| **Alma Linux** | **GCC** | **CLANG** | **INTEL** |
| 8.5 | 11.2.0 | 12.0.1 | 2021.5.0 |
| **MPI** | **OpenMP** | **CUDA** | **HIP** |
| 2021.4 | 5.0 | 11.5.50 | None |

highest optimization level) were activated. The Intel MPI library [133] was utilized. The `MPI_Wtime()` function, which is included in this library, was used for the resulting runtime determination. For the OpenMP standard the GNU OpenMP implementation (which is part of GCC [134]) was utilized. The CUDA library is distributed and maintained by NVIDIA [135]. One peculiarity is the fact that there is currently no version of the HIP library – which is distributed and maintained by AMD and part of the ROCm software development platform [136] – available on the VSC-5. Therefore, this parallelization library was not compared with the other implementations on the VSC-5. A seperate analysis using HIP measurements, executed on other hardware, is given in Section 4.3.

**4.1.2 PROCESS AND THREAD CONFIGURATION.** As previously mentioned, in total 16 hybrid nodes (CPU and GPU) were utilized for the benchmarks on the VSC-5. The distribution of the spawned MPI processes across the available hardware nodes and the corresponding thread assignment to every process is a very important decision by the user because of the nature of shared ccNUMA memory systems (see Definition 6). To investigate the impact and the ramifications of this choice, three distinct configuration types were selected and benchmarked:

➤ **ONE MPI PROCESS PER CPU.** Since every node contains two CPUs, the configuration with one MPI process for every CPU is realized by 32 processes (2 MPI processes per node; 16 nodes in total) with 64 threads each, represented in the following by the notation 32 Processes × 64 Threads or 32 × 64. This configuration reduces NUMA effects as all threads of a particular MPI process operate within the same NUMA domain of the given CPU.

➤ **ONE MPI PROCESS PER NODE.** This configuration utilizes 16 processes (1 MPI process per node; 16 nodes in total) with 128 threads (combined number of cores of two CPUs) each, represented in the following by the notation 16 Processes × 128 Threads or 16 × 128. This configuration maximizes NUMA effects as half of the threads are assigned to the NUMA domain of the second CPU.

➤ **ONE MPI PROCESS PER NODE WITH SIMULTANEOUS MULTITHREADING.** The here considered CPUs support SMT and as such favor the execution of two threads per core, in this case, 128 threads per CPU and 256 threads per node. This configuration results in 16 processes with 256 threads each, represented in the following by the notation 16 Processes × 256 Threads or 16 × 256. Again only one MPI process per node is active. The NUMA effects related to this configuration are the same as in the 16 × 128 case.

The Slurm Workload Manager, which manages the job schedule on the VSC-5, supports different methods to select these process and thread configurations. The desired results can either be achieved by utilizing a so-called bit-mask, were each thread is activated by flipping the corresponding bit in the mask, or by defining the number of tasks per core and node and additionally the number of processes and threads manually in the job script. For the generation of the results in this thesis the second method was chosen. The debug output option of the MPI library (activated with the command `export I_MPI_DEBUG=4` in the Slurm job script) was utilized to ensure that the correct number of processes and threads were assigned to the corresponding CPUs.

**4.1.3 Application Parameters.** Section 3.2 lists all available options for the execution of the ParticleStackBenchmark application. For the results presented in this section the following options were chosen.

- **Build Options.** All three compiler build parameters `GCC`, `CLANG` and `INTEL` were benchmarked, while only the `VSC5` implementation parameter was activated. The reason behind this choice is the fact that the HIP library is currently not available on the VSC-5. Therefore, only the four parallelization implementations SERIAL, SIMD, OPENMP and CUDA were benchmarked.
- **Mode and Input Parameters.** Since only the main benchmarking feature of the application is used for the results, the `BENCH` mode was activated. Every benchmark execution selected $10^4 = 10\,000$ timesteps in the time loop (`-TIM=4`), $10^6 = 1\,000\,000$ particles in the global particle ensemble (`-PAR=6`) and 5 individual benchmark measurements over which the resulting value is averaged (`-AVG=5`).

**4.1.4 Output Overview.** The following runtime values contain not only the operations of the test functions, but also the communication between the nodes via MPI (see Algorithm D in Section 3.4.8) and the communication between the CPU and GPU via CUDA or HIP (see Algorithm K in Section 3.4.13). The reason behind this decision are the access functions of the data structures (see Section 3.4.5), which are used in the aforementioned operations and are, therefore, also affecting the runtimes. All results are organized in three tables, one table for each benchmarked test function and page, namely `ReadAndWrite(01)` (Table 4.2), `ReadAndWrite(15)` (Table 4.3) and `SimulateParticles()` (Table 4.4); for a discussion on the test functions see Section 3.2.2. This structure enables the possibiliy to compare all configurations and all implementations at once and to select the best suited combination of all available choices. Every row contains the benchmark results for a concrete combination of a data structure, a container and a parallelization technique (also called *implementation*). Every parallelization method is a hybrid implementation with a MPI component, therefore, `SERIAL` stands for example for a hybrid MPI/SERIAL implementation, where MPI communicates between the multiple compute nodes of the cluster and the test function is run in serial on the compute nodes itself. This holds in similar fashion also for the `SIMD`, `OPENMP` and `CUDA` implementations. Since there are three data structures, three containers and four implementations available, there are 36 combinations and, therefore, 36 rows present in every table. Every combination was benchmarked with all three process and thread configurations discussed above (see Section 4.1.2) and all three considered compilers, which results in 9 different runtime values, represented by 9 columns in every table. The fastest of all runtimes in every single row is colored in green and the slowest runtime in red. This emphasises the comparison between the different compilers and configurations (columns) for every specific combination of data structure, container and implementation (rows). The fastest runtime of every column is framed in a blue box, which indicates the fastest data structure for a specific configuration and compiler combination. These runtimes are also plotted in Figure 4.1 (see Section 4.1.6).

**4.1.5 Missing Runtime Values.** One specific benchmark test case is not available in the result tables. This is the exception where the one MPI process per CPU (32 Processes $\times$ 64 Threads) configuration is combined with the `CUDA` implementation. All runtimes for this test case are replaced by a placeholder symbol (**X**). The reason behind this incompleteness is the fact that every node contains not only two CPUs, but also two GPUs. If every CPU has one MPI process attached to them, then every CPU should also use one of the two available GPUs for themself, otherwise there would arise communication inefficiencies and therefore performance drops. This feature requires the utilization of the multiple device programming methods provided by CUDA. Unfortunately these are currently not supported by the ParticleStackBenchmark application and therefore this test case was not possible to execute, since the results would not be reliable. All for the results executed Slurm job scripts defined the number of utilized GPUs per node as one (with the command `#SBATCH --gres=gpu:1`). The feature of multiple GPU devices per node utilization could be one of the possible extensions of the application in the future (see also Section 5.1.4).

**4.1.6 FASTEST IMPLEMENTATION COMPARISON.** The comparison between the different process and thread configurations and chosen compilers for every combination of data structure, container and implementation is possible by the analysis of every row in the runtime tables. The colorization of the fastest and slowest runtime per row supports this endeavor. However, the more important question (and the primary objective of this thesis) is the comparison between the combinations (or the rows) themselfs. This means that not the contained values of the rows, but the contained values of the columns of the tables have to be compared against each other. This task is simplified by Figure 4.1 which contains the fastest (and, therefore, the most interesting) implementations measured by the benchmark runtimes. The selection was achieved by comparing all runtimes in a single column and choosing the fastest implementation. The corresponding runtimes are framed in blue colored boxes in the tables. This was done for all 9 columns of a table, which results in 9 values for every test function. Every bar represents a combination of a data structure, a container and an implementation, as described on the $x$-axis. One has to keep in mind that the parallelization has always a MPI component incorporated, which means that `OPENMP` represents a hybrid MPI/OpenMP and `CUDA` a hybrid MPI/CUDA implementation. The selected process and thread configuration, as well as the chosen compiler, are shown on top of the bar. The color of the bar identifies the represented data structure, namely blue for `AOS`, green for `SOA` and red for `AOF`. The height of the bars constitute the value of the runtime, as described on the $y$-axis. The bars are sorted in a decreasing order from left to right. This means that the slowest implementation is placed on the left side and the fastest combination can be found on the right side of the plot. At the top the results for `ReadAndWrite(01)`, in the middle for `ReadAndWrite(15)` and at the bottom for `SimulateParticles()` can be found and analyzed. A detailed discussion of the results represented in the Tables 4.2, 4.3, 4.4 and Figure 4.1 can be found in Section 4.2.

## 4.2 INTERPRETATION AND DISCUSSION

After the presentation of the results it is now time to analyze the specific runtime output of every test function and deduce a hierarchy of best suited data structures, since this is the defined objective of this thesis. The different configurations and test functions are discussed in detail. Also the memory consumption of the data structures is investigated. At the end further considerations and recommendations regarding the implementation of future test functions is given. The conclusion of the analysis is given at the end of Section 4.3 in Section 4.3.6, which combines all the findings of this Chapter.

**4.2.1 CONFIGURATION ANALYSIS.** First we take a look at the Tables 4.2, 4.3 and 4.4. If we compare the fastest runtimes for every process and thread configuration by emphasising the distribution of the green colored runtimes – which represent the fastest runtime of every row respectively – we can deduce that almost all green values are achieved by utilizing the configuration where one MPI process per CPU is spawned (32 × 64). For both `ReadAndWrite()` test functions it holds that for the majority of test cases the 32 × 64 configuration is the fastest. Only when the `CUDA` implementation is benchmarked,

**Table 4.2** Benchmark results for test function `ReadAndWrite(01)`.

| Data Structure | Container | Implementation | Runtime in seconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 32 Processes × 64 Threads | | | 16 Processes × 128 Threads | | | 16 Processes × 256 Threads | | |
| | | | GCC | CLANG | INTEL | GCC | CLANG | INTEL | GCC | CLANG | INTEL |
| `Array of Structs` | `PSB::Array` | `SERIAL` | 0.457732 | 0.468985 | 0.458598 | 0.887131 | 0.912620 | 0.878505 | 0.893393 | 0.901364 | 0.899631 |
| `Array of Structs` | `PSB::Array` | `SIMD` | 0.413857 | 0.421301 | 0.669194 | 0.803658 | 0.824274 | 0.795725 | 0.825249 | 0.803926 | 0.813960 |
| `Array of Structs` | `PSB::Array` | `OPENMP` | $\boxed{0.139047}$ | 0.963229 | 1.033120 | 0.434735 | 1.685375 | 1.601594 | 1.089784 | 5.490309 | 5.168051 |
| `Array of Structs` | `PSB::Array` | `CUDA` | X | X | X | 0.631669 | 0.541207 | 0.705183 | 0.628786 | 0.641874 | 0.654232 |
| `Array of Structs` | `std::vector` | `SERIAL` | 0.460675 | 0.467224 | 0.458054 | 0.884493 | 0.868285 | 0.891558 | 0.878767 | 0.872817 | 0.878006 |
| `Array of Structs` | `std::vector` | `SIMD` | 0.411120 | 0.426531 | 0.413030 | 0.815021 | 0.798734 | 0.816932 | 0.804451 | 0.805668 | 0.805113 |
| `Array of Structs` | `std::vector` | `OPENMP` | 0.150433 | 0.224255 | 0.271598 | 0.368444 | 0.523172 | 0.326577 | 1.235229 | 3.301992 | 3.469730 |
| `Array of Structs` | `std::vector` | `CUDA` | X | X | X | 0.080370 | $\boxed{0.079859}$ | $\boxed{0.081065}$ | 0.083261 | 0.076005 | 0.086445 |
| `Array of Structs` | `std::deque` | `SERIAL` | 1.308396 | 1.303436 | 1.312692 | 2.541368 | 2.542933 | 2.600217 | 2.552244 | 2.548420 | 2.595579 |
| `Array of Structs` | `std::deque` | `SIMD` | 1.101619 | 1.100817 | 1.115126 | 2.197446 | 2.135427 | 2.174730 | 2.146426 | 2.140804 | 3.411833 |
| `Array of Structs` | `std::deque` | `OPENMP` | 0.237862 | 0.146511 | 0.152596 | 0.493659 | 0.438037 | 0.416689 | 1.270155 | 2.770612 | 2.904901 |
| `Array of Structs` | `std::deque` | `CUDA` | X | X | X | 0.114593 | 0.123776 | 0.122292 | 0.120317 | 0.120947 | 0.121202 |
| `Struct of Arrays` | `PSB::Array` | `SERIAL` | 0.418437 | 0.426145 | 0.424554 | 0.790518 | 0.798071 | 0.795520 | 0.819509 | 0.794432 | 0.796891 |
| `Struct of Arrays` | `PSB::Array` | `SIMD` | 0.337228 | 0.358384 | 0.344066 | 0.700157 | 0.639554 | 0.648908 | 1.710798 | 0.670761 | 0.666639 |
| `Struct of Arrays` | `PSB::Array` | `OPENMP` | 0.152973 | 0.074385 | 0.074836 | 0.348352 | 0.144043 | 0.117542 | 0.861617 | 2.103056 | 2.135650 |
| `Struct of Arrays` | `PSB::Array` | `CUDA` | X | X | X | $\boxed{0.078005}$ | 0.084947 | 0.082160 | $\boxed{0.074742}$ | 0.085065 | 0.087424 |
| `Struct of Arrays` | `std::vector` | `SERIAL` | 0.407608 | 0.422946 | 0.418981 | 0.817209 | 0.805174 | 0.805577 | 1.734317 | 0.791577 | 0.792045 |
| `Struct of Arrays` | `std::vector` | `SIMD` | 0.344902 | 0.334921 | 0.344367 | 0.689955 | 0.674291 | 0.632733 | 0.640049 | 0.642102 | 0.678632 |
| `Struct of Arrays` | `std::vector` | `OPENMP` | 0.155648 | $\boxed{0.073713}$ | $\boxed{0.070775}$ | 0.301390 | 0.146231 | 0.120491 | 0.824047 | 1.709740 | 1.555469 |
| `Struct of Arrays` | `std::vector` | `CUDA` | X | X | X | 0.080582 | 0.081291 | 0.083255 | 0.082499 | $\boxed{0.075505}$ | $\boxed{0.085225}$ |
| `Struct of Arrays` | `std::deque` | `SERIAL` | 1.137080 | 1.039646 | 1.201322 | 2.103521 | 2.020247 | 2.139515 | 2.121276 | 2.055263 | 2.112740 |
| `Struct of Arrays` | `std::deque` | `SIMD` | 0.724899 | 0.734315 | 0.753571 | 1.456397 | 1.750505 | 1.437926 | 1.434787 | 1.749329 | 1.434895 |
| `Struct of Arrays` | `std::deque` | `OPENMP` | 0.193834 | 0.103488 | 0.105834 | 0.324026 | 0.188558 | 0.147975 | 0.870147 | 1.632865 | 1.236739 |
| `Struct of Arrays` | `std::deque` | `CUDA` | X | X | X | 0.094641 | 0.093746 | 0.094814 | 0.090061 | 0.097584 | 0.095964 |
| `Array of Floats` | `PSB::Array` | `SERIAL` | 0.617587 | 1.440998 | 0.626946 | 1.284593 | 1.267080 | 1.274878 | 1.278625 | 1.265397 | 1.298389 |
| `Array of Floats` | `PSB::Array` | `SIMD` | 0.574814 | 0.558241 | 0.574139 | 1.099617 | 1.113973 | 1.085364 | 1.090460 | 1.123925 | 1.089662 |
| `Array of Floats` | `PSB::Array` | `OPENMP` | 0.156245 | 0.085986 | 0.086313 | 0.438668 | 0.398371 | 0.396746 | 1.073461 | 1.229098 | 1.095750 |
| `Array of Floats` | `PSB::Array` | `CUDA` | X | X | X | 0.086124 | 0.089615 | 0.088003 | 0.087741 | 0.085530 | 0.090968 |
| `Array of Floats` | `std::vector` | `SERIAL` | 0.655289 | 0.654750 | 0.658541 | 1.284764 | 1.336666 | 1.328810 | 1.264110 | 1.257457 | 1.268667 |
| `Array of Floats` | `std::vector` | `SIMD` | 0.553791 | 0.555534 | 0.561083 | 1.094388 | 1.088435 | 1.093719 | 1.106735 | 1.102396 | 1.086867 |
| `Array of Floats` | `std::vector` | `OPENMP` | 0.154872 | 0.088455 | 0.086015 | 0.319392 | 0.422412 | 0.418779 | 1.063931 | 1.158445 | 1.008450 |
| `Array of Floats` | `std::vector` | `CUDA` | X | X | X | 0.083536 | 0.086848 | 0.083097 | 0.084337 | 0.084329 | 0.085676 |
| `Array of Floats` | `std::deque` | `SERIAL` | 1.343441 | 1.440598 | 1.351862 | 2.792476 | 2.598803 | 2.789148 | 2.790317 | 2.598782 | 2.789703 |
| `Array of Floats` | `std::deque` | `SIMD` | 0.940017 | 0.944502 | 0.938542 | 1.783632 | 1.859449 | 1.766143 | 1.768265 | 1.857879 | 1.763804 |
| `Array of Floats` | `std::deque` | `OPENMP` | 0.245439 | 0.168551 | 0.170642 | 0.493769 | 0.465485 | 0.423864 | 1.159292 | 0.968661 | 1.036914 |
| `Array of Floats` | `std::deque` | `CUDA` | X | X | X | 0.136056 | 0.137931 | 0.138283 | 0.137876 | 0.138637 | 0.142314 |

**TABLE 4.3** Benchmark results for test function `ReadAndWrite(15)`.

| Data Structure | Container | Implementation | Runtime in seconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 32 Processes × 64 Threads | | | 16 Processes × 128 Threads | | | 16 Processes × 256 Threads | | |
| | | | GCC | CLANG | INTEL | GCC | CLANG | INTEL | GCC | CLANG | INTEL |
| Array of Structs | PSB::Array | SERIAL | 3.116669 | 3.139926 | 3.123519 | 6.236134 | 6.134870 | 6.326255 | 6.230377 | 6.163986 | 6.238513 |
| Array of Structs | PSB::Array | SIMD | 3.076619 | 3.085092 | 3.071778 | 6.226409 | 6.130307 | 6.143784 | 6.201618 | 6.198635 | 6.157267 |
| Array of Structs | PSB::Array | OPENMP | $\boxed{0.237147}$ | 0.192138 | 0.178906 | 0.497347 | 0.487703 | 0.481934 | 1.108498 | 0.871848 | 0.985905 |
| Array of Structs | PSB::Array | CUDA | X | X | X | 0.114883 | 0.119505 | 0.116462 | 0.116223 | 0.114849 | 0.116451 |
| Array of Structs | std::vector | SERIAL | 3.147183 | 3.205678 | 3.126000 | 6.219506 | 6.264529 | 6.315171 | 6.243988 | 6.178398 | 6.241619 |
| Array of Structs | std::vector | SIMD | 3.072508 | 3.123163 | 3.098771 | 7.115581 | 6.159002 | 6.148811 | 6.209414 | 6.179470 | 6.164037 |
| Array of Structs | std::vector | OPENMP | 0.249425 | 0.183376 | 0.181789 | 0.430923 | 0.553254 | 0.514381 | 1.144756 | 0.766570 | 0.884779 |
| Array of Structs | std::vector | CUDA | X | X | X | 0.116055 | 0.116790 | 0.115814 | 0.114090 | 0.114488 | 0.114799 |
| Array of Structs | std::deque | SERIAL | 14.177690 | 14.183935 | 14.193981 | 28.413084 | 28.626732 | 28.374220 | 28.414353 | 28.403722 | 28.411690 |
| Array of Structs | std::deque | SIMD | 14.911833 | 14.892112 | 14.917002 | 29.817708 | 29.711669 | 29.840975 | 29.852844 | 29.828177 | 29.832264 |
| Array of Structs | std::deque | OPENMP | 0.596308 | 0.641538 | 0.613861 | 0.974698 | 0.891122 | 0.907601 | 1.460621 | 1.221803 | 1.311098 |
| Array of Structs | std::deque | CUDA | X | X | X | 0.149692 | 0.154999 | 0.152279 | 0.149875 | 0.154910 | 0.156219 |
| Struct of Arrays | PSB::Array | SERIAL | 3.323874 | 3.369289 | 3.325028 | 6.612227 | 6.763664 | 6.705127 | 6.630714 | 6.737588 | 6.626820 |
| Struct of Arrays | PSB::Array | SIMD | 3.289219 | 3.285429 | 3.288848 | 6.541215 | 6.534361 | 6.566457 | 6.566415 | 6.568734 | 6.561883 |
| Struct of Arrays | PSB::Array | OPENMP | 0.654026 | 0.713173 | 0.722072 | 2.372818 | 2.849406 | 2.798488 | 4.226420 | 3.230773 | 3.537382 |
| Struct of Arrays | PSB::Array | CUDA | X | X | X | $\boxed{0.113933}$ | 0.114356 | 0.115404 | 0.114030 | 0.112152 | 0.115236 |
| Struct of Arrays | std::vector | SERIAL | 3.328596 | 3.343892 | 3.328706 | 6.602922 | 6.597269 | 6.624094 | 6.644959 | 6.634773 | 6.638861 |
| Struct of Arrays | std::vector | SIMD | 3.287704 | 3.293656 | 3.290005 | 6.531261 | 6.548367 | 6.608469 | 6.569319 | 6.556911 | 6.562185 |
| Struct of Arrays | std::vector | OPENMP | 0.649826 | 0.768279 | 0.797697 | 2.867980 | 2.535125 | 2.961164 | 5.203920 | 3.436591 | 3.725362 |
| Struct of Arrays | std::vector | CUDA | X | X | X | 0.114756 | $\boxed{0.114092}$ | $\boxed{0.115153}$ | $\boxed{0.110074}$ | $\boxed{0.110208}$ | $\boxed{0.113681}$ |
| Struct of Arrays | std::deque | SERIAL | 8.679403 | 8.469171 | 8.742727 | 16.680474 | 16.819599 | 16.714022 | 16.665886 | 16.897598 | 16.649501 |
| Struct of Arrays | std::deque | SIMD | 8.397686 | 8.053313 | 8.525329 | 16.422218 | 16.236787 | 16.416944 | 16.407541 | 16.215607 | 16.310581 |
| Struct of Arrays | std::deque | OPENMP | 0.952910 | 0.994305 | 0.927001 | 3.405608 | 2.694437 | 2.648398 | 5.978980 | 3.912891 | 4.142196 |
| Struct of Arrays | std::deque | CUDA | X | X | X | 0.124322 | 0.125133 | 0.122590 | 0.119855 | 0.120043 | 0.123340 |
| Array of Floats | PSB::Array | SERIAL | 4.332103 | 4.620412 | 4.348031 | 8.633983 | 9.254074 | 8.695243 | 8.676253 | 9.243069 | 8.675694 |
| Array of Floats | PSB::Array | SIMD | 3.837673 | 3.829642 | 4.244954 | 7.597779 | 7.601120 | 7.612005 | 7.646457 | 7.606337 | 7.632053 |
| Array of Floats | PSB::Array | OPENMP | 0.257808 | $\boxed{0.171603}$ | $\boxed{0.175150}$ | 0.639425 | 0.613047 | 0.527983 | 1.278056 | 0.943323 | 0.827358 |
| Array of Floats | PSB::Array | CUDA | X | X | X | 0.124473 | 0.124015 | 0.122303 | 0.121558 | 0.120599 | 0.122808 |
| Array of Floats | std::vector | SERIAL | 4.356219 | 4.384214 | 4.376465 | 8.659169 | 8.657746 | 8.680905 | 8.700146 | 8.698281 | 8.673510 |
| Array of Floats | std::vector | SIMD | 3.854321 | 3.837226 | 3.837524 | 7.597645 | 8.356654 | 7.688619 | 7.634566 | 7.621771 | 7.606615 |
| Array of Floats | std::vector | OPENMP | 0.241868 | 0.171789 | 0.178554 | 0.481220 | 0.610383 | 0.549228 | 1.220104 | 0.918931 | 0.847906 |
| Array of Floats | std::vector | CUDA | X | X | X | 0.121288 | 0.127484 | 0.126101 | 0.120043 | 0.120754 | 0.122737 |
| Array of Floats | std::deque | SERIAL | 9.508559 | 9.670529 | 9.526885 | 18.917669 | 19.217253 | 18.932457 | 18.981559 | 19.318114 | 18.914692 |
| Array of Floats | std::deque | SIMD | 9.419043 | 9.419558 | 9.426668 | 18.717035 | 18.740028 | 18.792696 | 18.809197 | 18.817382 | 18.730441 |
| Array of Floats | std::deque | OPENMP | 0.489622 | 0.445981 | 0.495990 | 0.839741 | 0.888250 | 0.843586 | 1.555052 | 1.044985 | 1.151183 |
| Array of Floats | std::deque | CUDA | X | X | X | 0.177997 | 0.180715 | 0.181273 | 0.179347 | 0.173047 | 0.179384 |

**Table 4.4** Benchmark results for test function `SimulateParticles()`.

| Data Structure | Container | Implementation | Runtime in seconds | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 32 Processes × 64 Threads | | | 16 Processes × 128 Threads | | | 16 Processes × 256 Threads | | |
| | | | GCC | CLANG | INTEL | GCC | CLANG | INTEL | GCC | CLANG | INTEL |
| `Array of Structs` | `PSB::Array` | `SERIAL` | 5.331124 | 5.475082 | 4.937634 | 13.475420 | 10.645546 | 11.229779 | 14.662764 | 15.045219 | 11.003683 |
| `Array of Structs` | `PSB::Array` | `SIMD` | 4.456276 | 4.551413 | 4.345412 | 8.331934 | 8.527541 | 11.320744 | 14.196852 | 11.621248 | 8.942292 |
| `Array of Structs` | `PSB::Array` | `OPENMP` | 1.656929 | 1.596196 | 1.607270 | 3.327176 | 3.446201 | 2.518103 | 4.784864 | 3.224199 | 4.114763 |
| `Array of Structs` | `PSB::Array` | `CUDA` | X | X | X | 2.024625 | 2.019987 | 2.062608 | 2.001690 | 1.689898 | 1.737098 |
| `Array of Structs` | `std::vector` | `SERIAL` | 5.316554 | 5.653298 | 5.652371 | 15.009832 | 15.091612 | 11.322432 | 14.247304 | 10.365909 | 10.595783 |
| `Array of Structs` | `std::vector` | `SIMD` | 3.839422 | 4.381981 | 4.522960 | 11.688818 | 10.401151 | 11.366694 | 11.981425 | 12.134481 | 9.072254 |
| `Array of Structs` | `std::vector` | `OPENMP` | $\boxed{1.530713}$ | 1.776325 | 1.582187 | 2.902887 | 1.853137 | 2.714634 | 3.696442 | 3.244535 | 3.841945 |
| `Array of Structs` | `std::vector` | `CUDA` | X | X | X | 1.590693 | 1.746834 | 2.105113 | $\boxed{1.679893}$ | 1.667049 | 2.114228 |
| `Array of Structs` | `std::deque` | `SERIAL` | 12.479969 | 9.723861 | 12.044876 | 37.719940 | 38.916448 | 25.117052 | 26.313136 | 24.840291 | 27.115389 |
| `Array of Structs` | `std::deque` | `SIMD` | 12.061620 | 12.299322 | 14.585397 | 26.004621 | 36.677808 | 24.537716 | 24.363000 | 23.815195 | 36.351527 |
| `Array of Structs` | `std::deque` | `OPENMP` | 1.815832 | 1.865860 | 1.751742 | 3.186296 | 1.841846 | 2.544277 | 2.659702 | 3.291943 | 4.192757 |
| `Array of Structs` | `std::deque` | `CUDA` | X | X | X | 1.828134 | 2.040005 | 1.976306 | 1.918187 | 2.260447 | 1.909636 |
| `Struct of Arrays` | `PSB::Array` | `SERIAL` | 5.121765 | 4.944207 | 4.872241 | 9.996845 | 13.789164 | 10.463962 | 14.069569 | 13.008827 | 14.074064 |
| `Struct of Arrays` | `PSB::Array` | `SIMD` | 4.494763 | 4.624082 | 4.135408 | 11.445183 | 9.148217 | 10.961268 | 9.443676 | 9.575235 | 8.310635 |
| `Struct of Arrays` | `PSB::Array` | `OPENMP` | 1.612535 | 1.555954 | $\boxed{1.372640}$ | 1.306023 | 0.911932 | 0.920160 | 2.565937 | 1.448484 | 1.311291 |
| `Struct of Arrays` | `PSB::Array` | `CUDA` | X | X | X | 1.615404 | 1.778450 | 1.764623 | 2.138220 | 1.749069 | 1.741641 |
| `Struct of Arrays` | `std::vector` | `SERIAL` | 4.468304 | 5.061011 | 4.957840 | 13.100005 | 13.049391 | 21.691707 | 13.220595 | 9.104831 | 9.641598 |
| `Struct of Arrays` | `std::vector` | `SIMD` | 5.083694 | 4.146402 | 4.209819 | 9.246434 | 9.147441 | 8.386445 | 13.053994 | 13.130813 | 9.759229 |
| `Struct of Arrays` | `std::vector` | `OPENMP` | 1.606581 | $\boxed{1.508192}$ | 1.390090 | 1.506844 | $\boxed{0.829616}$ | $\boxed{0.844267}$ | 2.239108 | $\boxed{0.951428}$ | $\boxed{1.017524}$ |
| `Struct of Arrays` | `std::vector` | `CUDA` | X | X | X | 1.851947 | 2.147906 | 1.996769 | 1.968813 | 1.996794 | 1.687569 |
| `Struct of Arrays` | `std::deque` | `SERIAL` | 9.064107 | 9.051824 | 8.925147 | 18.644264 | 18.946930 | 18.677280 | 26.760789 | 18.565405 | 27.950492 |
| `Struct of Arrays` | `std::deque` | `SIMD` | 8.205970 | 8.001376 | 8.304281 | 21.991095 | 24.189879 | 16.852514 | 16.423158 | 23.661990 | 22.450797 |
| `Struct of Arrays` | `std::deque` | `OPENMP` | 1.752750 | 1.671213 | 1.662319 | $\boxed{1.276387}$ | 1.059315 | 1.004321 | 2.215526 | 1.185153 | 1.390503 |
| `Struct of Arrays` | `std::deque` | `CUDA` | X | X | X | 1.766972 | 1.877533 | 1.854629 | 1.764237 | 2.260108 | 1.819025 |
| `Array of Floats` | `PSB::Array` | `SERIAL` | 3.779481 | 3.745582 | 3.800920 | 40.024265 | 52.730134 | 50.418760 | 43.052271 | 43.471049 | 44.904507 |
| `Array of Floats` | `PSB::Array` | `SIMD` | 3.162301 | 7.044269 | 5.417287 | 36.266418 | 42.176508 | 43.840188 | 31.090973 | 35.982775 | 32.244692 |
| `Array of Floats` | `PSB::Array` | `OPENMP` | 3.215605 | 2.803443 | 2.680192 | 6.992914 | 7.079965 | 4.888709 | 7.637238 | 6.541488 | 4.283998 |
| `Array of Floats` | `PSB::Array` | `CUDA` | X | X | X | 6.122040 | 5.870267 | 5.906017 | 6.590609 | 6.615529 | 5.838548 |
| `Array of Floats` | `std::vector` | `SERIAL` | 3.708120 | 3.678987 | 3.611078 | 41.521067 | 46.409881 | 51.845963 | 49.251292 | 49.031777 | 56.847085 |
| `Array of Floats` | `std::vector` | `SIMD` | 8.640221 | 3.146025 | 3.079745 | 40.780423 | 39.602934 | 54.060208 | 36.532806 | 34.704579 | 36.035049 |
| `Array of Floats` | `std::vector` | `OPENMP` | 2.830008 | 2.536741 | 2.754950 | 4.185551 | 4.530164 | 4.316428 | 4.512930 | 3.414711 | 3.162391 |
| `Array of Floats` | `std::vector` | `CUDA` | X | X | X | 7.889313 | 6.843924 | 5.732355 | 5.985912 | 6.131568 | 5.819965 |
| `Array of Floats` | `std::deque` | `SERIAL` | 23.906199 | 16.592667 | 6.207858 | 84.892967 | 92.920757 | 91.243691 | 89.312068 | 98.453161 | 103.982232 |
| `Array of Floats` | `std::deque` | `SIMD` | 19.676385 | 5.605011 | 5.639963 | 77.928292 | 83.527906 | 77.702176 | 83.196668 | 73.677960 | 91.352416 |
| `Array of Floats` | `std::deque` | `OPENMP` | 3.746804 | 3.287081 | 3.513153 | 4.711795 | 5.270743 | 6.119451 | 12.404846 | 9.395570 | 9.744684 |
| `Array of Floats` | `std::deque` | `CUDA` | X | X | X | 8.778522 | 8.695047 | 10.330003 | 7.197701 | 8.244961 | 7.874044 |

the runtime is faster for the 16 × 128 or 16 × 256 configurations, but this is inevitable since currently no CUDA parallelization is available for the 32 × 64 configuration. Only in the case of the last test function, namely `SimulateParticles()`, with the utilization of the `StructOfArrays` data structure and the OPENMP implementation, it happens that the 16 × 128 configuration is faster. The distribution of the fastest runtimes according to the utilized configuration are 78 runtimes for the 32 × 64, 12 runtimes for the 16 × 128 and 18 for the 16 × 256 configuration, where one has to keep in mind that the CUDA option for 32 × 64 was not included. If the CUDA implementation would be added to the application, it seems possible that the number of best runtimes for the 32 × 64 configuration would increase even more. Therefore, it can safely be assumed that the 32 × 64 configuration provides the best performance and should be chosen for further implementations. This is because of NUMA overhead in the case of the 16 × 128 configuration, as threads being executed on the second CPU would have to face memory overhead when accessing data residing in the domain of the first CPU. This effect is further multiplied if SMT in the case of the 16 × 256 configuration is activated, which doubles the number of operating threads and the communication between them and, therefore, between both CPUs. The results in the Tables 4.2, 4.3 and 4.4 show that the performance loss from these communication operations is very significant and has to be taken into account.

**4.2.2 `ReadAndWrite(01)` ANALYSIS.** Six of the nine implementations in Figure 4.1 are utilizing the `StructOfArrays` data structure. Even more significant is the fact that the five fastest implementations are all using the `StructOfArrays` class, which leads to the conclusion that `StructOfArrays` is the best suited data structure. Regarding the containers we see that `PSB::Array` and `std::vector` are represented, but `std::deque` is missing. The two fastest implementations use `std::vector`. The fastest parallelization is OPENMP, which could be connected to the fact that the workload of `ReadAndWrite(01)` is minimal (only one single particle property is read and written). Therefore, it is faster to let the CPU itself execute the operations, instead of transferring the data between the CPU and GPU, before the calculation can be performed. But, it should be noted that the CUDA implementation is not available for the 32 × 64 configuration and that this implementation could be even faster. Both the CLANG and INTEL compilers show better performance for OPENMP, while the GCC compiler is faster in combination with CUDA. For this test function the 32 × 64 configuration is the fastest, which coincides with the general analysis of the process and thread configurations above. All discussed results considered, the recommended combination of data structure and container for this test function is `StructOfArrays` with `std::vector`.

**4.2.3 `ReadAndWrite(15)` ANALYSIS.** The middle of Figure 4.1 contains the fastest implementations for the `ReadAndWrite(15)` test function. The six fastest implementations are utilizing again the `StructOfArrays` data structure and the top three implementations employ the `std::vector` container. This time the CUDA parallelization technique is faster than OPENMP, which could coincide with the increased workload of the `ReadAndWrite(15)` test function (now all 15 particle properties are modified). Therefore, the communication period between CPU and GPU is not that significant in comparison with the calculation period, since more operations have to be performed on the GPU. The GCC compiler shows the best performance if combined with the CUDA implementation, as already mentioned above. Interestingly this time the 16 × 256 configuration seems to be the fastest, but this could again be a consequence of the fact that the CUDA implementation is not available for 32 × 64. Again the `StructOfArrays` and `std::vector` combination seems to be the best option.

**4.2.4 `SimulateParticles()` ANALYSIS.** Similar to the other two test functions above, it can be seen at the bottom of Figure 4.1 that the `StructOfArrays` data structure in combination with the `std::vector` container is the best performing implementation, measured by the yielded benchmark runtimes. The big difference between the design of the `ReadAndWrite()` test functions and the

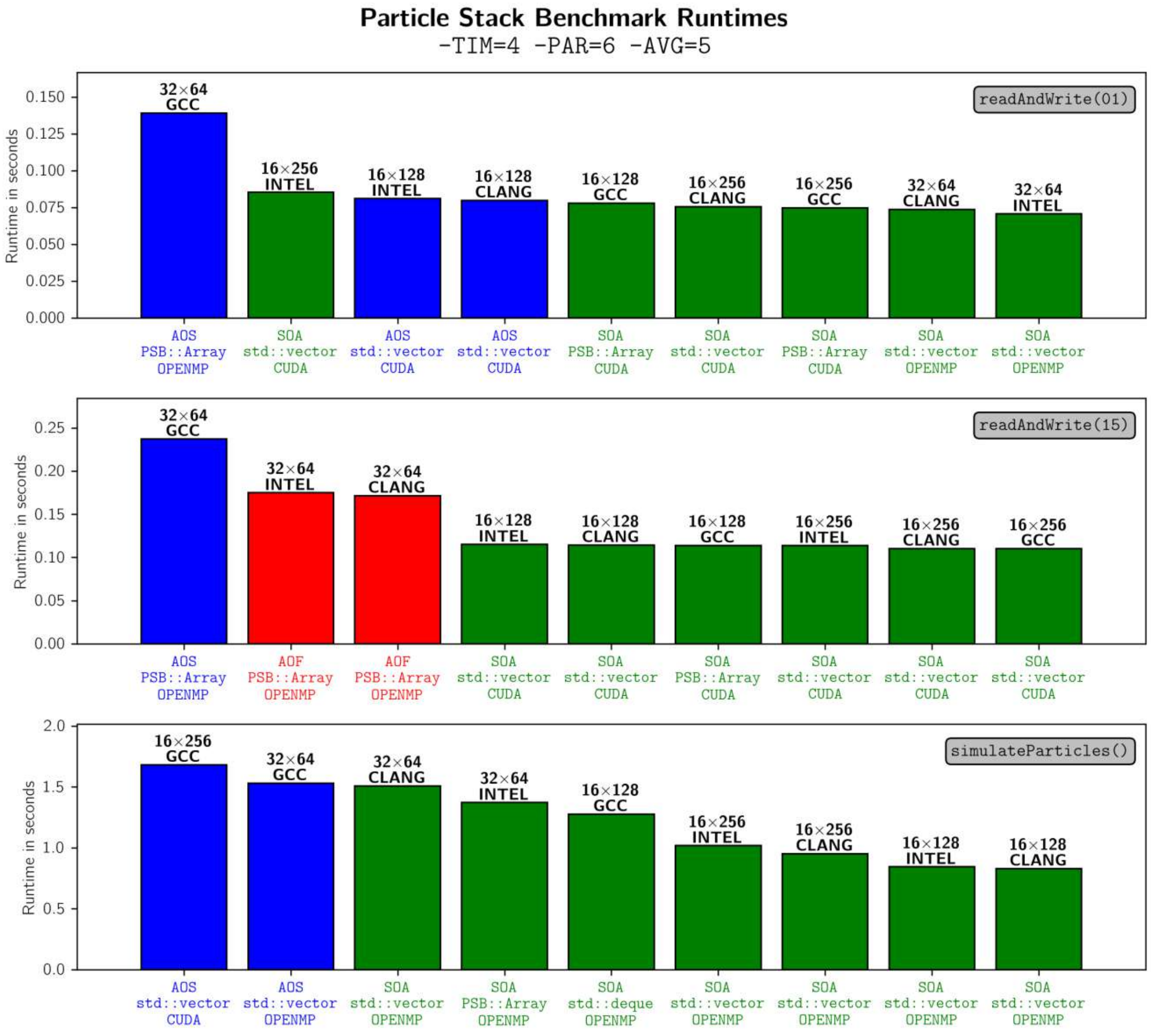

**FIGURE 4.1** Visualization of the fastest data structure implementations for every test function, measured by the benchmark runtimes obtained by the ParticleStackBenchmark application. The results are sorted in descending order from left to right, which means the fastest data structure for every test function can be found on the far right of the bar charts. Additional information can be found in Section 4.1.6.

`SimulateParticles()` test function is the frequent communication between the CPU and the GPU. While the `ReadAndWrite()` test functions only send the data one time at the beginning and one time at the end of the time loop between the CPU and the GPU, the `SimulateParticles()` test function has to communicate every single time step, once for the evolution step, and possibly another time for the annihilation step. This leads to a tremendous communication overload, which is represented by the fact that eight of nine implementations in Figure 4.1 utilize the `OPENMP` parallelization technique instead of `CUDA`. The CLANG and INTEL compilers should be used if the `OPENMP` implementation is employed. The fastest implementation is now achieved by the $16 \times 128$ configuration, which could be the case because the higher number of threads that can be used by the `OPENMP` implementation is more beneficial than the additional communication overload between the two CPUs. All in all we see that `StructOfArrays` with `std::vector` is yet again the winner.

**4.2.5 Memory Consumption.** The `Configuration` class contains a method that determines the size of a data structure in bytes with the help of the `sizeof()` operator and prints it to the command line. If we perform this operation on all available data structures, we obtain the following values in Table 4.5.

Table 4.5 Memory Consumption of the available data structures in bytes.

| Data Structure | Container | Size in Bytes |
|---|---|---|
| ArrayOfStructs | PSB::Array | 88 |
| ArrayOfStructs | std::vector | 72 |
| ArrayOfStructs | std::deque | 128 |
| StructOfArrays | PSB::Array | 648 |
| StructOfArrays | std::vector | 408 |
| StructOfArrays | std::deque | 1248 |
| ArrayOfFloats | PSB::Array | 96 |
| ArrayOfFloats | std::vector | 80 |
| ArrayOfFloats | std::deque | 136 |

The results show that both `ArrayOfStructs` and `ArrayOfFloats` require about 6 to 10 times less memory than `StructOfArrays`. This fact can be explained by the memory layout of the different data structures in mind. While both `ArrayOfStructs` and `ArrayOfFloats` data structures only utilize one single container for the memory allocation, the `StructOfArrays` data structure needs one container for every particle property, in our case 15 containers in total. This fact can be illustrated with, for example, the difference between `ArrayOfStructs<std::vector>` (72 bytes) and `StructOfArrays<std::vector>` (408 bytes). Since SOA possesses 14 additional containers – in this case `std::vector` with three pointers as member variables (in total 24 bytes) – in comparison to AOS, the number of bytes results in $72 + 14 \cdot 24 = 408$ bytes. From the point of view of the containers we see that both the `PSB::Array` and `std::vector` containers need about two thirds of the memory of the `std::deque` container. The reason behind this result is the more complicated design (which results in a larger number of member variables) of the `std::deque` in comparison to the other two containers (see Section 2.1.5).

**4.2.6 Further Considerations and Recommendations.** A few additional considerations and recommendations concerning the implementation of the test functions are now mentioned. Some of the weaknesses of the parallelization methods described above could be prevented by changing the design of the particle distribution inside the test functions. The OPENMP implementation could be extended by not using only one single data structure per MPI process, but multiple particle stacks per MPI process by generating thread specific data structures, where every thread possesses its own data structure (see also Section 5.1.3). This feature would make it possible to parallelize the growing operations from the generation events and the shrinking operations from the annihilation events, since now all potential race conditions and segmentation faults are circumvented. On the other hand there could be overhead at the end of every time step, where all these thread specific data structures have to be merged together to one MPI process specific data structure for further operations. Another recommendation would be the possibility to allocate the memory of the data structure combinations not only on the CPU, but also on the GPU itself (see also Section 5.1.4). Instead of the set of arrays that are transferred back and forth between the host and the device, all modules of the time step could now be performed on the GPU itself, where of course still special attention has to be spent on the critical sections with potential race conditions. This would prevent almost all of the communication overhead and therefore reduce the runtime of the test functions significantly.

## 4.3 GPU Libraries Comparison

Since the HIP library is not available on the VSC-5, another hardware, namely the personal workstation of the author, was chosen to measure the runtime of this implementation. This section describes, similar to the sections above, the general setup of the benchmark execution, the output and the discussion of the results. The emphasis lies on the comparison between the CUDA and the HIP implementation, since both are the libraries for GPU utilization.

**4.3.1 Hardware and Software Specification.** The workstation contains one CPU, an Intel Core i7 11800H, providing 8 cores and 16 threads with SMT. Since only one CPU is used in the benchmark execution, also only one MPI process is assigned to it. Therefore, no MPI communication is incorporated in the following benchmark results. Also one GPU, a Nvidia GeForce RTX 3060, is part of the workstation. This GPU performs the CUDA and HIP implementations. The hardware and software specifications and versions can be found in Table 4.6. All library implementations are the same as for the VSC-5 specification in Table 4.1, only with different versions. The only new additional library is HIP, which allows for another GPU parallelization implementation besides CUDA.

Table 4.6 Workstation Specification.

| Intel CPU | Cores per CPU | Nvidia GPU | Cores per GPU |
|---|---|---|---|
| Core i7 11800H | 8 | GeForce RTX 3060 | 3584 |
| **Ubuntu** | **GCC** | **CLANG** | **INTEL** |
| 20.04.4 | 9.4.0 | 10.0.0 | 2021.6.0 |
| **MPI** | **OpenMP** | **CUDA** | **HIP** |
| 2021.6 | 4.5 | 11.4.48 | 4.5.2 |

**4.3.2 Process and Thread Configuration.** The specific hardware results in a configuration of 1 process with 16 threads, represented by the notation 1 Process × 16 Threads or 1 × 16.

**4.3.3 Application Parameters.** For the results presented in this section the following options for the ParticleStackBenchmark application were chosen.

➤ **BUILD OPTIONS.** Again all three compiler build parameters `GCC`, `CLANG` and `INTEL` were benchmarked, while now the `ALL` implementation parameter was activated. In contrast to the `VSC5` parameter, which only considers the first four implementations, now all five implementations, including `HIP`, are compiled and ready for measurement.

➤ **MODE AND INPUT PARAMETERS.** The `BENCH` mode was activated for benchmark measurements. Every benchmark execution selected $10^4 = 10\,000$ timesteps in the time loop (`-TIM=4`), $10^5 = 100\,000$ particles in the global particle ensemble (`-PAR=5`) and 5 individual benchmark measurements over which the resulting value is averaged (`-AVG=5`). The only difference to the input parameters in Section 4.1.3 is the number of particles, which is 10 times smaller due to the much weaker computational power of the workstation in comparison to the VSC-5 cluster.

**4.3.4 OUTPUT OVERVIEW.** Table 4.7 contains all results of the performed benchmark executions. The test function, data structure and container are defined for each row. The selected compiler and the GPU implementation (`CUDA` or `HIP`) are defined by the corresponding column. The other implementations (`SERIAL`, `SIMD` and `OPENMP`) are omitted to emphasise the comparison of both GPU implementations. Similar to the tables for the VSC-5 results the fastest runtime of a row is colored in green and the slowest runtime in red. Also the fastest runtime of every column of every test function, similar to above, is framed in a blue box, which indicates the fastest data structure for a specific compiler and `CUDA`/`HIP` combination. These values in blue boxes are plotted in Figure 4.2.

**4.3.5 INTERPRETATION AND DISCUSSION.** The distribution of the fastest runtimes regarding the utilized GPU libraries is the following. From a total of 27 test cases, 23 implementations have the fastest runtime (green values) when `HIP` is utilized. In contrast only 4 runtimes are the fastest if `CUDA` was chosen. 13 of the 23 fastest runtimes were achieved by using the `GCC` compiler, which coincides with the above mentioned positive correlation between the utilization of the `GCC` compiler and GPU libraries. On the other hand 24 runtimes were the slowest version (red values) for the `CUDA` implementation, while only 3 were the slowest for `HIP`. This measurements lead to a very interesting result: Even though a GPU from NVIDIA was used, the native NVIDIA programming library (`CUDA`) is in the majority of the test cases slower than the programming library from another vendor, in this case `HIP` from AMD. From the results above it would be plausible to prefer the `HIP` implementation over `CUDA`, even though a NVIDIA card is used. Of course one has to keep in mind that the results in this section are only performed on a small workstation with only one MPI process and therefore no MPI communication. Further benchmark measurements on a cluster similar to the VSC-5 with different process and thread configurations have to be performed, before more precise statements can be given.

**4.3.6 CONCLUSION AND RECOMMENDATION.** The main objective of this thesis is the choice of the best suited data structure for the implementation of a particle stack. The optimal data structure should perform well in vastly different environments with varying conditions and diverse requirements to give developers a wider range of possible options and choices regarding the implementation aspects. The results in Figures 4.1 and 4.2 show that the `StructOfArrays` data structure provides without exception the fastest runtimes for all benchmarked test functions. One possible explanation for the large difference in performance between `StructOfArrays` and the other data structure classes could lie in the way the data is transferred between the seperate MPI processes and between the CPU and GPU. Both parallelization techniques use simple C-style arrays as data packets that are allocated, filled and then communicated from the source to the destination (see Algorithm D for the MPI and Algorithm K for the CUDA implementation). Since every array can only hold elements of the same data type, all arrays are filled with the same attribute from all particles. This feature leads to the advantage of the `StructOfArrays` data structure. The `StructOfArrays` class is the only out of the three data structures that organizes the data in a way where the particle attributes are „ready for shipping“, since they are already stored consecutive for each property in memory and only have to be shifted in the same order into the array for the MPI or CUDA communication. This situation is optimal for load operations, since every entry in the current cache line can be used. The other data structures store the attributes in a non-consecutive manner, which leads to non-optimal load access patterns when the arrays are filled and, therefore, results in increased execution times. All in all, the data structure `StructOfArrays` in combination with the container `std::vector` shows the best performance for all measured test functions, parallelization techniques, compilers and configurations. The only theoretical downside of this combination is the increased memory consumption of the data structure itself. But since this class size only considers the pointers to the containers and not the particle elements of the containers itself, this downside is insignificant in practical simulations due to the large number of particles in an ensemble.

**Table 4.7** Comparison between the `CUDA` and `HIP` implementations for all three test functions.

| **Test Function** | **Data Structure** | **Container** | **Runtime in seconds** | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | GCC | | CLANG | | INTEL | |
| | | | CUDA | HIP | CUDA | HIP | CUDA | HIP |
| `readAndWrite(01)` | `Array of Structs` | `PSB::Array` | 0.116091 | 0.070767 | 0.140513 | 0.091909 | 0.138887 | 0.071717 |
| `readAndWrite(01)` | `Array of Structs` | `std::vector` | 0.084040 | 0.064734 | 0.117830 | 0.095650 | 0.097543 | 0.071222 |
| `readAndWrite(01)` | `Array of Structs` | `std::deque` | $\boxed{0.082752}$ | 0.067486 | $\boxed{0.102443}$ | $\boxed{0.067560}$ | 0.111095 | 0.081463 |
| `readAndWrite(01)` | `Struct of Arrays` | `PSB::Array` | 0.090326 | 0.064322 | 0.108044 | 0.082125 | 0.139193 | 0.083782 |
| `readAndWrite(01)` | `Struct of Arrays` | `std::vector` | 0.095564 | 0.063988 | 0.134137 | 0.092577 | 0.106538 | 0.091609 |
| `readAndWrite(01)` | `Struct of Arrays` | `std::deque` | 0.091628 | $\boxed{0.062531}$ | 0.130437 | 0.079662 | 0.116310 | 0.082599 |
| `readAndWrite(01)` | `Array of Floats` | `PSB::Array` | 0.088659 | 0.073456 | 0.132609 | 0.108239 | $\boxed{0.096743}$ | 0.066468 |
| `readAndWrite(01)` | `Array of Floats` | `std::vector` | 0.088456 | 0.063556 | 0.126699 | 0.069530 | 0.112472 | 0.066267 |
| `readAndWrite(01)` | `Array of Floats` | `std::deque` | 0.101720 | 0.067650 | 0.114502 | 0.076159 | 0.100775 | $\boxed{0.065632}$ |
| `readAndWrite(15)` | `Array of Structs` | `PSB::Array` | 0.360370 | 0.322608 | 0.364878 | 0.322568 | 0.385759 | 0.323244 |
| `readAndWrite(15)` | `Array of Structs` | `std::vector` | 0.367524 | 0.322573 | 0.374494 | 0.322752 | 0.369175 | 0.322728 |
| `readAndWrite(15)` | `Array of Structs` | `std::deque` | 0.377467 | 0.324289 | $\boxed{0.361549}$ | 0.324983 | 0.372396 | 0.324388 |
| `readAndWrite(15)` | `Struct of Arrays` | `PSB::Array` | 0.363733 | $\boxed{0.322250}$ | 0.364626 | $\boxed{0.322454}$ | 0.377794 | 0.322574 |
| `readAndWrite(15)` | `Struct of Arrays` | `std::vector` | 0.391204 | 0.322258 | 0.389660 | 0.322460 | $\boxed{0.356553}$ | $\boxed{0.322288}$ |
| `readAndWrite(15)` | `Struct of Arrays` | `std::deque` | $\boxed{0.356324}$ | 0.323864 | 0.374155 | 0.323843 | 0.383680 | 0.323596 |
| `readAndWrite(15)` | `Array of Floats` | `PSB::Array` | 0.491371 | 0.438380 | 0.505128 | 0.438654 | 0.499095 | 0.438492 |
| `readAndWrite(15)` | `Array of Floats` | `std::vector` | 0.474463 | 0.438186 | 0.476764 | 0.438614 | 0.489361 | 0.438164 |
| `readAndWrite(15)` | `Array of Floats` | `std::deque` | 0.474183 | 0.439874 | 0.485285 | 0.439444 | 0.492163 | 0.439871 |
| `simulateParticles()` | `Array of Structs` | `PSB::Array` | 2.905858 | 2.957178 | $\boxed{2.066991}$ | 3.119318 | 3.109964 | 3.100532 |
| `simulateParticles()` | `Array of Structs` | `std::vector` | 2.226115 | 2.273706 | 3.128318 | 2.295857 | 2.120903 | 2.985961 |
| `simulateParticles()` | `Array of Structs` | `std::deque` | 2.400249 | 2.437826 | 2.456295 | 1.463314 | 1.494810 | 2.416379 |
| `simulateParticles()` | `Struct of Arrays` | `PSB::Array` | 3.278752 | 4.032866 | 4.156770 | 3.895215 | 4.249961 | 4.100079 |
| `simulateParticles()` | `Struct of Arrays` | `std::vector` | 3.329704 | 4.168434 | 4.075359 | 3.247387 | 3.337892 | 3.352701 |
| `simulateParticles()` | `Struct of Arrays` | `std::deque` | $\boxed{1.481348}$ | $\boxed{2.187278}$ | 2.420544 | $\boxed{1.436203}$ | $\boxed{1.469413}$ | $\boxed{1.447461}$ |
| `simulateParticles()` | `Array of Floats` | `PSB::Array` | 10.932758 | 12.413404 | 11.618637 | 11.804590 | 12.745600 | 11.806431 |
| `simulateParticles()` | `Array of Floats` | `std::vector` | 12.139223 | 10.765740 | 10.695360 | 12.287901 | 11.781323 | 10.304378 |
| `simulateParticles()` | `Array of Floats` | `std::deque` | 4.666979 | 4.602924 | 4.699635 | 4.829878 | 4.898854 | 4.761504 |

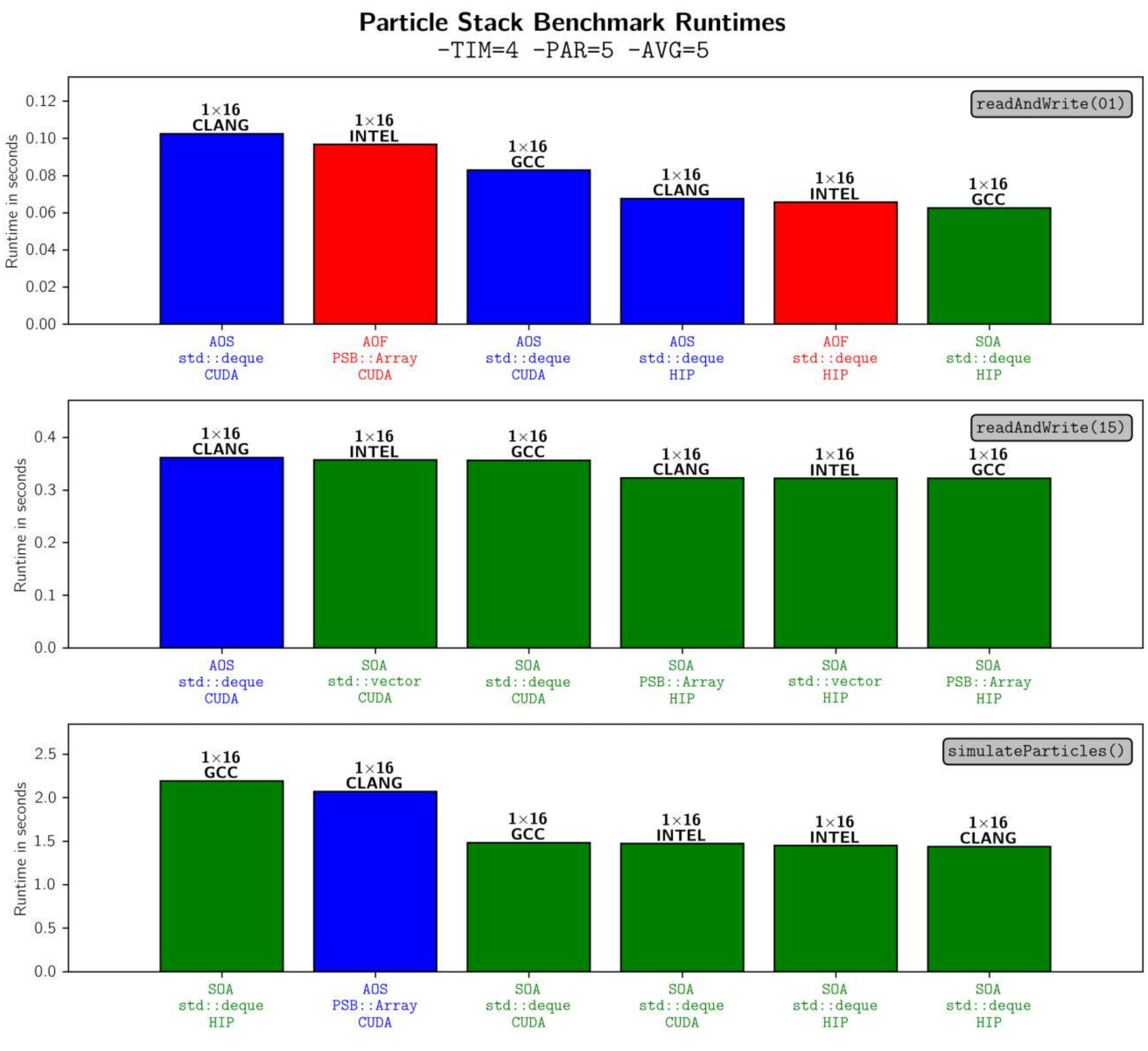

**FIGURE 4.2** Visualization of the fastest data structure implementations utilizing the `CUDA` and `HIP` libraries for every test function, measured by the benchmark runtimes obtained by the ParticleStackBenchmark application. The results are sorted in descending order from left to right, which means the fastest data structure for every test function can be found on the far right of the bar charts.

# 5 Outlook and Conclusion

This chapter describes future extensions which would be interesting possibilities to increase the functionality of the ParticleStackBenchmark application. Further implementation ideas and recommendations for future improvements are presented. These include improvements regarding the already existing data structure and test function classes, but also potential implementation extensions for the parallelization techniques to increase performance. The last section gives a summary of the investigations performed within the scope of this thesis and closes with the final conclusion of the analysis of the benchmark measurement results.

## 5.1 Future Extensions and Outlook

One of the objectives of the ParticleStackBenchmark application is the flexibility and expandability of the framework. This section discusses a collection of possible extensions – some of which were already mentioned throughout the thesis – that would increase the functionality of the framework, raise the investigative potential for future research ventures and lead to further results and insights.

**5.1.1 Additional Data Structure Classes.** The abstract `Datastructure` class (discussed in Section 3.4.3) enables the user to design custom data structures with the freedom to implement different memory layouts (see Section 3.4.4). The three already implemented data structures are conceptualized from the investigations in Chapter 2 and tailored for the utilization in Wigner signed-particle simulations, but this is just a small fraction of all accessible designs. Specifically the following improvements are possible.

- **More Sophisticated Containers and Data Structures.** The first step could be the utilization of better suited containers inside the already implemented data structures. As long as the new container possesses the same interface as the sequential containers in the $C^{++}$ standard library (especially an implementation of the index operator for random element access), only the template parameter for the container type has to be changed. Already existing containers could also be organized into other, more sophisticated data structures, that possess features that could be beneficial for the algorithm at hand. Further investigations regarding other data structures could lead to new ideas and implementations.

- **Adjustable Number of Particle Attributes.** It was already mentioned in Section 3.4.2 that currently the number of physical or numerical properties of every particle is fixed to 15 (this number is deduced from the ViennaWD reference implementation in Section 2.2.1). Since the data types of the properties are changeable, it would also be very useful if the number of properties could be selected by the software user. Unfortunately, the statically typed characteristics of the $C^{++}$ programming language make this endeavor a rather complicated problem regarding the concrete implementation (see Section 3.4.5 for the current data type deduction). The data type of every property has to be deduced from the data structure class definition by the `Benchmark` class for further operations. If the number of particle properties is not known at the time of implementation, it can also not be deduced by the `Benchmark` class, since the data type and member function deduction relies on this information. Another problem is the fact that an adjustable number of particle properties would have to be handled by a loop construct, either by iterating over an array of data types or over an array of `std::function` objects. The problem in the first case is that data types *per se* cannot be saved as elements inside an array, and the problem in the second case is the fact that the `std::function` objects would have different types themselfs for different access functions. Although there are constructs in the language that offer the possibility to store different data types in one single array (for example `std::any` [137], `std::variant` [138] and `std::optional` [139]), the type deducing class still has to know the number, order and the data types itself of the deduced types at compile time. Further investigations regarding different C$^{++}$ language features of interest, for example variadic templates in the context of parameter packs [140], could maybe lead to a possible implementation of this feature in the future.

**5.1.2 Additional Test Function Classes.** Currently there are two test functions available, which are appropriate for benchmark measurements that should help to determine which data structures are best suited for the Wigner signed-particle method. The next step would be to enhance these test functions to yield more precise and practical results. This can be done with the following ideas (see also Section 3.4.14).

- **More Sophisticated Modules.** The `SimulateParticles()` test function contains all important modules that are also present in the Wigner signed-particle algorithm, but the modules itself consist of very basic operations. These operations try to emulate the memory access pattern of the algorithm, but do not really have any physical interpretation (see also Section 3.4.10). It would be very interesting to extend the functionality of these modules to the point where the operations resemble mechanisms that are closer to real physical processes. One example would be the phonon scattering module, which currently just modifies the momentum vector of the particles by multiplying it with a real number. Here more realistic methods such as random number generation for scattering angles could be an upgrade. Another example is the annihilation condition, which currently only considers the position space, but not the momentum space. This condition could be expanded to resemble the algorithm more closely.

- **New Test Functions.** The next step would be the development of completely new test functions that are not based on the two already existing. This would include test functions with more modules than already available or with entire different functionalities. It would also be interesting to not try to emulate the complete Wigner signed-particle algorithm in one single test function, but to divide the algorithm into a number of different test functions and measure the runtimes for all of them in a systematic manner. This could lead to results where different data structures are best suited for different steps of the algorithm. The simulator would then transfer the data between various data structures after the end of the previous module and before the beginning of the next module of the algorithm to yield optimal performance.

**5.1.3 Improved OpenMP Implementation.** The current OpenMP implementation of the test functions is rather basic. The only parallelization technique utilized is the ordinary `#pragma omp parallel for` construct which works on the elements of one single data structure per MPI process. This simple test function structure can be improved by the following ideas.

- **Thread specific Data Structures.** Currently only one single data structure per MPI process is utilized. This means that a large number of OpenMP threads work concurrently on the same elements in the containers. As long as the workload can be distributed between the threads and the operations do not affect the working space of the other threads, this is not a problem. For example the evolution step or the marking of the `active` flag are parallelizable in a straight forward manner. But this is not true for the critical sections in the `generateParticles()` and `eraseParticles()` modules. In this sections the number of particles and, therefore, the length of the containers are changed. If this is done in parallel, it is highly probable that a race condition or a segmentation fault occurs, since a large number of threads is concurrently changing the indices of the remaining particles. One way to solve this problem would be the implementation of thread specific data structures (see also Section 4.2.6). This means that every OpenMP thread possesses a seperate data structure for itself. Now the working space is evenly divided between the threads and the problems discussed above do not arise anymore, which means that all modules, even the critical sections, could now be executed in parallel. Of cource further investigations regarding the splitting and merging of the small thread data structures from and to the large MPI process data structure have to be performed, since this operations could also lead to a performance drop in total.

➤ **ADVANCED OPENMP FEATURES.** The OpenMP library offers a large number of different pre-processor directives that help to optimize the performance. It would be interesting to employ some of these features and keywords to investigate the impact on the runtime measurements. One example would be the utilization of the OpenMP scheduling mechanism that allows for a specific scheduling concept of the iteration partition (such as `static`, `dynamic` and `guided`) when the elements of the parallelized loop construct are processed.

**5.1.4 IMPROVED CUDA IMPLEMENTATION.** Similar to the OpenMP implementation of the test functions there is also further potential for the CUDA implementation. Some enhancement possibilities for the GPU parallelized test functions are now discussed.

➤ **DEVICE MEMORY SPECIFIC DATA STRUCTURES.** Currently the data structures are located on the memory of the host device, namely the CPU. This means that if a module of the test function should be performed in parallel, the data has to be transferred from the CPU to the GPU and back. This communication overhead can lead to tremendous performance drops, especially in the current `SimulateParticles()` implementation, where the data is sent and received every single time step due to the critical sections of the generation and annihilation events. These critical sections have to be performed in serial on the CPU, because the data structures are located in the memory of the CPU itself. The communication overhead could be prevented by the implementation of seperate data structures which are located directly on the device memory, namely the GPU (see also Section 4.2.6). Now all operations can be performed directly on the GPU, even the critical sections, which would then employ only one single thread for the critical operations. Therefore, the communication between the CPU and GPU would only be necessary at the beginning and at the end of the complete time loop, not in every single time step.

➤ **MULTIPLE DEVICE PROGRAMMING.** As already mentioned in the last chapter (see Section 4.1.5), the results indicate that the $32 \times 64$ configuration shows the most promising performance. Unfortunalely, there is currently only the OpenMP implementation available, but the CUDA implementation is still missing. The reason behind this inconvenience is the fact that the current implementation only considers one single GPU as a possible device. This means that both MPI processes on the compute node will utilize the same GPU, while the other GPU would stay idle. The solution to this problem is the multiple device programming feature of the CUDA library. This feature allows to redirect the communication to more than one single GPU device and to manage the transfer between two CPUs (or rather two MPI processes) and two GPUs on the same compute node. It would be very promising to implement this parallelization method for the 32 × 64 configuration, since all results in the last chapter point to the possibility that this implementation could provide even faster runtimes than currently available.

## 5.2 Thesis Summary and Conclusion

This closing section recapitulates the content of the thesis and summarizes the developement of the ParticleStackBenchmark application. The last section gives a final conclusion which is based on the interpretation from the results of the benchmark measurements presented in Chapter 4.

**5.2.1 Thesis Summary.** Within the scope of this thesis a software application was developed to measure the runtime of different test functions while utilizing a variety of distinct data structures. The objective was to determine a hierarchy of the best suited data structures for the execution of the Wigner signed-particle method. The Wigner signed-particle solution approach was introduced and the corresponding operations of the algorithm were described to establish a basic understanding for the peculiarities of the simulation method. One of the challenges of the algorithm is the necessity of a large number of particles inside the simulation ensemble to yield values for the physical quantities of interest that are reliable. Therefore, it is inevitable to utilize modern hardware architectures that provide parallelization methods and other performance boosting techniques. The most important and widely-used architectures and the corresponding parallel APIs and software libraries were discussed. Investigations regarding the design of adequate data structures were performed, starting with the analysis of the algorithmic complexity of the methods of the considered data structures. The ViennaWD reference implementation was consulted to derive the necessary particle properties. Related open source projects, especially Monte Carlo particle simulators, were examined to obtain further possibilities for different implementations. With the algorithm, the hardware and the data structure design in mind, the already mentioned benchmark framework was developed. The software specification including the objectives and requirements was formulated as a basis. The software was described from the point of view of three different software user types, starting with the end user, followed by the advanced user and finished with the developer perspective. The input parameter and output options, the possibility of custom data structure and test function design and the implementation of the class system was described and discussed in detail. This benchmark application was then used for a series of runtime measurements, where a large quantity of different options and parameters were combined to a plethora of test cases. The results of these measurements were organized in a structured manner by combining the large number of values in tabulated representations and diagram visualizations. These results were interpreted and discussed to obtain the best suitable data structures for the solution method at hand. At last a number of future extensions of the framework were mentioned.

**5.2.2 Final Conclusion.** The objective of this thesis was the derivation of a number of data structures, sorted by a specific performance measure – in this case the execution runtime –, which show the most potential for the use inside scientific software that performs Wigner signed-particle simulations. The analysis of the benchmark results, especially the fastest implementations displayed in Figure 4.1, leads to the conclusion that from all combinations of data structures and containers measured, one specific combination shows by far the most potential regarding the ability to execute the desired operations in the shortest runtime. This combination consists of the `StructOfArrays` (SOA) data structure (see Sections 2.3.2 and 3.4.4) with the `std::vector` container [35]. This specific combination performes better than all other measured data structures under vast different environments, including varying compilers, diverging process/thread configurations and diverse parallelization techniques.

# Bibliography

[1] M. Nedjalkov, D. Querlioz, P. Dollfus, and H. Kosina, *Wigner Function Approach*, in: *Nano-Electronic Devices: Semiclassical and Quantum Transport Modeling*, ed. by D. Vasileska and S. M. Goodnick, Springer, 2011, pp. 289–358, doi: 10.1007/978-1-4419-8840-9_5.

[2] J. Weinbub and D. K. Ferry, *Recent Advances in Wigner Function Approaches*, Applied Physics Reviews 5 (2018), p. 041104, doi: 10.1063/1.5046663.

[3] D. E. Stevenson and R. M. Panoff, *Experiences in Building the Clemson Computational Sciences Program*, Proceedings of the ACM Conference on Supercomputing (1990), pp. 366–375, doi: 10.1109/SUPERC.1990.130043.

[4] A. Sameh, G. Cybenko, M. Kalos, K. Neves, J. Rice, D. Sorensen, and F. Sullivan, *Computational Science and Engineering*, ACM Computing Surveys 28.4 (1996), pp. 810–817, doi: 10.1145/242223.246865.

[5] U. Rüde, K. Willcox, L. C. McInnes, and H. D. Sterck, *Research and Education in Computational Science and Engineering*, SIAM Review 60.3 (2018), pp. 707–754, doi: 10.1137/16M1096840.

[6] P. Prabhu, T. B. Jablin, A. Raman, Y. Zhang, J. Huang, H. Kim, N. P. Johnson, F. Liu, S. Ghosh, S. Beard, T. Oh, M. Zoufaly, D. Walker, and D. I. August, *A Survey of the Practice of Computational Science*, Association for Computing Machinery (2011), doi: 10.1145/2063348.2063374.

[7] O. Yasar and R. H. Landau, *Elements of Computational Science and Engineering Education*, SIAM Review 45.4 (2003), pp. 787–805, doi: 10.1137/S0036144502408075.

[8] B. Hendrickson, *Computational Science: Emerging Opportunities and Challenges*, Journal of Physics: Conference Series 180 (2009), p. 012013, doi: 10.1088/1742-6596/180/1/012013.

[9] D. Vasileska and S. M. Goodnick, *Computational Electronics*, Materials Science and Engineering: R: Reports 38.5 (2002), pp. 181–236, doi: 10.1016/S0927-796X(02)00039-6.

[10] Y.-C. Wu and Y.-R. Jhan, *3D TCAD Simulation for CMOS Nanoeletronic Devices*, Springer, 2018, doi: 10.1007/978-981-10-3066-6.

[11] C. K. Maiti, *Introducing Technology Computer-Aided Design (TCAD)*, Pan Stanford Publishing, 2017, isbn: 978-1-315-36450-6.

[12] D. K. Ferry, J. Weinbub, M. Nedjalkov, and S. Selberherr, *A Review of Quantum Transport in Field-Effect Transistors*, Semiconductor Science and Technology 37.4 (2022), p. 043001, doi: 10.1088/1361-6641/ac4405.

[13] J. Weinbub and R. Kosik, *Computational Perspective on Recent Advances in Quantum Electronics: From Electron Quantum Optics to Nanoelectronic Devices and Systems*, Journal of Physics: Condensed Matter 34.16 (2022), p. 163001, doi: 10.1088/1361-648X/ac49c6.

[14] E. WIGNER, *On the Quantum Correction for Thermodynamic Equilibrium*, Physical Review 40.5 (1932), pp. 749–759, DOI: 10.1103/PhysRev.40.749.

[15] P. ELLINGHAUS, *Two-Dimensional Wigner Monte Carlo Simulation for Time-Resolved Quantum Transport with Scattering*, Dissertation, Fakultät für Elektrotechnik und Informationstechnik, Technische Universität Wien, 2016, URL: https://www.iue.tuwien.ac.at/phd/ellinghaus/.

[16] V. SVERDLOV, E. UNGERSBOECK, H. KOSINA, and S. SELBERHERR, *Current Transport Models for Nanoscale Semiconductor Devices*, Materials Science and Engineering: R: Reports 58.6 (2008), pp. 228–270, DOI: 10.1016/j.mser.2007.11.001.

[17] D. K. FERRY and M. NEDJALKOV, *The Wigner Function in Science and Technology*, IOP Publishing, 2018, DOI: 10.1088/978-0-7503-1671-2.

[18] J. M. SELLIER, M. NEDJALKOV, and I. DIMOV, *An Introduction to Applied Quantum Mechanics in the Wigner Monte Carlo Formalism*, Physics Reports 577 (2015), pp. 1–34, DOI: 10.1016/j.physrep.2015.03.001.

[19] L. SHIFREN and D. K. FERRY, *Particle Monte Carlo Simulation of Wigner Function Tunneling*, Physics Letters A 285.3 (2001), pp. 217–221, DOI: 10.1016/S0375-9601(01)00344-9.

[20] ViennaWD – *Wigner Ensemble Monte Carlo Simulator*, URL: https://viennawd.sourceforge.net.

[21] G. HAGER and G. WELLEIN, *Introduction to High Performance Computing for Scientists and Engineers*, Taylor and Francis Group, 2011, ISBN: 978-1-4398-1192-4.

[22] T. STERLING, M. ANDERSON, and M. BRODOWICZ, *High Performance Computing: Modern Systems and Practices*, Elsevier, 2018, ISBN: 978-0-12-420158-3.

[23] VSC-4 *System Specification*, URL: https://vsc.ac.at//systems/vsc-4/.

[24] VSC-5 *System Specification*, URL: https://vsc.ac.at//systems/vsc-5/.

[25] MESSAGE PASSING INTERFACE FORUM, MPI*: A Message-Passing Interface Standard, Version* 4.0, 2021, URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[26] OPENMP ARCHITECTURE REVIEW BOARD, OpenMP*: Application Programming Interface Specification, Version* 5.2, 2021, URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[27] W.-M. W. HWU, *GPU Computing Gems, Emerald Edition*, Elsevier, 2011, ISBN: 978-0-12-384988-5.

[28] NVIDIA, CUDA C$^{++}$ *Programming Guide, Version* 12.0, 2023, URL: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[29] ADVANCED MICRO DEVICES, HIP *Programming Guide, Version* 1.0, 2022, URL: https://github.com/RadeonOpenCompute/ROCm/blob/rocm-4.5.2/AMD_HIP_Programming_Guide.pdf.

[30] D. A. PATTERSON and J. L. HENNESSY, *Computer Organization and Design: The Hardware and Software Interface*, Elsevier, 2014, ISBN: 978-0-12-407726-3.

[31] J. WEINBUB, *Frameworks for Micro- and Nanoelectronics Device Simulation*, Dissertation, Fakultät für Elektrotechnik und Informationstechnik, Technische Universität Wien, 2014, URL: https://www.iue.tuwien.ac.at/phd/weinbub/.

[32] D. E. KNUTH, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison Wesley Longman, 1997, ISBN: 978-0-321-75104-1.

[33] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, and C. STEIN, *Introduction to Algorithms*, The MIT Press, 2009, ISBN: 978-0-262-03384-8.

[34] E. DEMAINE, J. KU, and J. SOLOMON, *Lecture Notes for the Course Introduction to Algorithms*, Massachusetts Institute of Technology, 2020.

[35] *Documentation for* `std::vector`, URL: https://en.cppreference.com/w/cpp/container/vector.

[36] *Documentation for* `std::deque`, URL: https://en.cppreference.com/w/cpp/container/deque.

[37] W. L. JORGENSEN and J. TIRADO–RIVES, *Molecular Modeling of Organic and Biomolecular Systems Using* BOSS *and* MCPRO, Journal of Computational Chemistry 26.16 (2005), pp. 1689–1700, DOI: 10.1002/jcc.20297.

[38] C. AHDIDA et al., *New Capabilities of the* FLUKA *Multi-Purpose Code*, Frontiers in Physics 9 (2022), p. 788253, DOI: 10.3389/fphy.2021.788253.

[39] T. GOORLEY et al., *Features of* MCNP6, Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo (2013).

[40] T. SATO et al., *Features of Particle and Heavy Ion Transport Code System* (PHITS) *version* 3.02, Journal of Nuclear Science and Technology 55.6 (2018), pp. 684–690, DOI: 10.1080/00223131.2017.1419890.

[41] J. LEPPÄNEN et al., *The* Serpent *Monte Carlo Code: Status, Development and Applications in* 2013, Annals of Nuclear Energy 82 (2015), pp. 142–150, DOI: 10.1016/j.anucene.2014.08.024.

[42] E. BRUN et al., TRIPOLI-4*, CEA, EDF and AREVA Reference Monte Carlo Code*, Annals of Nuclear Energy 82 (2015), pp. 151–160, DOI: 10.1016/j.anucene.2014.07.053.

[43] R. HENS et al., Brick-CFCMC*: Open Source Software for Monte Carlo Simulations of Phase and Reaction Equilibria Using the Continuous Fractional Component Method*, Journal of Chemical Information and Modeling 60.6 (2020), pp. 2678–2682, DOI: 10.1021/acs.jcim.0c00334.

[44] BRICK-CFCMC *Website*, URL: https://iraspa.org/brick-cfcmc-a-new-monte-carlo-simulation-code-for-forcefield-based-simulations-in-open-ensembles.

[45] BRICK-CFCMC *Source Code*, URL: https://gitlab.com/ETh_TU_Delft/Brick-CFCMC.

[46] J. K. SHAH et al., Cassandra*: An Open Source Monte Carlo Package for Molecular Simulation*, Journal of Computational Chemistry 38.19 (2017), pp. 1727–1739, DOI: 10.1002/jcc.24807.

[47] Cassandra *Website*, URL: https://cassandra.nd.edu.

[48] Cassandra *Source Code*, URL: https://github.com/MaginnGroup/Cassandra.

[49] T. D. KÜHNE et al., CP2K*: An Electronic Structure and Molecular Dynamics Software Package – Quickstep: Efficient and Accurate Electronic Structure Calculations*, Journal of Chemical Physics 152 (2020), p. 194103, DOI: 10.1063/5.0007045.

[50] CP2K *Website*, URL: https://www.cp2k.org.

[51] CP2K *Source Code*, URL: https://github.com/cp2k/cp2k.

[52] A. V. Brukhno et al., DLMONTE*: A Multipurpose Code for Monte Carlo Simulation*, Molecular Simulation 47.2-3 (2021), pp. 131–151, doi: 10.1080/08927022.2019.1569760.

[53] DLMONTE *Website*, url: https://dl_monte.gitlab.io/dl_monte-tutorials-pages.

[54] DLMONTE *Source Code*, url: https://gitlab.com/dl_monte/DL_MONTE-2.

[55] I. Kawrakow, *Accurate Condensed History Monte Carlo Simulation of Electron Transport. I.* EGSnrc*, the New* EGS4 *Version*, Medical Physics 27.3 (2000), pp. 485–498, doi: 10.1118/1.598917.

[56] EGSnrc *Website*, url: https://nrc-cnrc.github.io/EGSnrc.

[57] EGSnrc *Source Code*, url: https://github.com/nrc-cnrc/EGSnrc.

[58] F. Weik et al., ESPResSo 4.0 *– An Extensible Software Package for Simulating Soft Matter Systems*, European Physical Journal Special Topics 227 (2019), pp. 1789–1816, doi: 10.1140/epjst/e2019-800186-9.

[59] ESPResSo *Website*, url: https://espressomd.org.

[60] ESPResSo *Source Code*, url: https://github.com/espressomd/espresso.

[61] A. J. Schultz and D. A. Kofke, Etomica*: An Object-Oriented Framework for Molecular Simulation*, Journal of Computational Chemistry 36.8 (2015), pp. 573–583, doi: 10.1002/jcc.23823.

[62] Etomica *Website*, url: https://www.etomica.org.

[63] Etomica *Source Code*, url: https://github.com/etomica/etomica.

[64] H. W. Hatch, N. A. Mahynski, and V. K. Shen, FEASST*: Free Energy and Advanced Sampling Simulation Toolkit*, Journal of Research of National Institute of Standards and Technology 123 (2018), p. 123004, doi: 10.6028/jres.123.004.

[65] FEASST *Website*, url: https://pages.nist.gov/feasst.

[66] FEASST *Source Code*, url: https://github.com/usnistgov/feasst.

[67] D. Sarrut et al., *Advanced Monte Carlo Simulations of Emission Tomography Imaging Systems with* GATE, Physics in Medicine and Biology 66 (2021), 10TR03, doi: 10.1088/1361-6560/abf276.

[68] GATE *Website*, url: http://www.opengatecollaboration.org.

[69] GATE *Source Code*, url: https://github.com/OpenGATE/Gate.

[70] J. Allison et al., *Recent Developments in* GEANT4, Nuclear Instruments and Methods in Physics Research Section A 835 (2016), pp. 186–225, doi: 10.1016/j.nima.2016.06.125.

[71] GEANT4 *Website*, url: https://geant4.web.cern.ch.

[72] GEANT4 *Source Code*, url: https://github.com/Geant4/geant4.

[73] J. Bert et al., Geant4-*Based Monte Carlo Simulations on GPU for Medical Applications*, Physics in Medicine and Biology 58 (2013), pp. 5593–5611, doi: 10.1088/0031-9155/58/16/5593.

[74] GGEMS *Website*, url: https://ggems.fr.

[75] GGEMS *Source Code*, url: https://github.com/GGEMS/ggems.

[76] Y. Nejahi et al., GOMC*: GPU Optimized Monte Carlo for the Simulation of Phase Equilibria and Physical Properties of Complex Fluids*, SoftwareX 9 (2019), pp. 20–27, DOI: 10.1016/j.softx.2018.11.005.

[77] GOMC *Website*, URL: https://gomc.eng.wayne.edu.

[78] GOMC *Source Code*, URL: https://github.com/GOMC-WSU/GOMC.

[79] J. A. Anderson et al., HOOMD-blue*: A Python Package for High-Performance Molecular Dynamics and Hard Particle Monte Carlo Simulations*, Computational Materials Science 173 (2020), p. 109363, DOI: 10.1016/j.commatsci.2019.109363.

[80] HOOMD-blue *Website*, URL: https://glotzerlab.engin.umich.edu/hoomd-blue.

[81] HOOMD-blue *Source Code*, URL: https://github.com/glotzerlab/hoomd-blue.

[82] A. P. Thompson et al., LAMMPS *– A Flexible Simulation Tool for Particle-Based Materials Modeling at the Atomic, Meso, and Continuum Scales*, Computer Physics Communications 271 (2022), p. 108171, DOI: 10.1016/j.cpc.2021.108171.

[83] LAMMPS *Website*, URL: https://www.lammps.org.

[84] LAMMPS *Source Code*, URL: https://github.com/lammps/lammps.

[85] R. Kersevan and M. Ady, *Recent Developments of Monte-Carlo Codes* MolFlow+ *and* SynRad+, Proceedings of the International Particle Accelerator Conference (2019), DOI: 10.18429/JACoW-IPAC2019-TUPMP037.

[86] MolFlow+ *Website*, URL: https://molflow.web.cern.ch.

[87] MolFlow+ *Source Code*, URL: https://github.com/szakeetm/molflow_sources.

[88] R. Fingerhut et al., ms2*: A Molecular Simulation Tool for Thermodynamic Properties, Release* 4.0, Computer Physics Communications 262 (2021), p. 107860, DOI: 10.1016/j.cpc.2021.107860.

[89] ms2 *Website*, URL: https://www.ms-2.de/home.html.

[90] ms2 *Source Code*, URL: https://data.mendeley.com/datasets/nsfj67wydx/3.

[91] T. Misawa et al., mVMC *– Open-Source Software for Many-Variable Variational Monte Carlo Method*, Computer Physics Communications 235 (2019), pp. 447–462, DOI: 10.1016/j.cpc.2018.08.014.

[92] mVMC *Website*, URL: https://www.pasums.issp.u-tokyo.ac.jp/mvmc/en.

[93] mVMC *Source Code*, URL: https://github.com/issp-center-dev/mVMC.

[94] P. K. Romano et al., OpenMC*: A State-of-the-art Monte Carlo Code for Research and Development*, Annals of Nuclear Energy 82 (2015), pp. 90–97, DOI: 10.1016/j.anucene.2014.07.048.

[95] OpenMC *Website*, URL: https://docs.openmc.org/en/stable.

[96] OpenMC *Source Code*, URL: https://github.com/openmc-dev/openmc.

[97] P. Eastman et al., OpenMM 7*: Rapid Development of High Performance Algorithms for Molecular Dynamics*, PLoS Computational Biology 13.7 (2017), e1005659, DOI: 10.1371/journal.pcbi.1005659.

[98] OpenMM *Website*, URL: https://openmm.org.

[99] OpenMM *Source Code*, URL: https://github.com/openmm/openmm.

[100] J. KIM et al., QMCPACK*: An Open Source ab initio Quantum Monte Carlo Package for the Electronic Structure of Atoms, Molecules and Solids*, Journal of Physics: Condensed Matter 30 (2018), p. 195901, DOI: 10.1088/1361-648X/aab9c3.

[101] QMCPACK *Website*, URL: https://qmcpack.org.

[102] QMCPACK *Source Code*, URL: https://github.com/QMCPACK/qmcpack.

[103] D. DUBBELDAM et al., RASPA*: Molecular Simulation Software for Adsorption and Diffusion in Flexible Nanoporous Materials*, Molecular Simulation 42.2 (2016), pp. 81–101, DOI: 10.1080/08927022.2015.1010082.

[104] RASPA *Website*, URL: https://iraspa.org/raspa.

[105] RASPA *Source Code*, URL: https://github.com/iRASPA/RASPA2.

[106] M. A. KOWALSKI et al., SCONE*: A Student-Oriented Modifiable Monte Carlo Particle Transport Framework*, Journal of Nuclear Engineering 2.1 (2021), pp. 57–64, DOI: 10.3390/jne2010006.

[107] SCONE *Website*, URL: https://scone.readthedocs.io.

[108] SCONE *Source Code*, URL: https://bitbucket.org/Mikolaj_Adam_Kowalski/scone.

[109] S. PLIMPTON et al., *Crossing the Mesoscale No-Man's Land via Parallel Kinetic Monte Carlo*, Sandia National Laboratories, 2009.

[110] SPPARKS *Website*, URL: https://spparks.github.io.

[111] SPPARKS *Source Code*, URL: https://github.com/spparks/spparks.

[112] J. A. RACKERS et al., Tinker 8*: Software Tools for Molecular Design*, Journal of Chemical Theory and Computation 14.10 (2018), pp. 5273–5289, DOI: 10.1021/acs.jctc.8b00529.

[113] Tinker *Website*, URL: https://dasher.wustl.edu/tinker.

[114] Tinker *Source Code*, URL: https://github.com/TinkerTools/tinker.

[115] R. M. BERGMANN and J. L. VUJIC, *Algorithmic Choices in* WARP *– A Framework for Continuous Energy Monte Carlo Neutron Transport in General 3D Geometries on GPUs*, Annals of Nuclear Energy 77 (2015), pp. 176–193, DOI: 10.1016/j.anucene.2014.10.039.

[116] WARP *Source Code*, URL: https://github.com/weft/warp.

[117] ParticleStackBenchmark *Framework Source Code*, URL: https://gitlab.tuwien.ac.at/josef.weinbub/master-adel.

[118] B. STROUSTRUP, *The* C++ *Programming Language*, Addison-Wesley Professional, 2013, ISBN: 978-0-321-56384-2.

[119] CMake *Build Tool Website*, URL: https://cmake.org/.

[120] Slurm *Workload Manager Documentation*, URL: https://slurm.schedmd.com/documentation.html.

[121] Python *Software Foundation Website*, URL: https://www.python.org/.

[122] *Documentation for the* C++ *Containers Library*, URL: https://en.cppreference.com/w/cpp/container.

[123] GCC *– GNU Compiler Collection Website*, URL: https://gcc.gnu.org/.

[124] CLANG C++ *Compiler Website*, URL: https://clang.llvm.org/.

[125] INTEL *oneAPI* C++ *Compiler Website*, URL: http://software.intel.com/en-us/intel-compilers/.

[126] *Documentation for* std::declval, URL: https://en.cppreference.com/w/cpp/utility/declval.

[127] *Documentation for* std::bind, URL: https://en.cppreference.com/w/cpp/utility/functional/bind.

[128] *Documentation for* std::uniform_real_distribution, URL: https://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution.

[129] *Documentation for* std::mersenne_twister_engine, URL: https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine.

[130] VSC – *Vienna Scientific Cluster Website*, URL: https://vsc.ac.at/home/.

[131] VSC-5 *in the* TOP500 *list*, URL: https://www.top500.org/system/180056/.

[132] Alma Linux *Distribution Website*, URL: https://almalinux.org/.

[133] INTEL *MPI Library Website*, URL: https://software.intel.com/intel-mpi-library/.

[134] GCC *OpenMP Library Website*, URL: https://gcc.gnu.org/projects/gomp/.

[135] CUDA *Toolkit Website*, URL: https://developer.nvidia.com/cuda-toolkit.

[136] ROCm *Software Development Platform Website*, URL: https://rocmdocs.amd.com/en/latest/index.html.

[137] *Documentation for* std::any, URL: https://en.cppreference.com/w/cpp/utility/any.

[138] *Documentation for* std::variant, URL: https://en.cppreference.com/w/cpp/utility/variant.

[139] *Documentation for* std::optional, URL: https://en.cppreference.com/w/cpp/utility/optional.

[140] *Documentation for* Parameter pack *in* C$^{++}$, URL: https://en.cppreference.com/w/cpp/language/parameter_pack.

## Erklärung

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Ort, Datum

Unterschrift

Name