HLRS

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHT

CONCEPTS FOR SCALABLE MOLECULAR DYNAMICS SIMULATIONS ON FUTURE HPC SYSTEMS

Christoph Walter Martin Niethammer

ISSN 0941 - 4665 Juni 2023 HLRS-26

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

CONCEPTS FOR SCALABLE MOLECULAR DYNAMICS SIMULATIONS ON FUTURE HPC SYSTEMS

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung
vorgelegt von

Christoph Walter Martin Niethammer
aus Stuttgart

Hauptberichter:	Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter:	Prof. Dr.-Ing. habil. Jadran Vrabec
Tag der Einreichung:	7. Juni 2021
Tag der mündlichen Prüfung:	7. Juli 2022

D93

ISSN 0941 - 4665 Juni 2023 HLRS-26

Abbreviations and symbols

Abbreviations

ABFT	application-based fault tolerance
C/R	checkpoint/restart
CPU	central processing unit
DAG	directed acyclic graph
fcc	face centered cubic
HPC	high performance computing
I/O	input and output
LJ	Lennard-Jones
LJTS	Lennard-Jones Truncated Shifted
MD	Molecular Dynamics
MPI	Message Passing Interface
MTBF	mean time between failure
NRMSD	normalized root mean square deviation
NUMA	non-uniform memory access
OS	operating system
PGAS	Partitioned Global Address Space
RDF	radial distribution function
RDMA	remote direct memory access
ULFM	User Level Failure Mitigation
VLE	vapor-liquid equilibrium

Symbols

r_c	cut-off radius
ϵ	energy parameter of the Lennard-Jones potential
ρ	density
σ	size parameter of the Lennard-Jones potential
S	speedup

Abstract

Molecular Dynamics (MD) simulation is one of the fundamental tools for researchers to get insights into processes in Biology, Chemistry, Materials Science, Physics, and many other fields at a microscopic level. Since the beginning of MD simulations, they are one of the use cases for the largest high performance computing (HPC) systems in the world. Despite their underlying algorithms being well-suited for parallelization, there are many problems to overcome when running MD simulations at scale due to the size and complexity of today's HPC systems. These problems limit the size and amount of detail of the simulations that researchers can perform. So there is a constant need for improvement and adaptation of MD codes for current and future HPC systems.

The goal of this thesis is to address several of these problems to improve the capabilities of MD simulations on current and future HPC systems. Therefore, the work discusses concepts for scalable MD simulations in the context of these problems. The discussed concepts cover two problem areas: First, the performance and second, the fault tolerance of MD codes that are used to run these simulations. In both areas, state of the art approaches and solutions are evaluated and new models for a better understanding of algorithms and respective code implementations are introduced. Improvements over the current solutions are proposed and implemented in kernels or real-world MD codes to prove their applicability and quantify the possible performance increases. The following topics are discussed in detail: I/O, intra-node-level performance, inter-node-level performance, and fault tolerance.

The aspect of application I/O and setup of simulation at scale is identified to be a problem that MD code users struggle with due to their lack of specific HPC know-how. Hence, a new scenario generation framework is presented. The corresponding implementation in the ls1-MarDyn code is easy to use. Its transparent parallelization is shown to scale up to thousands of processes even for complex simulation scenarios.

With respect to intra-node-level performance several subproblems are addressed. First, the choice of an efficient algorithm is a common issue. While the complexity and implementation efficiency of the fundamental algorithms for MD have already been extensively discussed in the past with respect to a large number of particles, the increasing need for parallelism shifts the focus back to their use with fewer molecules. To increase the understanding for the algorithms' behaviour in the latter case, cost models are created for the naïve, linked-cell, and neighbour-list algorithms. On their basis, an algorithm selection diagram is derived that takes into account fundamental simulation scenario values as well as algorithm tuning parameters. Second, for the fundamental naïve algorithm a large set of implementation possibilities is provided and compared on cache and vector based systems. The best implementations are found to be multiple times faster than some other straight forward implementations. Based on the results, implementation advice for MD code developers is provided. Third, the problem of extracting sufficient parallelism to distribute work over the many compute cores of HPC systems is addressed by a look at task-based programming models. Here, the focus is laid on the latest ideas for shared memory implementations at the node level. A potential pitfall is identified in the context of the currently available programming models. Solutions to resolve the related scaling limits are proposed and a benchmark implementation demonstrates the successful application for actual codes. Feedback is provided to task runtime developers to improve their programming models.

With respect to inter-node-level performance, traditional distributed-memory parallelization using the domain decomposition technique and message passing is studied. The folding-based neighbour communication approach is found to work best at scale on current HPC systems. A large problem identified here for the future is the correct placement of processes for a good hardware to application topology mapping. Mitigation of this problem via overlapping of communication and computation is proposed. Different communication patterns to achieve overlapping are implemented in ls1-MarDyn and show up to 8 percent improvement for the chosen simulation scenario.

In the context of decreasing reliability of HPC systems, fault tolerance is discussed. Here, a concept for a new application-based fault tolerance (ABFT) approach for

MD simulations is presented. Its applicability for a standard simulation scenario and its performance advantage over the classical checkpoint/restart (C/R) approach is shown.

The results of this work obtained in all these topics together, help to improve MD codes to run larger simulations on current and future HPC systems.

Zusammenfassung

Die Molekulardynamik (MD) Simulation stellt eine der fundamentalen Hilfsmittel für Forscher dar, um Einblicke in Prozesse in der Biologie, Chemie, Materialwissenschaften, Physik und vielen anderen Feldern auf einem mikroskopischem Level zu gewinnen. Seit dem Beginn der MD Simulationen stellen diese einen der Anwendungsfälle für die größten Hochleistungsrechner (HPC) der Welt dar. Doch obwohl deren zugrundeliegenden Algorithmen sich gut für die Parallelisierung eignen, müssen aufgrund der Größe und Komplexität heutiger HPC Systeme eine Vielzahl an Problemen bei der Skalierung überwunden werden. Diese Probleme limitieren die Größe und den Detailgrad der Simulationen, die Forscher durchführen können. Daher besteht ein stetiger Bedarf an Verbesserung und Anpassung von MD Programmen für aktuelle und zukünftige HPC Systeme.

Das Ziel dieser Doktorarbeit ist es einige dieser Probleme zu beheben um die Fähigkeiten von MD Simulationen auf aktuellen und zukünftigen HPC Systemen zu verbessern. Hierzu diskutiert die Arbeit Konzepte skalierbarer MD Simulationen im Kontext dieser Probleme. Die diskutierten Konzepte umfassen zwei Gebiete: Erstens die Performance und zweitens die Fehlertoleranz von MD Programmen, die für die Simulationen eingesetzt werden. In beiden Gebieten werden Methoden und Lösungen nach aktuellem Stand der Technik evaluiert und neue Modelle für ein besseres Verständnis der Algorithmen und entsprechender Implementierungen in den Programmen eingeführt. Verbesserungen der aktuellen Lösungen werden vorgeschlagen und als Kernel oder in echten MD-Programmen implementiert um deren Anwendbarkeit zu zeigen und die möglichen Performance-Verbesserungen zu quantifizieren. Die folgenden Punkte werden im Detail behandelt: I/O, intra-knoten-level Performance, inter-knoten-level Performance und Fehlertoleranz.

Der Aspekt des I/O der Anwendung und des Aufsetzens der Simulation *at Scale* stellt sich als ein Problem heraus, mit dem MD Programm Benutzer Schwierigkeiten haben, da ihnen das spezifische HPC Wissen fehlt. Aus diesem Grund wird ein neues Szenario-Generierungs-Framework vorgestellt. Die Implementierung im ls1-MarDyn Programm ist einfach zu benutzen. Die Skalierbarkeit ihrer transparente Parallelisierung bis hin zu tausenden von Prozessen wird demonstriert und selbst für ein komplexes Simulations-Szenario erreicht.

Im Hinblick auf die Performance auf dem Intra-Knoten-Level werden mehrere Probleme adressiert. Zuerst einmal ist die Wahl eines effizienten Algorithmus ein wichtiger Punkt. Während die Komplexität und Effizienz von Implementierungen für die fundamentalen Algorithmen der MD in der Vergangenheit im Hinblick auf große Molekülzahlen bereits umfassend untersucht wurden, verschiebt der zunehmende Bedarf an Parallelität den Fokus zurück zu deren Nutzung mit weniger Molekülen. Um das Verständnis der Algorithmen für letzteren Fall zu verbessern, werden Kostenmodelle für den naiven, linked-cell und Nachbarschaftslisten Algorithmus erstellt. Auf deren Grundlage wird ein Auswahldiagramm für Algorithmen abgeleitet, das Werte aus dem simulierten Szenario als auch Tuningparameter der Algorithmen berücksichtigt. Anschließend werden für den naiven Algorithmus eine große Zahl verschiedener möglicher Implementierungen aufgezeigt und auf Cache- und Vektor-Systemen verglichen. Die besten Implementierungen erweisen sich hierbei um ein Mehrfaches schneller als einige der anderen intuitiven Implementierungen. Basierend auf den Ergebnissen werden Hinweise für MD Programm Entwickler gegeben. Weiter wird das Problem des Extrahierens ausreichender Parallelität zur Verteilung der Arbeit über die vielen Rechenkerne eines HPC Systems adressiert, wobei ein Blick auf die task-basierten Programmiermodelle geworfen wird. Der Fokus wird dabei auf die neuesten Ideen zur Shared Memory Implementierung auf Knotenebene gelegt. Hier wird eine potentielle Schwierigkeit im Kontext der aktuell verfügbaren Programiermodelle identifiziert. Vorschläge zum Lösen der damit verbundenen Limitierung der Skalierung werden gemacht und eine Benchmarkimplementierung demonstiert die erfolgreiche Anwendbarkeit für echte Codes. Feedback für die Task-Laufzeit-Entwickler wird gegeben, so dass diese ihre Programiermodelle verbessern können.

Im Hinblick auf die Performance auf dem Inter-Knoten-Level wird die traditionelle Distributed Memory Parallelisierung auf Basis von Domain Decomposition Techniken und Message Passing untersucht. Der Ansatz auf Basis der faltenden Nachbarschaftskommunikation erweist sich hierbei als der am besten Skalierende auf aktuellen HPC Systemen. Als ein in Zukunft größeres Problem stellt sich hierbei die richtige Verteilung der Prozesse für ein gutes Hardware- zu Anwendungs-Topologie Mapping heraus. Eine Verringerung der gefundenen negativen Effekte mittels Überlappung von Kommunikation mit Berechnungen wird vorgeschlagen. Verschiedene Kommunikations-Muster zur Überlappung wurden in ls1-MarDyn implementiert und zeigen eine Verbesserung von bis zu 8 Prozent für das ausgewählte Simulationsszenario.

Im Kontext sich verringernder Ausfallsicherheit von HPC Systemen wird Fehlertoleranz diskutiert. Hier wird eine neue Methode zur anwendungsbasierten Fehlertoleranz bei MD Simulationen vorgestellt. Ihre Anwendbarkeit für ein gewöhnliches Simulations-Szenario sowie ihre Performance-Vorteile gegenüber dem klassischen Checkpoint/Restart Ansatz wird gezeigt.

Die in dieser Arbeit gewonnenen Ergebnisse zu all diesen Punkten helfen MD Programme zu verbessern und damit größere Simulationen auf aktuellen und zukünftigen HPC Systemen durchzuführen.

Acknowledgements

First of all, I would like to shout out a big “Thank you!” to my parents for their long-lasting support over the time. Without your support, this work would not have been possible.

Many thanks go also to all my friends for the interesting and entertaining discussions and activities together in the past years. You helped me a lot in maintaining a healthy work-life balance.

I thank Prof. Michael Resch for giving me the opportunity and resources to perform my research at HLRS. With his effort, HLRS became one of the leading HPC centers worldwide with employees from all over the world providing a great working environment. I thank Prof. Jadran Vrabec for the collaboration and for taking over the responsibility of the second report for my work.

I would like to express my gratitude to Prof. Rainer Keller for getting me interested in high performance computing, giving me the chance to join this exciting field in his group at HLRS as a student, and being my mentor for a long time.

I thank Dr. José Gracia for his support and the possibility to continue my work in his group in the past years. I appreciate his helpful comments and questions, keeping me on the right track.

I thank Dr. Rolf Rabenseifner and Dr. Joseph Schuchart for the many technical and non-technical discussions working together on so many topics in MPI, OpenMP, and the MPI-Forum. I also would like to thank Dr. Martin Bernreuther for always sharing his deep knowledge about informatics and HPC architecture with me.

Further, I would like to thank Prof. Jesus Labarta, Prof. Hans Hasse, Prof. Hans-Joachim Bungartz, Prof. Martin Horsch, Dr. Bernd Mohr, and Dr. Christian Terboven as well as their teams for all the inspiring discussions and great work in the many projects together.

My acknowledgement goes to all the members from HLRS as well as NEC and Cray that provided me with technical support and many inspiring dialogues. It is great to be a part of such a diverse and knowledgeable team.

My special thanks goes to my brother Matthias: Thanks for spending hours of "whiteboard discussions" together — and most important keeping my mood up in hard times!

Contents

List of Figures

List of Tables

1 Introduction

Adler and Wainwright introduced the method of Molecular Dynamics (MD) in 1959 [7]. Since then, MD has become one of the major tools to study both thermodynamic properties as well as time dependent phenomena at the nanoscopic scale. Such simulations are used across many scientific fields to deepen the understanding of phenomena in biology [55], physics [45], and chemistry [93] as well as for the optimization of materials and processes in engineering [50, 86]. They provide access to data that would otherwise be difficult to access experimentally, such as critical point data [96], mobilities at liquid-solid interfaces [64], crack formation processes [91], and further help to reduce the number of experiments with toxic or explosive substances.

There is a constant necessity in the MD community to simulate large systems. Large system simulations today include up to billions of particles and span millions of time steps. One example is the study of time dependent fluid phenomena with nanostructures at the atomic scale, resulting in a very large number of atoms to be simulated for a long time. Another example is the determination of physical properties near the critical point where the convergence rate of simulations is very low and so extremely long runs are required to achieve a meaningful result. Thus, the evolution of MD simulations, the development of the fastest computing systems of their time, and the field of high performance computing (HPC) are interwoven. This manifests visibly in the history of the Gordon Bell prize, honoring several advances in MD simulations [14] with the last one in 2020 [53].

While MD codes like ls1-MarDyn [70], IMD [85], LAMMPS [78], and GROMACS [16] have already shown their applicability for large computer systems running on thousands of nodes, they have to face new challenges when it comes to their use on even larger scale machines in the future. According to the TOP500 list [3,65], the top four super computer systems in the world by 2020 provide each more than one million cores and include accelerator cards in addition to conventional central processing units (CPUs). In order to use these systems to their full potential in a simulation, it is necessary to divide the problem into sufficiently small subproblems and distribute them across the available compute resources such that there is as little need for communication between as many compute cores as possible. Further, the small subproblems have to be executed as efficiently as possible on the various hardware components. This makes it important to tune the algorithms for all the problems and their implementations for the target architecture in particular.

One problem, becoming more of an issue in the field of HPC systems, is application input and output (I/O). Processor performance was following Moore's law [67] for a long time, increasing the computational speed exponentially. Storage capacity also increased at a similar rate while following Kryder's law [97]. However, the speed of the I/O system did not see the same type of exponential growth, ultimately leading to a gap that makes it difficult to read and write huge amounts of simulation data to and from the available storage. As the I/O of HPC systems is also commonly a shared resource between all running applications, this becomes a tremendous bottleneck and causes frequent problems in HPC system operation.

Another challenge becoming more common with the advent of an increasingly larger number of components in HPC systems, is reliability [24]. Hosting a larger, more complex system requires larger efforts to protect simulations from failures that can lead to data loss or corrupted results of computations. Hardware solutions to keep the mean time between failure (MTBF) of the whole system at an acceptable level require a lot of additional hardware resources and energy. Software and algorithm based solutions beyond the classical checkpoint/restart (C/R) approach are therefore becoming increasingly important [29].

1.1 Contribution of this work

MD code developers have to react to the challenges described above to allow their codes to scale on future HPC systems. It is therefore necessary to re-evaluate existing solutions and come up with new concepts for these codes. This work contributes to these efforts by evaluating selected aspects in different areas, contriving solutions to improve existing codes and providing new methods in areas where no suitable solutions exist so far.

The first aspect covered in this work is the setup of an initial simulation state during program start. Due to the increasing size of the simulations, the tools and the I/O system used to generate and read initial molecular configurations become a bottleneck for large MD simulations. Therefore, in a first attempt, the existing infrastructure is analyzed and optimized for a real MD code. From this gained experience, a new flexible in-memory scenario-generation system is developed that offers transparent parallelization to the user. Its improved performance and usability is shown on the basis of an implementation in the ls1-MarDyn MD code and real world simulation scenarios.

The second aspect to be addressed by this work is the performance of the algorithms for force calculations in MD simulations. To distribute the computational work across HPC systems at increasingly high scale, the simulated problems are divided into increasingly more subproblems, which consequently become smaller and smaller. Up to now, only asymptotic performance complexity models for an infinite number of molecules are used to compare and select the algorithm to be used. These models also do not cover details of the simulated scenario. Therefore, new analytical cost models are developed in this work that describe the performance of common MD algorithms for a small number of molecules. They take into account properties of the simulation volume of interest as well as algorithm specific tuning parameters. The models are tested on the basis of synthetic benchmarks and kernels. Algorithm selection criteria are derived from the obtained results that can help MD code implementers and users.

Another important aspect for the performance of the force calculation is the implementation of the algorithms. Here the naïve algorithm is analysed in detail in this work, as it is the foundation of all other commonly used algorithms. The behaviour of a large set of different implementations for it is compared using kernels in combination with different compilers across various hardware platforms. Based on the results, guidelines for the selection of an algorithm implementation are given.

One problem coming up with the even higher core counts of HPC systems is the extraction of sufficient parallelism from the application to utilize all resources. While hybrid approaches using the Message Passing Interface (MPI) for inter-node communication and OpenMP for intra-node parallelism help reduce overheads, they still leave this challenge to the programmer. One possible solution here is to use task-based approaches that automatically extract parallelism from provided dependencies [43, 75]. To that end, a task-based approach for intra-node-level parallelism is studied based on the OMPSs task runtime. Some pitfalls of this model are brought up and a solution to overcome them is presented.

As the number of nodes in HPC systems increases, the inter-node communication is becoming more important. One aspect addressed in this work evaluates the underlying neighbour communication used in domain-decomposition-based MD simulations. Two communication patterns are tested with respect to their scalability. Further, the important aspect of communication-computation overlapping is addressed by the implementation and analysis of different communication-computation strategies in ls1-MarDyn for the widely used folding based communication pattern.

The last part of this work addresses the increasingly important aspect of fault tolerance. A new application-based fault tolerance (ABFT) approach is introduced for MD simulations. Its applicability is studied on the basis of a common application scenario and its potential performance benefits over the classical C/R approach are shown.

1.2 Outline

The work is structured into six chapters as follows: Chapter 2 introduces the general background for this work. It includes the necessary basics and current state of the art, which is used in the following chapters. This includes an introduction into supercomputers and their architectures, the concepts of MD simulations and the algorithms used for simulations, as well as important aspects of fault tolerance.

The following four chapters include the contribution of this work related to different fields. Chapter 3 focuses on the general problem of the initial configuration creation at scale. Therefore, an analysis of the current practice for the ls1-MarDyn code is made and performance as well as usability issues are identified. From there, a generalized scalable initialisation approach is designed. The approach is then implemented in ls1-MarDyn and evaluated against the original implementation. Chapter 4 deals with the optimization of MD codes at the node level. Here, algorithm choice and implementation are studied in detail. Analytical performance models for common algorithms are developed and tested with different benchmarks. This is followed by an evaluation of different ways to implement the basic and most important force computation kernel including a behavioural study on different hardware platforms with different compilers. The question about efficient parallelization at the node level is then examined in the context of a task-based parallelization approach. Chapter 5 shifts the focus towards scalability to high numbers of compute nodes and efficient inter-node communication. The two most common communication schemes are compared and their communication-computation overlapping potential is exploited for the ls1-MarDyn code. Finally, Chapter 6 deals with the problem of fault tolerance by presenting a new ABFT approach for simulation recovery. Its applicability is studied on the basis of a common MD simulation scenario.

Chapter 7 concludes this work. It summarizes the results and gives some outlook on future research directions.

2 Background and related work

This chapter introduces the necessary background and state of the art for the following parts of the work. The HPC system architecture is shown and its different components are explained as they are currently found in leading HPC systems. A short introduction into parallel programming and the parallel programming models, which are used in this work follows. The basics of Molecular Dynamics (MD) simulations are then presented, including the most important algorithms and parallelization strategies. Here, the MD code ls1-MarDyn is described shortly, which will be used for many of the evaluations. At last, the area of fault tolerance is outlined, showing the currently most important approaches as well as the existing basic theory model to describe application overheads.

2.1 Supercomputer architecture and hardware

This section gives a short introduction into the architecture of current supercomputers and the hardware components they are built of. At the end some examples for HPC systems are provided, including systems used for performance evaluations in this work.

2.1.1 Overall architecture

The TOP 500 list ranks supercomputers based on their performance in the LINPACK benchmark [30]. Looking at the leading systems in the list's latest release for 2020, today's supercomputer architecture is dominated by multi-node systems connected via fast interconnects. For most systems, the nodes are of the same type—resulting in a homogeneous system. The single nodes are in most cases diskless and the system's storage is attached via network and I/O nodes to the rest of the system. To achieve the necessary high I/O bandwidth to store the huge amount of data from the simulations, a parallel file system is typically used on the storage system, such as Lustre [21, 26]. Figure 2.1 depicts the architecture of such a typical system.

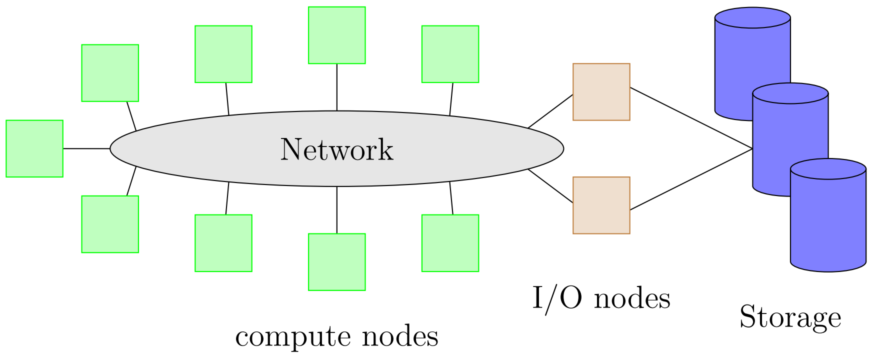


Figure 2.1: Common architecture of an HPC system including compute nodes, I/O nodes, and storage

Normally, users do not access the compute nodes in the system directly. Instead, they prepare and start their computations from a login node using scripts, which they submit to a job scheduler. This scheduler then starts the parallel application on the compute nodes, managing all the different user requests across the available resources.

2.1.2 Compute nodes

A compute node is the basic unit of a supercomputer. It consists out of three main components: The central processing unit (CPU), memory, and the network interface. Most nodes of today's HPC systems follow a non-uniform memory access (NUMA) node architecture and come with two CPUs as shown in Figure 2.2. Therein, each CPU connects to memory via a memory controller, which is in most cases integrated into the CPU. The two CPUs are connected via some interconnect to each other and allow access to the memory that is attached to the other CPU. However, this memory access is slower than the access to the CPU's local memory. Each node includes one network interface to connect the node to the other nodes in the system. This network interface is often attached via a bus to one of the CPUs, and allows remote direct memory access (RDMA) operation.

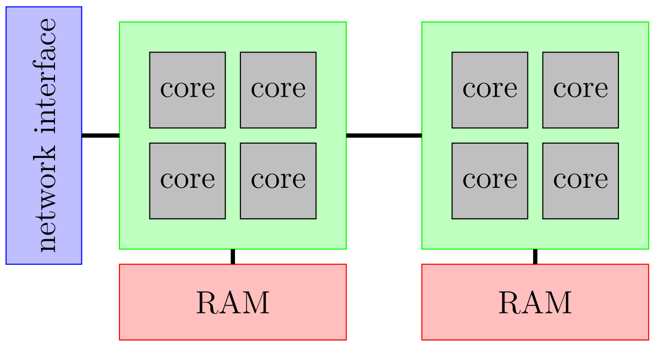


Figure 2.2: Common architecture of a compute node

2.1.3 Network

Different network technologies are found in today's leading supercomputers. Typical technologies are InfiniBand [3, 83] and custom interconnects like, e.g., Cray's Gemini and Aries chips. InfiniBand based systems use in most cases a fat tree topology [59], while the custom interconnects come with more elaborate topologies based on 3D or 4D grids, hypercubes or the dragonfly topology [56]. The networks support message and RDMA based communications for high bandwidths and low latencies with Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) parallel programming models.

2.1.4 HPC platforms used in this work

Laki/Vulcan HLRS Laki/Vulcan, is a Linux cluster based mainly on dual socket nodes with Intel processors. It consists of roughly 600 nodes, which are connected by a two level InfiniBand fat tree with a blocking factor of 5 : 1. The InfiniBand network uses a variety of Mellanox cards and switches operating at DDR and FDR speeds. The nodes of the system are heterogeneous and vary in the CPU model as well as amount and type of memory.

HLRS Cray XT5/XC systems Over the time of this work a variety of Cray XT and XC systems was used: A **Cray XT5m** with 112 dual socket Quad Core AMD Opteron 6276 (Interlagos) processor nodes connected via the Cray SeaStar interconnect [22] and 16 GB of RAM each. **Hermit**, a Cray XE6, consisting out of 3552 dual socket nodes with AMD Opeteron 6276 (Interlagos) processors and 32/64GB RAM per node, connected via the Cray Gemini interconnect [9]. **Hornet/Hazel Hen**, a Cray XC30/XC40, consisting out of 7712 dual socket Intel Xeon Haswell E5-2680v3 nodes and 128 GB RAM per node, all connected via the Cray Aries interconnect [37].

NEC SX-ACE The NEC SX-ACE is a vector processor based supercomputer architecture [2]. The NEC SX-ACE system at HLRS consists out of 256 nodes. Each node is equipped with one processor with 4 vector cores running at 1 GHz and 64 GB of main memory which is assisted by 1 MB of vector cache (ADB) in each CPU core. The special feature of this system is its very high memory bandwidth of 256 GB s^{-1} that can be used by a single core or shared by all four cores resulting in a very high overall byte to flop ratio of 1 B/FLOP. The vector length is 256 elements. The nodes are connected via the IXS interconnect.

2.2 Parallel programming and parallel programming models

Parallelization is the concept of distributing work on several resources, where resources can be manifold. With respect to the computer systems in this work, these resources range from resources inside a single CPU core in the form of multiple instruction ports, over multiple CPU cores in a processor up to multiple compute nodes in a larger computer system. For high performance computing (HPC) systems, it can be assumed that a system consists mainly out of homogeneous resources, located at one place and that these resources are well connected via a fast interconnect.

To make use of these resources a variety of programming models came up. In general, it can be distinguished between two programming model classes: parallel processes with message driven communication and models that communicate over a shared memory. The first model is mostly used to parallelize an application across a network, the second model is commonly applied within a single node.

2.2.1 Scalability

Different metrics exist to measure the resource usage efficiency of a parallel application. One of the most common metrics is the so-called speedup. The speedup $S(n)$ of a parallel program using n processing elements is defined as the fraction between the execution time of the serial application t_{serial} and the parallel program's execution time $t(n)$. In this context often the serial execution time is assumed to be the same as the execution time of the parallel application using only one processing element $t(1)$:

$$S(n) = \frac{t_{\text{serial}}}{t(n)} \approx \frac{t(1)}{t(n)} \tag{2.1}$$

If a fraction f of the program can be parallelized and there are no additional overheads due to the parallelization, the speedup can be expressed by

$$S_{\max}(n) \leq \frac{1}{(1-f) + \frac{f}{n}} \; . \tag{2.2}$$

This formula describes the performance model, which is known as Amdahl's law [10]. So, the ideal speedup for an application that is parallelized to 100 % is n. However, a so called superscalar speedup may be observed with increasing n, which can sometimes result in higher values than n.

2.2.2 Message passing

Message Passing Interface MPI is the leading standard when it comes to highly parallel HPC applications since it was introduced in 1994 [25,41]. In the MPI world, the MPI process is the basic operational unit. MPI processes are distinguished by their individual rank number in a process group and can communicate with each other using the MPI API. MPI is mostly message driven and therefore the basic API functions are send and receive operations. But there are also optimized collective communication patterns or a parallel I/O interface as well as support for one-sided communication with get and put operations. Important aspects in running MPI applications efficiently are overlapping of communication and computation as well as the reduction of slow network communication by optimized data distribution and process placement based on the hardware topology. The current MPI-3.1 standard does not provide failure handling beyond a simple fail stop model for now. An extension for fine grained failure handling in MPI is discussed for the future in the User Level Failure Mitigation (ULFM) proposal [4, 17].

2.2.3 Multi threading

User level threads are execution streams that have some private state. Multiple threads can run within a single process as depicted in Figure 2.3. The threads share and can access resources of the parent process, e.g., code, data, and open files. They share a common memory address space, which allows them to communicate not only via messages but also via shared data in the memory. So, communication between threads mostly requires synchronisation and no data transfer. Therefore, thread based parallelization allows very good performance in tightly coupled parallel computations. A common implementation are POSIX threads [5], which often serve as a portable basis for other higher level parallel programming models.

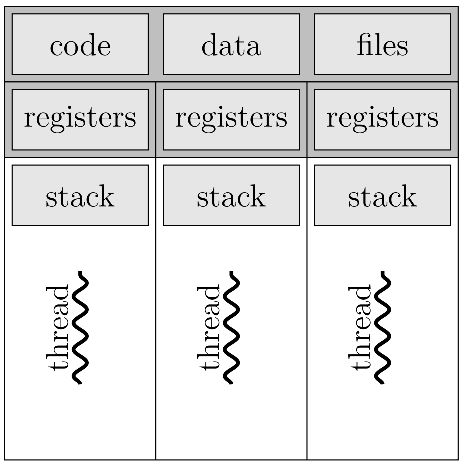


Figure 2.3: Multi threaded process

OpenMP OpenMP is an application programming interface standard for writing multi-threaded applications in C, C++, and Fortran, which was first introduced in

1997 by the OpenMP Architecture Review Board [19]. At its beginning OpenMP was mostly loop parallelization driven, however, today it includes advanced features like tasks or nesting as well as support for accelerator offloading and optimization hints for compilers [20].

Task dependence based parallelization models A relatively new approach for parallelization using threads is the application of a data flow execution model with user level software tasks. The idea here is not to parallelize an application by directly specifying the parallelism, but by simple code annotated tasks, which are executed in parallel by a runtime system with respect to their dependencies. A task in this model may be a function or a code region. Several approaches to specify task dependencies exist and were implemented for different hardware architectures: StarSs [76], OmpSs [31], StarPU [11]. Basic support of the model for shared memory systems was also introduced in the OpenMP standard with the 4.0 version [74]. Most of the approaches specify dependencies by marking input and output parameters for all tasks. During program execution, this information is used to create task dependencies, which are then translated into a directed acyclic graph (DAG) from which the task execution order is derived, i.e., the programming model runtime executes the tasks for which dependencies in the DAG are fulfilled.

In case of StarSs and OmpSs, the dependencies are handled by the memory addresses of the parameters in the case of C/C++. This has some advantages and disadvantages: On the one side using memory addresses is a very natural way which considers automatically different variables belonging to the same memory location - which is important especially in C/C++. On the other side using a single memory address may not be very useful when it comes to parameters which represent memory areas, i.e., arrays or data-structures. Here different strategies to circumvent this shortcoming can be applied. One of them is the so-called sentinel technique, which uses a sentinel variable to represent the elements of an array or data-structure [61]. However, this has the disadvantage of additional code and increased program complexity[1].

To achieve an efficient parallelization with this model, two approaches can be used: (1) starting with large tasks that take a long time to compute, and refine them until enough parallelism in the task DAG is reached to get sufficiently good scaling [88] or

[1]The new OmpSs2 implementation added an array notation for dependencies.

(2) start at the instruction level using the DAG within the compiler representation and combine instructions until sufficiently large tasks are created, so scaling is not limited by the overhead of the runtime system [60].

The big advantage of this model comes from the automatic extraction of parallelism from a program, allowing also programmers with little to no knowledge of parallel programming to parallelize their codes.

2.3 Molecular dynamics simulations

Classical MD simulations are based on classical mechanics where the movement of atoms with mass m_i and spatial position $\vec{r_i}$ are described by a set of coupled Newton's equations of motion

$$m_i\ddot{\vec{r_i}} = F(\vec{r_i}) = \nabla U(\vec{r}) \ . \tag{2.3}$$

Here, $F(\vec{r_i})$ are the forces acting on molecule i and $U(\vec{r})$ is the interaction potential between all atoms. The basic problem of MD simulations is now to solve this coupled system of differential equations for a system of N atoms with coordinates $\vec{r}_1, \vec{r}_2, \ldots, \vec{r}_N$ and a given potential $U(\vec{r})$ in an efficient way. The atoms within the simulated systems can occur either as single atoms or as molecules, which can be used to reduce the number of equations, e.g., by introduction of positional constraints for atoms within a molecule.

2.3.1 Molecular interactions

The interatomic potential function $U(\vec{r})$ describes the potential energy of an atom at position $\vec{r}$ within the interaction field caused by all atoms in the system. It can be represented by a series expansion in the following form:

$$U(r) = \sum_i u_1(r_i) + \sum_{i,j>i} u_2(r_i, r_j) + \sum_{i,j>i,k>j} u_3(r_i, r_j, r_k) + \ldots \ . \tag{2.4}$$

The first three addends therein correspond to the contributing parts single atom potential, pair interaction potential, and three body interaction potential. The interaction potentials between molecules are the key component of MD simulations.

Classical MD simulations of fluids take into account the pair interaction and neglect the three body interactions in most cases. This is possible as bonds are here not as important as for the simulation of solids or large bio molecules.

A wide variety of pair-interaction potentials exists. Well known are the simple Coulomb potential and the more sophisticated Buckingham or Morse potentials. Beside this, Neural network based interaction models are now introduced [103]. However, one of the most important pair interaction potentials is still the simple Lennard-Jones (LJ) potential. It is briefly described in the following and will be used within the simulations performed in this work.

LJ potential To approximate Van-der-Waals interactions between non-bonded neutral atoms, the LJ potential is often used. [8] It is derived from the exact attractive London formula for dipole-dipole interactions and an approximate dispersive force, which is often modelled as a power law resulting in the form

$$\Phi_{\mathrm{LJ},n}(r) = 4\epsilon \left[\left(\frac{\sigma_n}{r} \right)^n - \left(\frac{\sigma}{r} \right)^6 \right] . \tag{2.5}$$

Here ϵ describes the potential depth and σ the interaction range. Due to efficient computation often the LJ-(12-6) potential is used:

$$\Phi_{\mathrm{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] . \tag{2.6}$$

Within this work LJ will be used as a synonym for the LJ-(12-6) potential.

Short range potentials

Often short-range potentials are used in MD simulations. They are characterized, by the property, that the main contribution of the potential energy of a given particle arises from its neighbours that are closer than a given cutoff distance r_c [42]. This allows the introduction of a cut-off radius for the potential within which the function is evaluated for interactions. For the contributions of interactions with distances larger than the cut-off, a correction function can be computed. This is not exact, but will give in most cases a sufficiently good approximation [84]. The LJ potential falls into the class of short-range potentials.

When the LJ potential is used with a cut-off radius, effects of the discontinuity at $r = r_\mathrm{c}$ can cause problems in the physical behaviour. These problems can be eliminated by an additional offset for most cases. Fixing the discontinuity in the potential results in the Lennard-Jones Truncated Shifted (LJTS) potential [92]

$$\Phi_\mathrm{LJTS}(r) = \begin{cases} \Phi_\mathrm{LJ}(r) - \Phi_\mathrm{LJ}(r_\mathrm{c}) & r < r_\mathrm{c} \\ 0 & r \geq r_\mathrm{c} \end{cases} . \tag{2.7}$$

2.3.2 Boundary conditions

Simulations performed on a computer are limited to a finite data set that fits into memory and can be computed in a decent runtime. This fact restricts the number of atoms and molecules, which can be simulated, and makes it necessary to specify boundary conditions. There are different types of boundary conditions. The most common once are:

- **reflecting boundary:** Particles are reflected at the simulation volume boundaries. The reflection mechanics can be implemented in different ways, e.g., using velocity flipping or mirror particles [8].
- **inflow and outflow:** Here particles are inserted at specific points of the simulation volume while particles crossing simulation volume boundaries are removed from it.
- **periodic boundary:** This is a variant of the inflow and outflow condition, where particles leaving on one side are inserted at the other again.

In the case of periodic boundary conditions, the so called minimum image convention is commonly applied, which takes into account only the nearest copy of an atom for interactions [8]. This condition becomes important for small systems, when the interaction range is larger than half the system diameter. The minimum distance between two atoms i and j is then given by:

$$r_{ij} = |\vec{r_{ij}}| = \min_n |\vec{r_i} - \vec{r_j} + \boldsymbol{L}\vec{n}| \ , \tag{2.8}$$

where $\vec{n} \in \mathcal{N}^3$ and $\boldsymbol{L}$ is a matrix with the dimensions of the simulation volume as diagonal entries.

2.3.3 Algorithms in molecular dynamics

MD simulations solve Newton's equation of motion for many particles by numerical calculations on computer systems. The main task solving this system of coupled differential equations is the computation of distances and forces between the molecules or atoms based on the interaction potentials introduced in Section 2.3.1.

Since the beginning of MD simulations in 1961 many different algorithms have been developed for this purpose, each of them targeting the efficient handling of a specific problem. Most of these algorithms are designed for two body interactions and short range potentials for which they reduce the computational costs effectively [45]. The most common of these algorithms that are used in this work are presented in the following sections.

For all algorithms the computational costs may be halved by the help of Newton's "actio est reactio" principle [28]. This allows to compute forces only once for a pair of particles and updating both at the same time. However, this results in write after write conflicts during parallelization. Commonly used techniques to handle those conflicts are synchronization or colouring schemes [36]. The following presentation of the basic algorithms omits the use of "actio est reactio". If "actio est reactio" is used in a later part of this work it will be stated explicitly.

Naïve algorithm

The naïve algorithm just computes all molecule pair distances and interaction forces [42]. While this algorithm is very simple to understand and can be executed efficiently on most hardware (see Section 4.2), the computational complexity of this algorithm is $\mathcal{O}(N^2)$ for a system with N molecules. Not scaling linearly with the system size makes it inappropriate for higher numbers of molecules.

```
for molecule1 $\in$ molecules do
    for molecule2 $\in$ molecules with molecule2 $\neq$ molecule1 do
        compute(molecule1, molecule2)
    end
end
```

Algorithm 1: Naïve algorithm

Naïve algorithm with cut-off

The naïve algorithm can be sped up for short range potentials by skipping the force calculation for molecule pairs that are further apart than a specified cut-off distance. However, the computational costs for the distance calculations still remain at a complexity of $\mathcal{O}(N^2)$.

```
for molecule1 ∈ molecules do
    for molecule2 ∈ molecules with molecule2 ≠ molecule1 do
        dr = distance(molecule1, molecule2)
        if dr ≤ rc then
            compute(molecule1, molecule2)
        end
    end
end
```

Algorithm 2: Naïve algorithm only computing short range potential parts.

Verlet lists

One of the oldest methods to speed up the calculation of multi-particle interactions for short range potentials is the Verlet neighbour list [94]. The idea of Verlet lists is a slowly changing environment implying that neighbour relationships between molecules do not change rapidly. This is especially the case for solid state problems, where atoms or molecules in a crystal do not travel large distances, but rather stay localized, or biomolecules, where bonds establish a stable molecular backbone.

The algorithm stores for each molecule a list of its neighbours, which is assumed to be valid for n time steps. The neighbours in the lists are selected to be within a certain distance $r_\mathrm{c} + r_\mathrm{s}$ to the corresponding molecule, where r_s is referred to as skin. This reduces the number of molecules to be checked for interactions in the n time steps from all molecules to molecules in the neighbour lists of each molecule. The neighbour lists have to be re-created after n time steps. Figure 2.4 shows the application of the Verlet list in 2D.

Taking the maximal relative velocity v_max between any two particles and the cut-off radius r_c of the interaction potential, the neighbour list has to be updated after a time of $t = r_\mathrm{s}/v_\mathrm{max}$ to guarantee that all neighbours are considered. Therefore, when simulating a system with root-mean-square molecule velocity $\bar{v}$, time step length h, and n time steps between list updates, the skin radius should hold the condition [94]

$$r_\mathrm{s} \gtrsim n\bar{v}h \; . \tag{2.9}$$

A typical set of parameters for LJ-fluid simulations are $r_\mathrm{s} = 0.3$, $n = 10$, and $h = 0.005$ where all values are in reduced LJ-units [81].

A drawback of the Verlet list algorithm is its memory consumption, which scales as the number of molecules times the average number of interaction partners. This can become a problem for large simulations. The time consuming list creation itself can be sped up using the linked-cell algorithm presented in the following.

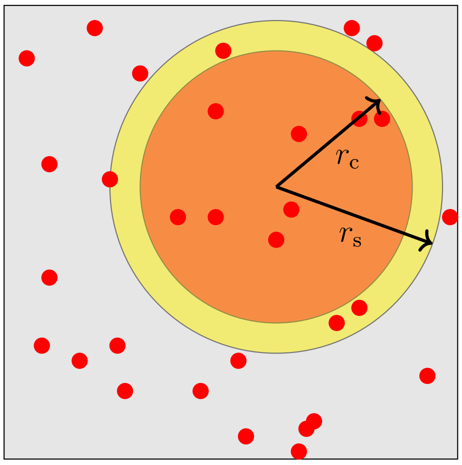


Figure 2.4: Verlet list algorithm in 2D: For each molecule, molecules within a distance r_s to it (yellow and orange area) are stored in a list. Neighbouring molecules within r_c can then be found in this list. The list has to be updated as soon as molecules from outside pass the skin (yellow area) and enter the inner area (orange).

The linked-cell algorithm

The standard algorithm used to reduce the number of pair interactions for short range potentials within a spatial volume is the linked-cell algorithm [78]. It decreases the numerical complexity of the distance calculation from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$.

The idea behind the linked-cell algorithm is to divide the volume into smaller subvolumes — referred to as cells — for faster access to spatial neighbours. In the original algorithm the cell dimension is chosen to be at least the cut-off radius of the interaction potential. This guarantees that interaction partners of molecules in one cell can be found in the directly neighbouring cells. Figure 2.5 shows the application of the linked-cell algorithm in 2D, where neighbours are found in the surrounding eight cells. In the 3D case, 26 neighbouring cells have to be checked.

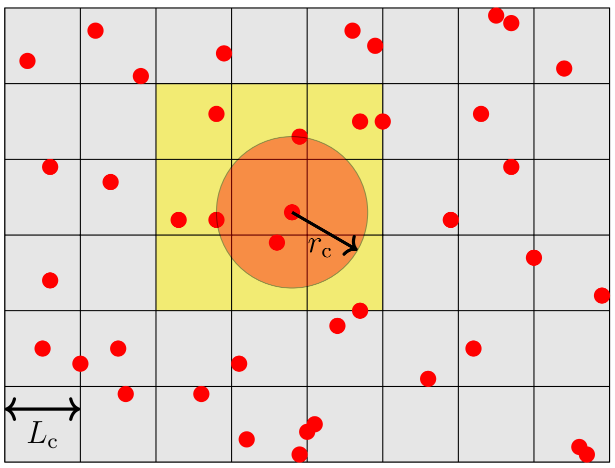


Figure 2.5: Linked-Cell algorithm in 2D: Molecules (red dots) are sorted into cells with a width $L_c \geq r_c$. The interaction neighbours of a molecule within r_c (orange circle) can then be found within the cell containing the molecule itself and its 8 neighbouring cells (yellow shaded area).

The algorithm can be extended to allow arbitrary dimensions for the cells, which can reduce the computational overhead for some applications [63]. Beside this, several other improvements exist for the algorithm [100]. For example, by sorting particles onto grid points [62] or by an approach sorting molecules inside cells along the projection of the cell-connecting vectors [44]. In this work the original algorithm will be studied as all results can be applied to the different variations and also most MD codes make use of it—including ls1-MarDyn (see Section 2.4).

2.3.4 Parallelization approaches

Running MD simulations on modern supercomputers requires work distribution across many processing units, as HPC systems provide nowadays up to several million

CPU cores [3]. Different work sharing concepts for MD simulations were developed over time. In the following, the standard approaches atom decomposition, force decomposition, and domain decomposition described by Plimpton [78] are outlined.

Atom decomposition

The simplest approach to parallelize MD simulations is the data parallelism based atom decomposition. The atoms are distributed across the available compute resources. Each compute resource computes only the corresponding forces acting on the atoms it owns. However, the computation requires to know the location of almost all molecules by each compute resource. This requires a global state update after each time step turning the approach impracticable for larger number of molecules to scale on modern HPC systems.

Force Decomposition

Looking at the naïve implementation for the computation of interaction forces in Algorithm 2, the force computations can be seen as a 2D matrix holding all the individual force contributions from all the interaction pairs. Each element F_{ij} in this table corresponds to the force between two molecules i and j. This table can be split up into blocks. The computation of these blocks is then distributed across the compute resources. As soon as each block is computed, a reduction over all blocks is performed to get the final forces on the molecules. The blocking allows for a reduction of the molecule position data needed by each process, improving the scalability of the algorithm in comparison to the atom decomposition. However, still expensive data exchange and reduction operations are required, turning it inefficient on modern HPC systems.

(Standard) Domain Decomposition

The most common approach to parallelize computations in many fields is the domain decomposition. Here the underlying simulation domain is divided into subdomains to build junks of work, which are then distributed across the compute resources.

The simplest approach for domain decomposition is to divide the simulation domain into cubic boxes. This approach is especially successful when computations require only local and not global data. For MD simulations, the domain decomposition can be seen as a dynamic atom decomposition where atoms are distributed based on their time dependent spatial distribution and the amount of molecule positions to be exchanged is reduced to a minimal amount.

2.4 The ls1-MarDyn Molecular Dynamics code

The ls1-MarDyn code is an MD code targeting large scale systems with millions of molecules that run for long time scales [33]. Typical scenarios computed with ls1-MarDyn are nano fluid simulations. These include for example condensation and evaporation processes to determine the corresponding condensation and evaporation rates [51] or flow simulations of a fluid alongside solid interfaces to study the viscous behaviour in given geometries [49].

These scenarios are demanding challenges for MD simulations and the ls1-MarDyn code: The simulation code must be very efficient in terms of computation to handle the large number of interactions and the long time scales. Further, the code has to handle the large number of molecules and must therefore also be efficient in terms of memory usage. To handle these challenges, ls1-MarDyn was already designed with a very efficient computational kernel and a scalable parallelization for large HPC systems in mind [70].

Around the computational kernel, ls1-MarDyn provides a flexible infrastructure to extend it with new functionalities. ls1-MarDyn uses here object-oriented programming techniques that simply the addition of, e.g., new communication patterns for particle exchange, or implement different particle types. In addition, it provides a flexible I/O plugin infrastructure that allows users to implement, e.g., new evaluation methods for their scientific research or additional output for subsequent data analysis or visualisation.

2.5 Fault tolerance

Fault tolerance describes the property of a system, which allows it to continue operation after a failure event in one of its components. For computer simulations, this means that an application can survive a hardware failure and continue the computation leading to the correct result at the end. Failures in HPC systems are most likely to show up as the loss of a node during runtime of a parallel computation. With the increasing number of components, this is becoming more of an issue and statistically one node in an HPC system may be affected once per hour by an error according to predictions [29, 48].

2.5.1 Software fault tolerance strategies

To achieve fault tolerance, different approaches can be taken either in hardware or in software and at different hardware or software levels [29]. As this work focuses on the application level, only the most common software based fault tolerance techniques shall be outlined here: Checkpoint/Restart, communication replay, and replication.

Replication Replication performs all operations multiple times using multiple instances of a code running on different hardware resources, e.g., nodes [38]. The state of the replicated instances is synchronized to keep them the same during normal operation and to detect failures. This approach is very expensive because of the need for at least twice the amount of hardware resources for the computations as well as the additional overhead in contiguous synchronisation between the different instances.

Checkpoint/Restart C/R is the most widely used fault tolerance approach in HPC applications today. It is based on the rollback recovery technique [35]. The state of a simulation is stored on a persistent storage system in a specified interval. When the system is subject to a failure, the simulation is restarted from the last successful checkpoint. The state of all resources is reset to the saved state and the simulation is continued from this point on. This requires the re-computation of all the simulation steps since the failure event. To overcome bandwidth limitations of normally slow persistent storage, it can be combined with a diskless checkpointing approach. Here checkpoint data are distributed over the memory of the entire application [77].

The checkpoint can be obtained at several levels. In most cases application level checkpointing results in the smallest amount of checkpoint data and is therefore preferred [12]. However, this approach requires additional effort by the users to implement checkpoint/restart capabilities into their program compared to a transparent operating system level approach. Alternatively, operating system (OS) level checkpointing may be used [46].

Log-based recovery Log-based recovery tries to overcome the problem of large checkpoints and restoration of the full simulation state by keeping a log of all messages sent over the network since a checkpoint [87]. This allows to continue the simulation only by restoring the failed process from the checkpoint and then replaying the communication for the failed resource until it reaches the same state as the still alive resources. While only the restoration of affected processes is necessary in this approach, the restoration time is likely to be the same as for checkpoint/restart (C/R) because the process restoration is not sped up itself.

2.5.2 Application-based fault tolerance

A relatively new approach for fault tolerance is the application-based fault tolerance (ABFT)[2] technique [95]. The idea behind this approach is to leverage the knowledge of the application itself to recover from a failure. This helps to reduce the amount of restoration relevant data and also the overhead in the recovery process. The classical C/R approach will recover the exact state of the application prior to failure. In contrast, the ABFT approach leaves the application the freedom to continue with a non-exact state and correct it at any point, so that it will achieve the desired correct result at the end, but with less overhead due to the failure handling. A special variant of this is the algorithm based fault tolerance method. Here, the failure can be corrected in an algorithmically exact way. For example, in some linear algebra calculations, it is possible to use a hot replacement technique: Here, data lost due to

[2]The abbreviation ABFT is also used for algorithm based fault tolerance, however, in this work ABFT is used as abbreviation for application based fault tolerance.

a failed process are replaced by checksum data and the result is corrected at the end based on the knowledge of the transformation between original data and checksum data [98].

2.5.3 Execution time of fault tolerant applications

The addition of fault tolerance features introduces overheads increasing program execution time. The total execution time t_{tot} of a program capable to recover after a failure using C/R was first studied systematically by Young [102]. This study was extended by Daly in [27] to a more detailed failure model and the following formula

$$t_{\text{tot}} = t_{\text{S}} + t_{\text{D}} + t_{\text{R}} + t_{\text{L}} \; . \tag{2.10}$$

Here, t_{S} is the time required to obtain the solution in a failure free environment, t_{D} is the dump time needed to write checkpoint data, t_{R} is the restore time to set up computation after a failure from the checkpoints, and t_{L} is the rework time necessary to recalculate lost intermediate results since the last checkpoint. Based on this model, an optimal checkpoint interval τ_{opt} can be derived. According to [27], a good first order approximation for larger mean time between failure (MTBF) values is

$$\tau_{\text{opt}} = \sqrt{2\delta(M+R)} \qquad \text{for} \quad \tau + \delta \ll M \; , \tag{2.11}$$

with the MTBF M, restart time R necessary to set up computation from a single checkpoint, the single checkpoint dump time δ and the calculation segment length τ between checkpoints.

Taking failures during the recovery process into account and applying a more precise second order approximation, Daly derived the following formula for MTBF close to the checkpoint dump time

$$\tau_{\text{opt}} = \begin{cases} \sqrt{2\delta M} - \delta & \text{for} \quad \delta < \frac{1}{2}M \\ M & \text{for} \quad \delta \geq \frac{1}{2}M \; . \end{cases} \tag{2.12}$$

So, the optimal checkpoint interval is mainly influenced by the checkpoint writing time and MTBF.

3 I/O optimization and initial configuration preparation

At the beginning of each Molecular Dynamics (MD) simulation an initial phase space configuration has to be created. A point in the phase space is defined by the exact position, orientation, velocity and angular momentum of each molecule. This initial configuration comes with several requirements or constrains from the scenario that the user wants to simulate. For example, the density of molecules and the temperature of the system to be simulated may be given, or the molecular distribution shall exhibit a certain spatial distribution, e.g., in a standard vapor-liquid equilibrium (VLE) simulation.

Before an MD simulation can be used to collect data, it has to undergo an equilibration phase in which the effects of the initially constructed non-physical state are reduced. This is done by simulating the system for some time letting it evolve. Criteria, which can help to decide when a sufficiently good equilibration is achieved are, e.g., the velocity distribution of the molecules or the internal energy and pressure of the system. In order to keep the cost of the equilibration as low as possible, the initial configuration shall be close to a physical state. Therefore, molecules must not be too close to each other and molecule velocities should reflect the desired temperature.

A trivial approach to assign molecules to an initial configuration would be to place them randomly into the simulation volume and add them only to the initial configuration in case that they are not too close to the molecules added to the configuration so far. But this approach is neither efficient nor effective because it relies too much on the creation of random positions, which may not fulfil the placement requirements. So, the probability to pick an appropriate random position decreases with the number of inserted molecules ending—worst case—in an infinite process. This is especially an issue for the creation of molecule configurations with high molecule density as needed for a liquid state simulation. Therefore, the common way to have appropriate molecular distances is to place them on a lattice, which guarantees a minimal distance between molecules [45].

The assignment of velocities is the second task in the setup of the initial configuration. In this case, the constraint from the user comes from the desired temperature. The relation between temperature and the thermal velocity of a set of rigid molecules is given by the formula

$$\langle E_{\text{kin}} \rangle = \frac{1}{2} m \langle v^2 \rangle + \frac{1}{2} L \langle \omega^2 \rangle = \frac{f}{2} k_B T \ , \tag{3.1}$$

with f being the number of degrees of freedom of the molecule [28]. A noble gas atom has three translational degrees of freedom and hence $f = 3$. For a molecule, the number of degrees of freedom has to be incremented from this value by one for each non-vanishing principal moment of the molecule's inertia component. For example, it is $f = 5$ for a linear molecule like nitrogen (N_2) and $f = 6$ for a bent molecule like water (H_2O). At high temperatures, internal degrees of freedom will start playing a role thus further increasing the number of degrees. However, in this work only rigid molecular models are used as it is implemented in ls1-MarDyn.

For very large simulations, the initial setup itself can require a large amount of computational effort. Naturally, the generation of a molecular configuration is done at the beginning of a simulation. But generation of initial configurations is also required for some regeneration strategies used by the application-based fault tolerance (ABFT) techniques described in Chapter 6. Here molecule configuration generation becomes a more frequent task, also necessary during later steps of the simulation. Therefore, the performance of configuration generation becomes an increasingly important issue.

The ls1-MarDyn MD code targets extremely large simulations and therefore has to face the problem of scenario generation at current and future HPC system scales. The focus of this section is to improve ls1-MarDyn's initialisation step. Therefore, a new user-friendly and scalable initialisation concept shall be developed. Starting with an analysis of the current initialisation infrastructure of ls1-MarDyn, the bottlenecks in the implementation and general scenario generation approach are identified.

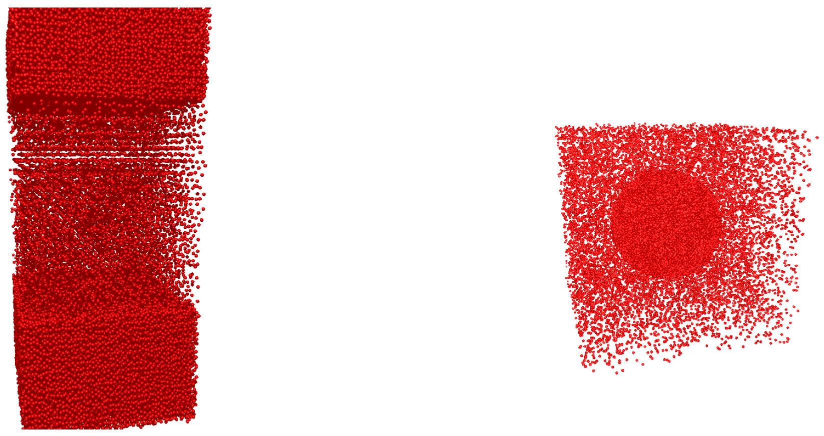

(a) Vapour-liquid interface. (b) Droplet in vapour environment

Figure 3.1: Scenarios used during the ls1-MarDyn I/O system analysis

Where possible, the implementation is then improved gradually. Based on the findings from the improvements and knowhow gained from the existing infrastructure, requirements for a new, scalable scenario setup infrastructure are then defined. From those requirements, a new, generalized concept for initialisation of MD simulations at scale is developed. The concept is then implemented in ls1-MarDyn and evaluated with a real world example.

3.1 I/O implementation and improvement of ls1-MarDyn

3.1.1 Common simulation scenarios with ls1-MarDyn

Common scenarios simulated with ls1-MarDyn are: condensation processes, flow processes at the nanoscopic scale, and liquid vapour phase boundaries in large systems. Figure 3.1 shows two representative system configurations, which are used for the following studies of the I/O part: A vapour-liquid interface and a droplet in a vapour environment. These systems typically consist of many millions of molecules and have to be simulated for several hundred thousand time steps.

3.1.2 Analysis and problems of the original I/O system

The technique used to initialize ls1-MarDyn is implemented around ASCII-based configuration files containing the phase space data including molecular positions and velocities[1]. The generation of these input files is performed by external tools that are developed by the users. The input files generated with these tools are then read independently by all MPI processes of ls1-MarDyn at the start of the simulation.

This I/O system of ls1-MarDyn has several problems when it comes to the simulation of very large systems [69]:

1. The generators written by the users are serial programs. Figure 3.2 shows the input file generation time and write speed for scenarios of different sizes with the animake (homogeneous fluid), mkesfera (droplet), and mkTcTS (VLE) generator. As can be seen, the creation time of the input files increases linearly with the system size. The achieved write speed of around 20 MB/s is slow compared to the theoretical system peak of 6 GB/s [1]. Evidently, the user provided serial generators are not designed to make use of MPI or parallel I/O, which is necessary to achieve good I/O performance. It also has to be noted that these generators often run into problems associated with a limited amount of memory as they build the entire configuration in memory before writing it out to the input file.

2. The used input file format is inefficient and unsuited. The ASCII based input requires parsing and conversion to the CPU's number representation. This requires a fair amount of CPU time compared to binary data. Further, it leads to issues with precision due to the intermediate conversion to base 10 numbers, if generators write out floating point numbers with default precision representation. For example, the default decimal format used in C++ output with `std::cout` from the GCC C++ library has 6 significant digits in scientific notation. With this format, floating point numbers cannot be stored precisely in most cases. For the full representation of a single precision IEEE 754 with a 23 bit mantissa and 8 bit exponent, the scientific notation representation (sign + digits including comma + E + sign + digits) will need in the worst case 14

[1]The same data format is also used to write checkpointing data.

characters and for a double precision IEEE 754 floating point number with a 52 bit mantissa and 11 bit exponent the representation will need in the worst case 24 characters.

3. The I/O system of ls1-MarDyn does not scale in the original version. Figure 3.3 shows the I/O statistics obtained for one process in a 4096 PE run on HLRS Hermit where ls1-MarDyn was started with a small input file containing only 40 000 molecules and a total size of 2.5 MB. While this process performed only one open and one read operation on the input file, the wait and read times are extremely large, resulting in an effective I/O rate of below $1\,\mathrm{MB\,s^{-1}}$. Comparing this to the peak I/O bandwidth of $150\,\mathrm{GB\,s^{-1}}$ of the used lustre file system on Hermit, ls1-MarDyn is far away from utilizing it in an efficient way. The problem here is the parallel access of all MPI processes to the input file via C++ streams, which stresses the system's I/O infrastructure.

In the following sections solutions are developed, which overcome these issues.

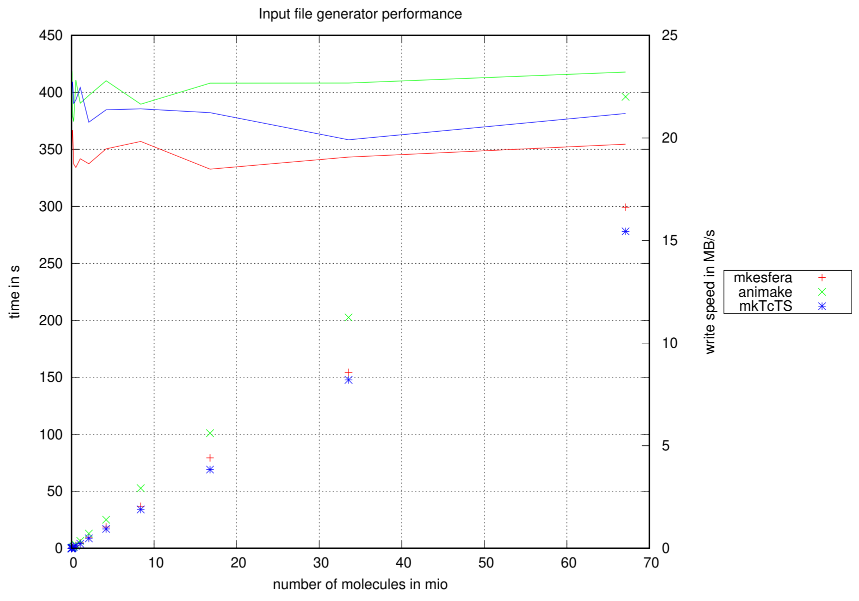


Figure 3.2: Input file generation times with the user provided external animake, mkesfera, and mkTcTS scenario generators for different scenario sizes on HLRS laki/vulcan /lustre/ws2 using gcc 7.3.0

```
PE 1086: File "lj40000_t300.inp"
            Calls     Seconds     MB/sec
Read            1    2.185619   0.003748
Open            1    0.278424
Close           1    0.731858
Buffer Read     2    3.685361   0.569049
I/O Wait        2    2.184243   0.960128
Buffers used        2 (2 MB)
Prefetches          1
```

Figure 3.3: Performance data reported by the Cray *iobuf* tool for the access to a 40 000 molecule input file by process 1086 during a 4096 PE run on HLRS Hermit

3.1.3 Improvement of ls1-MarDyn's original input file I/O

The first of the identified I/O problems of ls1-MarDyn to be addressed is the individual I/O performed by all MPI processes. As described above, the initial configuration is read by all processes individually, which stresses the I/O system due to the concurrent file access. A solution for this problem that does not break compatibility with the existing I/O system and the users' input file creation tools is a master read only approach: Here, only one process reads the input file and broadcasts the data via the fast communication network to all other processes. This approach was implemented in ls1-MarDyn. It has to be noted that the collective MPI parallel I/O is not used here because of the unstructured ASCII file format.

As the input files can become very large, the entire input data cannot be held in memory. Hence, a buffering approach is used at this point. Therefore, the phase space data are split into smaller junks, which are broadcasted. This approach also overlaps the reading of molecules by the master process with the MPI communication and insertion of particles in the remaining processes.

Figure 3.4 shows the scalability of the improved implementation for a variety of phase space configuration sizes on the HLRS Hermit system. With this implementation, the I/O time is close to constant and independent of the number of used processes. It is limited by the capability of a single process to read and parse the ASCII input data. The communication overhead is negligible. The achieved I/O bandwidth is around 268 000 molecules per second. This is one order of magnitude faster than the original version, that achieved 20 000 molecules per second as shown above. However, while this implementation enabled the usage of existing input data at scale, it is still far from optimal due to the limited speed of the required ASCII data parsing.

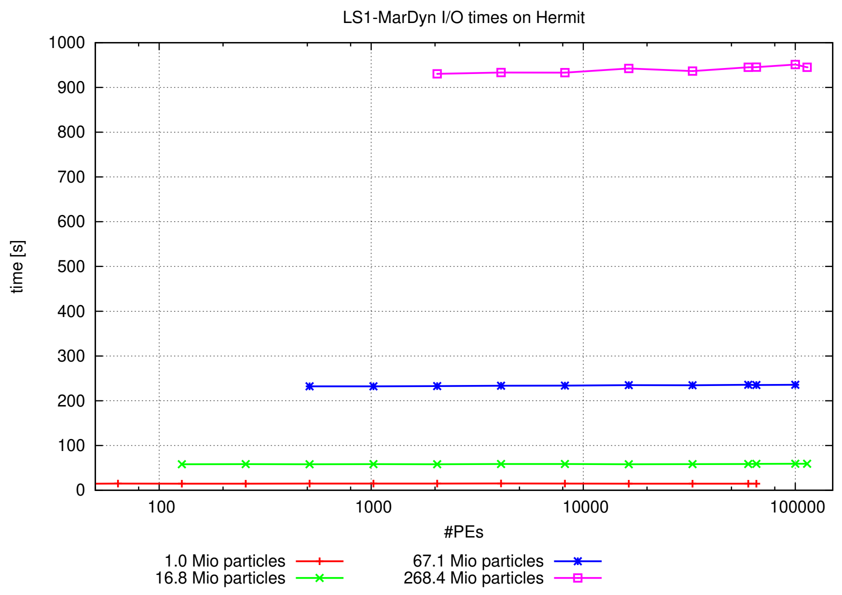


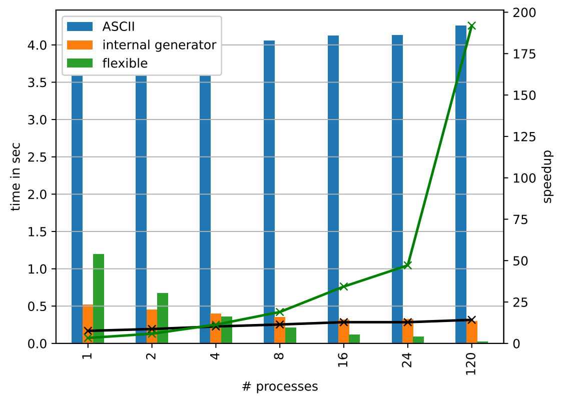
Figure 3.4: Times for the new phase space file I/O part in ls1-MarDyn for different input file sizes on HLRS Hermit

3.1.4 Conversion to internal scenario generation

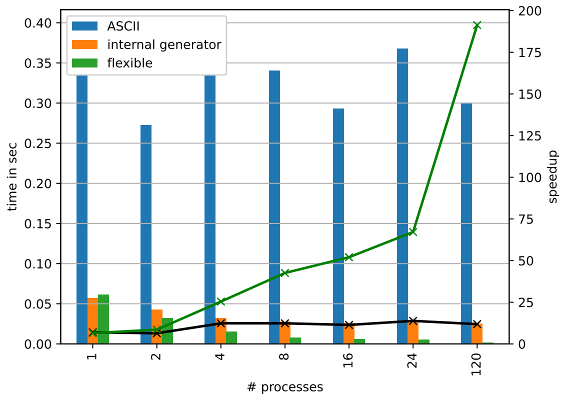
The second problem found in the original I/O system is the ASCII based input file format. From Figure 3.4 it can be seen that even the improved I/O takes considerable time with larger scenarios. A faster solution compared to reading the initial configuration from a file is to generate the phase space data directly in the simulation. At the same time, this solves all problems caused by the ASCII based number representation format.

To demonstrate the benefits of this in-memory generation, the two external tools for the generation of a two layer system (mkTcTS) and a droplet (mkesfera) were adapted to ls1-MarDyn's internal InputBase interface. Instead of writing the phase space data to a phase space configuration file that is read in later by ls1-MarDyn, here the molecules are directly inserted into ls1-MarDyn's in-memory data structure. To steer the generation process, the original command line parameters of the tools were replaced by corresponding parameters in the ls1-MarDyn input file so that all features of the original user tool were preserved.

Figure 3.5 shows the obtained results. The internal generation is around one order of magnitude faster than the improved file based I/O, independently of the number of processes. The slight variations in the speedup are mostly the result from the fluctuations in the times for the ASCII I/O. The figure includes additional data for a corresponding scenario that is set up using the new flexible in-memory scenario generator, which is introduced and described in detail in the following section. This can outperform both of these approaches for higher numbers of processes as it is fully parallelised.



(a) mkTcTS scenario



(b) mkesfera scenario

Figure 3.5: Phase space initialisation times using the improved ASCII reader (blue), internal generator version (orange) of it, and the flexible in-memory generator (green). The data lines show the achieved speedup in comparison to the improved ASCII reader (black: internal generator, green: flexible in-memory generator).

3.2 Flexible in-memory scenario generation

The study of surface and flow phenomena at the nanoscopic scale is one of the main target-fields of ls1-MarDyn. This often requires the creation of initial states with arbitrary geometries that are filled with varying molecule densities. Examples are the study of wetting effects simulated from an droplet generated on a structured surface or the investigation of liquid flow in a nano-nozzle simulated from a liquid in a solid channel [13].

While the inclusion of the initial configuration generators directly into ls1-MarDyn is the way to overcome time and memory limits of external tools, it is not flexible with respect to fast prototyping. Also the task of parallelization of the generator is left to the developer of the generator—which most of the users are not capable of as we have seen from the original toolset. Therefore, it is desirable to provide a flexible generator, which is configurable via the input file.

The requirements for this generator are: (1) the possibility to define arbitrary 3D objects, (2) to allow the filling of objects in different ways with molecules (gas, liquid, solid), (3) the assignment of individual temperatures for different objects, and (4) a transparent parallelization of the scenario generation process.

3.2.1 Implementation overview

The concept of the flexible in-memory scenario generator, which was developed within this work, is to create the desired 3D geometry and populate it with molecules. The molecules are assigned with velocities according to a given temperature and velocity distribution. The entire generation process is parallelized using the knowledge of the process local subdomains.

To allow the description of arbitrary objects, the generator includes a constructive solid geometry layer. Once an object is defined, molecules can be placed inside using a so-called filler. A filler for arbitrary Bravais lattices is included and used as the basis for other fillers. To set the temperature of an object, the velocity of the molecules can be modified using velocity assigners. An assigner for an initial Maxwell Boltzmann velocity distribution as well as an assigner for equal valued velocities with random orientation are provided. The molecules are created one by one to reduce the memory requirements to a minimum.

Figure 3.6 shows the general design of the flexible in-memory scenario generator, which was implemented as an I/O plugin in ls1-MarDyn named ObjectGenerator. The ObjectGenerator itself is a composition of the object to be filled, the filler used to assign positions, and the velocity assigner. All components are again implemented as plugins to allow for maximal flexibility and easy extendability.

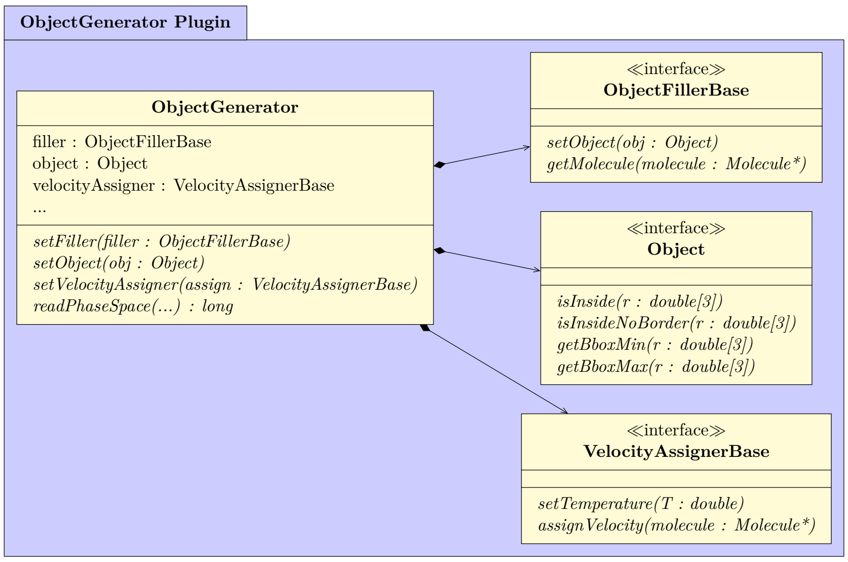

ObjectGenerator Plugin
ObjectGenerator
filler : ObjectFillerBase
object : Object
velocityAssigner : VelocityAssignerBase
...
setFiller(filler : ObjectFillerBase)
setObject(obj : Object)
setVelocityAssigner(assign : VelocityAssignerBase)
readPhaseSpace(...) : long
≪interface≫
ObjectFillerBase
setObject(obj : Object)
getMolecule(molecule : Molecule*)
≪interface≫
Object
isInside(r : double[3])
isInsideNoBorder(r : double[3])
getBboxMin(r : double[3])
getBboxMax(r : double[3])
≪interface≫
VelocityAssignerBase
setTemperature(T : double)
assignVelocity(molecule : Molecule*)

Figure 3.6: Design of the flexible scenario generator in ls1-MarDyn implemented as ObjectGenerator plugin

3.2.2 Arbitrary volume objects

To build arbitrary volume objects constructive solid geometry is used [40]. Here, arbitrary volumes are described by a set of primitive 3D objects which are combined by boolean operations into a object tree. For the functionality of the generator, each object has to implement the methods `isInside()`, `isInsideNoBorder()` and `getBoundingBoxMin/Max()` returning the coordinates of a cuboidal bounding box around the object. The decision if a point is inside a volume object is necessary for the molecule insertion. The bounding box becomes important for the efficient filling of the objects later on.

Within this work, cuboids, spheres and cylinders were implemented as primitive object types. More primitives can be added easily at any time following the provided interface. Abstract objects implementing boolean operations for subtraction, intersections and union are provided and can be used to create more complex volumes.

Volume union The union is implemented to return for the bounding box the minimum point created by the two bounding box minima and for `isInside()`, if the specified point lies at least within one of the two objects.

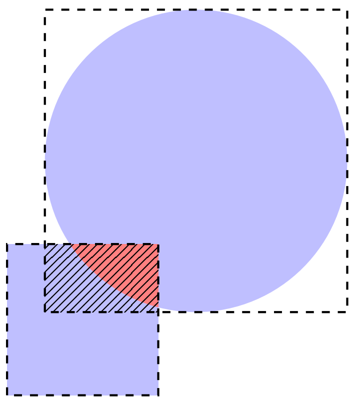


Figure 3.7: Intersection (red) of two objects (blue) in 2D. The bounding box of the new object (shaded) is created as intersection of the original bounding boxes (dashed lines) and may not be the minimal bounding box.

Volume subtraction The subtraction is implemented to return for the bounding box the bounding box of the original object and for `isInside()`, if the specified point lies in the original object but not in the subtracted one.

Volume intersection The intersection of two objects determines if a point lies within both objects. A bounding box of the object created by the intersection is determined by using the bounding boxes of the two original bounding boxes and intersecting those. It has to be noted here, that this approach to obtain a bounding box may not provide the minimal bounding box of the new object in all cases as demonstrated in Figure 3.7. More sophisticated approaches as, e.g., iterative bounding box refinement [66], may be used here. However, as a non-optimal bounding box is not necessary for the general functionality of the generator, such sophisticated improvements are left for the future at this point.

3.2.3 Placing molecules into objects

The second functionality that is required, is the insertion of molecules into the defined volume objects. Therefore, so called fillers were implemented, which are applied to the objects. As already mentioned the common technique to create an initial configuration for MD simulations is placement of molecules on a lattice. Beside the generation of solids, this helps also creating configurations for liquids by ensuring a minimal distance between insertion points.

Lattice description In solid state physics, the structure of a crystal is described by a Bravais lattice [57] at whose lattice sites a basis, consisting out of one or multiple atoms or molecules, is positioned. The construction is shown in Figure 3.8.

The Bravais lattice is defined as an infinite group of lattice points $\vec{r'}$ in space, which can be described by the combination of a lattice system

$$\vec{r'} = \vec{r} + n_1\vec{a}_1 + n_2\vec{a}_2 + n_3\vec{a}_3 \qquad \text{with} \qquad n_1, n_2, n_3 \in \mathbb{Z} \ , \tag{3.2}$$

where $\vec{a}_1, \vec{a}_2, \vec{a}_3$ are the lattice vectors and $\vec{r}$ is an arbitrary base point of the lattice and a lattice centering: primitive (P), body (I), face (F) and base (A, B or C). In three dimensions, 14 distinct Bravais lattice types exist.

At each point of the lattice, a basis consisting of one or more atoms or molecules is positioned. The position of the atoms or molecules in the basis can be described by their relative positions $\vec{r}_j$ to a lattice point

$$\vec{r}_j = x_{j,1}\vec{a}_1 + x_{j,2}\vec{a}_2 + x_{j,3}\vec{a}_3 \qquad \text{with} \qquad 0 \leq x_{j,1}, x_{j,2}, x_{j,3} < 1 \ . \tag{3.3}$$

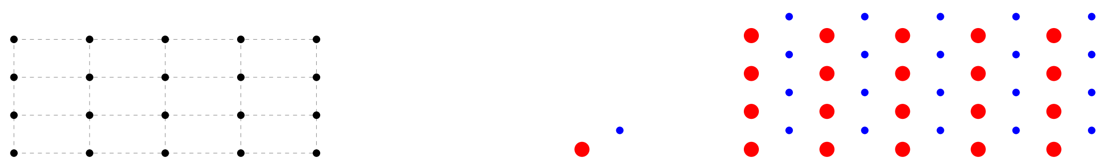

(a) Lattice (b) Two atomic basis (c) Crystal structure

Figure 3.8: Crystal structure description: Attaching to each point of the lattice in (a) the basis from (b) creates the crystal structure shown in (c).

While this describes an infinite lattice, a real solid will have a finite extent.

Grid Filler

A filler plugin was implemented for the placement of molecules on a lattice inside an object. The design of this grid filler follows the physical description and requires a Bravais lattice and a basis as input. Additionally, a base point for the lattice and a spatial extent have to be provided in addition to the object to be filled with molecules.

The lattice is created by specifying the lattice system, lattice centering and three lattice vectors or alternatively the desired density. As there are restrictions for the possible lattices, the plugin allows to check, if the provided input parameters represent a valid lattice description, e.g., if the chosen lattice centering is allowed for the selected lattice system. In this case, a cubic lattice is assumed and the lattice vectors are adapted appropriately.

The basis is defined by relative coordinates and type identifiers. The position of each atom or molecule in the basis is described using the relative coordinates in Equation (3.3). The atom or molecule type is specified by a component id.

The object to be filled is provided to the grid filler as well as a base point $\vec{G}$ where the previously defined lattice shall be attached in space. From these data, the grid filler computes the necessary extent for the lattice to overlap the entire object. The extent is defined internally as multiples of the lattice vectors with the base point $\vec{G}$ as coordinate system origin. Because the lattice vectors can have arbitrary orientation and length, the filling of the volume with the lattice is a non trivial task: To fill the volume with lattice points in an efficient manner, the minimal lattice coordinate range required to overlap the entire object with a finite lattice has to be determined as shown in Figure 3.9.

This is achieved by a coordinate transformation of the coordinates of the object's bounding box corners $\vec{r}_i$ into the coordinates $\vec{c}_i$ in the system defined by the lattice vectors $\vec{a}_1$, $\vec{a}_2$, and $\vec{a}_3$ according to

$$\begin{pmatrix} \vec{a}_1 & \vec{a}_2 & \vec{a}_3 \end{pmatrix} \vec{c}_i = \vec{r}_i \ . \tag{3.4}$$

The minimum ceiling and maximum floor values from the components of the coordinate vectors $\vec{c}_i$ are selected and the lattice points in this range are checked, omitting all points outside the specified volume.

Stochastic molecule positioning

While the insertion of molecules using a Bravais lattice is helpful to keep minimal distances between inserted molecules, it does not represent a fluid or gas state very well. To get a starting point closer to those states, molecule positions have to be distributed

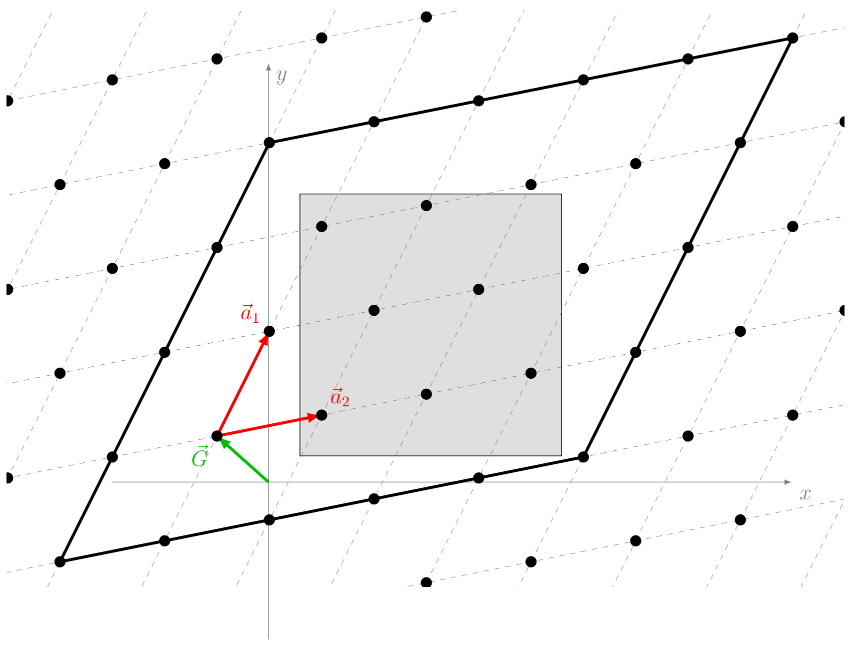


Figure 3.9: Lattice generation within a rectangular area in 2D: The lattice is defined by the vectors $\vec{a}_1, \vec{a}_2$ and the base point $\vec{G}$. To find all points (black dots) that are in the shaded area, the lattice points within the area marked by the solid black line are checked.

more randomly in space. Therefore, two different strategies were implemented: Random shifting of positions and incomplete lattice filling.

Random shifting of positions for fluids The lattice structure is far away from the unordered structure of the liquid state. To generate a more fluid-like state, random shifting of the lattice positions can be used [82]. Here each molecule i is moved by a different random shift $\Delta\vec{r_i}$ from its lattice position. The random shift $\Delta\vec{r_i}$ for each molecule is determined using the three lattice vectors, where each vector is multiplied with a uniformly distributed random variable $p_{i,j}$ in the range $[0, 1]$ and a given global maximal displacement factor $k \in [0, 1]$:

$$\Delta\vec{r_i} = k \sum_{j\in\{1,2,3\}} p_{i,j}\vec{a}_j \tag{3.5}$$

Incomplete filling of a lattice for gas states Sometimes random shifting of positions is not sufficient. To introduce more randomness in the placement of molecules, e.g., for gas phases, incomplete filling of a fine grained lattice is used. Here, molecules are only assigned to a part of the lattice points and the others are left empty. The parameter to control the randomness of positions in this approach is the occupancy $\eta = \frac{\text{used lattice positions}}{\text{total lattice positions}}$. Thereby the size of the lattice vectors for the fine grid hast to be at least as small as

$$l = \sqrt[3]{\frac{n}{\eta\rho}} , \tag{3.6}$$

with n being the number of sites in the lattice's unit cell. The lattice sites are then populated with the probability η. Therefore, a random number is used for each position to decide wether a molecule should be placed or not.

While this approach is very efficient, it can end up with a deviation of the density from the target value for small number of molecules, depending on the random numbers. Since this may become an issue for a simulation, Robert Floyd's algorithm can be used to address this shortcoming [15]. This is a time and memory efficient algorithm to select m distinct values from a set of n values with the complexity $\mathcal{O}(m \log(m))$. However, the observable deviation in density becomes smaller for larger number of molecules and is not relevant for the tests and use cases in this work. Therefore, the implementation of this additional correction is left out for now.

Replica Filler

A second common approach to set up MD simulations is the reuse of already equilibrated configurations. Therefore, the Replica Filler was created, which allows to read an existing configuration and repeat it in all dimensions to fill out the entire object. The implementation is based on the previously described Grid Filler with a primitive cubic lattice. The lattice vectors are chosen to be the extents of the equilibrated configuration and the configuration itself is used as the basis of the lattice.

3.2.4 Initial molecule velocities

The velocity of each molecule has to be set for the initial configuration. The average velocity of molecules is correlated to the desired system temperature according to Equation (3.1). In most cases, the velocities of the molecules follow the Maxwell

Boltzmann distribution, while in special scenarios this may be different. Two approaches were implemented as velocity assigner plugins:

- **EqualVelocityAssigner:** Assignment of velocities with equal absolute value but with random orientation. Here one can select between randomly chosen velocity orientations along the grid coordinates and uniformly distributed velocity orientations on the sphere based on [99].
- **MaxwellVelocityAssigner:** Assignment based on the Maxwell Boltzmann distribution with uniformly distributed velocity orientations on the sphere.

At the end of the velocity assignment to all molecules, the global momentum of the system is computed. This momentum is then corrected to prevent an unwanted drift of the system.

3.2.5 Parallel particle insertion

The master broadcast optimization presented in Section 3.1.3 and the porting of the external generators to internal ones in Section 3.1.4 improved the I/O, but both modifications kept the input time constant as they did not make use of parallelism for the generation step itself. The new flexible in-memory scenario generator addresses this problem: The initial configuration creation itself is parallelized using the provided domain decomposition.

Each process generates molecules based on the specified generation methods and object definitions only in its local subvolume. To determine the region to be filled by a process, the ObjectGenerator takes the object to be filled with molecules and intersects it with an object representing the process's associated subvolume. The resulting object is then passed to the filler and velocity assigner. The implementation in ls1-MarDyn is straight forward and makes use of the already existing constructive solid geometry functionality from Section 3.2.2.

Solving the problem of unique molecule identifiers

In ls1-MarDyn each molecule is assigned a unique identifier (ID) to allow tracking of trajectories or for self-correlation computations. However, now that all processes generate the initial molecule configuration in parallel, a new challenge arises: The original code assigns unique identifiers using a simple global counter, which gets incremented with every globally added molecule. For the parallel creation, such a global counter becomes a bottleneck. Therefore, an ID pool was introduced, which guarantees that no ID is assigned twice. Different ways to implement such an ID pool exist. In this case, the simple approach of ID space subdivision was used: The maximal number of IDs, which can be represented by the ID type (`unsigned long`), is divided in as many equal sized blocks as there are processes. Each process then gets one of those blocks and assigns IDs only from his block starting with his block's lowest ID. This approach does not require any synchronisation between the processes and exhibits perfect scalability as the number of processes as well as the maximal ID range are known to every process and does not require any communication.

3.2.6 Application example

In the following, a geometry to simulate a micro nozzle is used as an example to demonstrate the flexibility and performance of the new in-memory generator. The model configuration of the nozzle consists out of two layers of a solid, where one has an indentation and the facing side an elevation. In the gap between the solid layers is a gas. The geometry is constructed using the primitive objects in combination with unification and subtraction methods. For the generation of the solid parts, the GridFiller is used with a face centered cubic (fcc) lattice, 100% fill and no random shifting. The gas is created using the GridFiller with the method of partial lattice filling (fcc lattice) with an occupancy of 30 %. Molecule velocities are assigned to the different parts according to the given temperatures. For the solid, molecule velocities are assigned according to the specified temperature as equal velocities with random orientation and for the gas using the Maxwell distribution. Figure 3.10 shows the entire scenario.

Results of a strong scaling experiment on Hazel Hen with this scenario are shown in Figure 3.11. The scenario was run for 100 time steps with the standard domain decomposition. As can be seen, the phase space creation time is reduced with more processes reaching a speed-up of 30 for 120 and 80 for 1200 processes.

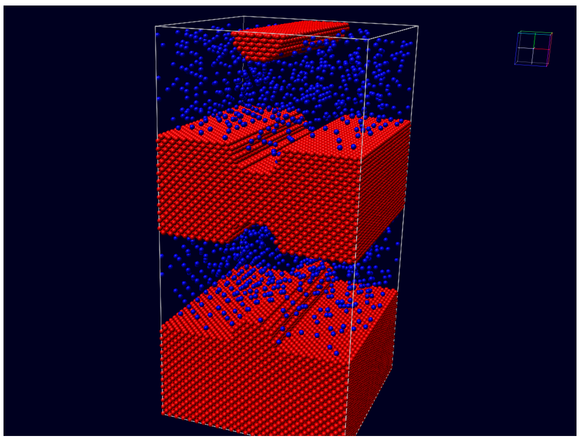


Figure 3.10: Nozzle configuration created with the arbitrary object creation functionality. Blue: fluid, red: solid, total number of molecules: 67 541

This can be considered good, as it is on par with the scalability of the computation.

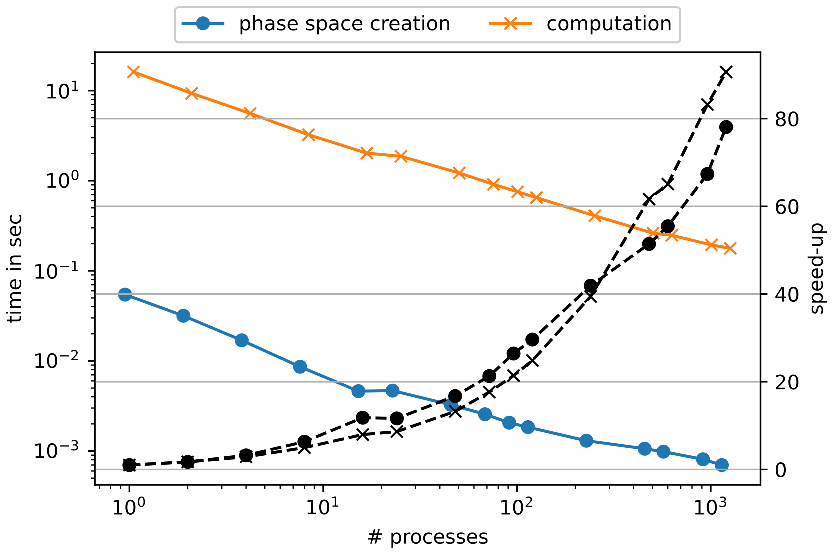


Figure 3.11: Scenario generation times and speedup for the nozzle configuration scenario with the flexible in-memory scenario generator on Hazel Hen. Runtime spent for phase space generation with the in-memory generator and computation follows the coloured solid lines, speed-up follows the dashed back lines.

3.3 Chapter summary

In this chapter, the I/O system of ls1-MarDyn was analysed. Several issues with the original implementation were identified including the serial nature of the external generators, the overheads of the ASCII based file format and non-existing scalability of the file I/O.

First, a broadcast based solution to improve the scalability of the original I/O plugin was presented. The implementation of it eliminated contention issues caused by the concurrent I/O of all processes to the same file found in the original code. It was shown that with this modification ls1-MarDyn could now be used on more than 100 000 MPI processes.

Second, the problem of the ASCII file format and its slow parsing speed was addressed by moving the external generators into the ls1-MarDyn code to create the scenarios in-memory. The superiority of the internal in-memory over the external generation was demonstrated by showing that scenario generation speed was improved by an order of magnitude. Both improvements together enabled runs at full system scale of Hazel Hen, which were not possible before.

The third problem addressed in this chapter was the need for a scalable but simple generator for complex scenarios, which can be used by users without knowledge of parallelization techniques to set up arbitrary large scenarios. Therefore, a new flexible in-memory generator was developed and implemented in ls1-MarDyn. The implementation includes a constructive solid geometry layer to create arbitrary geometries, two different grid based particle placement methods as well as two temperature assignment functionalities based on constant but random oriented or Boltzmann distributed velocities. Scalability of the scenario generation is achieved by the use of a domain decomposition approach. The capabilities of this new flexible in-memory generator were demonstrated with the original user scenarios as well as a complex nozzle geometry scenario. For the latter, the new generator showcased not only its flexibility and user-friendliness, but also an excellent scalability of the user-transparent parallelization—and so provides a replacement for all existing external generator solutions in ls1-MarDyn.

4 Node level performance analysis/engineering

There are different aspects, which have to be considered with respect to node-level performance: The choice of the algorithms, the tuning of these algorithms for the target hardware, and an efficient parallelization for multi-core systems. In the following, all three aspects are addressed for Molecular Dynamics (MD) simulation codes. Therefore, the first section of this chapter will address the development of a performance model for the important algorithms in the field, which can help to choose the appropriate algorithm for a simulation scenario. The second section will evaluate different implementations of the most important algorithms on different hardware architectures. And the last section will study the parallelization of the commonly used linked-cell algorithm with a task-based parallelization approach.

4.1 Performance modelling of basic MD algorithms

The usage of an efficient algorithm to solve a problem can reduce the time to solution drastically. Therefore, a good understanding of the available algorithms is crucial. The well established algorithms used in MD simulations were already presented in Section 2.3.3. The asymptotic time complexity of these algorithms is well-known but gives only information about their behaviour for an infinite number of molecules. However, in the case of highly parallel MD codes, a domain decomposition is used and the large problem is divided into many smaller sub-problems. Therefore, the behaviour of the algorithms for a smaller, limited amount of molecules is of interest and the knowledge of the algorithm's asymptotic complexity turns out to be less important. Moreover, some algorithms have identical complexity but differ in their efficiency with regard to particle densities, or when they are faced with dynamic, fast changing system configurations. So, a large number of comparisons of implementations for specific scenarios can be found in the literature [8, 62, 94, 101]. However, this does not help much for a deeper understanding of the best algorithm choice.

In order to improve the understanding of the algorithms, theoretical cost models are developed in this section of the work. The models target specifically the description of the algorithms' behaviour for small molecule counts. The models are then evaluated for a set of critical parameters and compared to experimental results from a set of sample implementations of the algorithms. The obtained cost models will help later in the decision finding process, which algorithm is used best for a specific scenario—either by the user or automatically by the program.

4.1.1 Description of the studied system

The following study is limited to short range interaction potentials. Further, a system with a homogeneous particle distribution inside a cubic box with periodic boundary conditions is assumed. The system shall consist of N particles with an average particle density ρ. Choosing a homogeneous particle distribution does not limit the applicability of the later results to homogeneous systems as an inhomogeneous system can be decomposed in most cases into smaller, almost homogeneous sub-systems, to which the results can then be applied.

4.1.2 Computational cost models

The cost models for the algorithms shall describe their efficiency for a molecular system consisting of N molecules with homogeneous density ρ, where the molecule-molecule interaction range is limited by the cut-off radius r_c. Recapping from Section 2.3.3, the most common algorithms used for the force calculation of short-ranged interaction potentials in MD simulations are the naïve algorithm with a cut-off, the Verlet neighbour list algorithm, and the linked-cell algorithm. All three have common algorithmic sub parts: 1) distance calculation, 2) particle sorting into bins, and 3) potential calculation.

Each of the three algorithmic subparts is assigned with a computational cost parameter, which reflects the computational costs for a single molecule-molecule interaction. The parameters are denoted c_{dist}, c_{bin}, and c_{pot}, respectively.

Potential calculation

The computational cost for the potential calculation of all chosen algorithms using short-range potentials in combination with a cut-off can be modelled in the same way. This is obvious, as all algorithms have to handle the same number of molecular interactions. With the cut-off radius r_{c}, the potential calculation cost C_{pot} for a system with N molecules is approximated by

$$C_{\mathrm{pot}} = c_{\mathrm{pot}} N \left(\frac{4}{3} \pi r_{\mathrm{c}}^3 \rho - 1 \right) , \tag{4.1}$$

where the factor c_{pot} describes the costs for the potential calculation of a single pair interaction and the factor in brackets is an approximation for the number of molecules that a single molecule interacts with inside a sphere of radius r_{c} around it.

Naïve algorithm with cut-off

In the naïve algorithm, the distance between all atoms is computed and only for those within a specified cut-off radius, the pair interaction potential and forces are computed. So the computational costs consist of the distance calculations and the potential evaluation:

$$C^{(\mathrm{NA})} = C_{\mathrm{dist}}^{(\mathrm{NA})} + C_{\mathrm{pot}} , \tag{4.2}$$

where the distance calculation costs are

$$C_{\mathrm{dist}}^{(\mathrm{NA})} = c_{\mathrm{dist}} N (N-1) , \tag{4.3}$$

with c_{dist} specifying the costs for a single distance calculation.

Often periodic boundary conditions are used to simulate bulk systems. Those can be implemented in two ways: By direct implementation in the distance calculation or by the introduction of modified copies of molecules at the simulation volume boundaries, which are removed again after the force computation. The first approach leads to increased distance calculation costs c_{dist}. The second approach leads to an increased number of particles to be iterated over.

The modified distance based approach is not very flexible and requires the adaptation of the force kernel for the specific target scenario dependent on the chosen periodicity and geometry. The copy based approach is more flexible and can easily be implemented together with the often used parallelization via domain decomposition, which already handles the periodicity and geometry. So the later approach will be used in the following and the cost model is adapted accordingly.

To estimate the number of additional molecule copies N' necessary for the boundary handling in the model, the best case of a cubic simulation domain with volume V is assumed. Therewith, the approximate number of additional molecules is given by

$$N' = \rho \left(6 r_{\text{c}} l^2 + 12 r_{\text{c}}^2 l + 8 r_{\text{c}}^3\right) \qquad \text{with} \qquad l = \sqrt[3]{N/\rho} \tag{4.4}$$

and the modified cost model becomes

$$C_{\text{dist}}^{(\text{NA})} = c_{\text{dist}} N (N + N' - 1) \; . \tag{4.5}$$

Linked-cell algorithm

The link cell algorithm reduces the necessary number of distance calculations by binning of the molecules into cells with a length of the cut-off radius as described in Section 2.3.3. Thus, the neighbour molecules of a molecule are found in its own and the neighbouring cells. The search space for neighbour molecules shrinks from the entire simulation volume to $27 r_{\text{c}}^3$.

So, the algorithmic costs $C^{(\text{LC})}$ for the link cell algorithm consist of the costs for binning molecules into cells $C_{\text{bin}}^{(\text{LC})}$, the distance calculations $C_{\text{dist}}^{(\text{LC})}$, and the potential evaluation C_{pot}:

$$C^{(\text{LC})} = C_{\text{bin}}^{(\text{LC})} + C_{\text{dist}}^{(\text{LC})} + C_{\text{pot}} \; . \tag{4.6}$$

Here the binning costs are

$$C_{\text{bin}}^{(\text{LC})} = c_{\text{bin}} N \; , \tag{4.7}$$

with c_{bin} describing the binning cost per molecule. The distance calculation costs are

$$C_{\text{dist}}^{(\text{LC})} = c_{\text{dist}} N \left(27 \rho r_c^3 - 1 \right) , \tag{4.8}$$

with the factor in brackets approximating the number of molecules to be checked for a single molecule to interact with its own and the surrounding 26 cells of diameter r_{c}.

This model for the linked-cell algorithm includes already the boundary handling with a boundary width of r_{c} and implies the computational overheads due to additional halo volume management around the regular simulation volume. In the case of periodic boundaries, the halo contains modified copies of molecules as already discussed for the naïve algorithm.

Verlet neighbour list algorithm

The Verlet neighbour list approach stores neighbour information for each molecule as described in Section 2.3.3. This information is updated in regular intervals. Hereafter, the update frequency shall be denoted by p. Therewith, the update interval becomes $1/p$ time steps. Molecules are added to the neighbour list of a molecule, if they are within a skin radius $r_c + r_s = kr_c$ with $k > 1$, which guarantees that no molecule from $r > r_s$ enters $r \leq r_c$ until the next neighbour list update. So, the computational costs in the model consist of the neighbour list creation costs occurring every $1/p$ time steps and the distance calculation and potential evaluation costs in each time step:

$$C^{(\text{NL})} = pC_{\text{list}}^{(\text{NL})} + C_{\text{dist}}^{(\text{NL})} + C_{\text{pot}} . \tag{4.9}$$

Commonly, the neighbour lists are created with the help of the linked-cell algorithm, so the list creation costs are

$$C_{\text{list}}^{(\text{NL})} = c_{\text{bin}} N + c_{\text{dist}} N \left[27 \rho (k r_c)^3 - 1 \right] . \tag{4.10}$$

The distance calculation costs for the Verlet neighbour list algorithm are

$$C_{\text{dist}}^{(\text{NL})} = c_{\text{dist}} N \left[\frac{4}{3} \pi \rho (k r_c)^3 - 1 \right] . \tag{4.11}$$

4.1.3 First exploration of the model

In order to get a first impression for the computational cost parameters and the algorithm cost models, a set for c_{dist}, c_{bin}, and c_{pot} is obtained using streaming kernels, written in pure C, which perform distance calculations, sorting into bins, and potential evaluation for the Lennard-Jones (LJ) potential, respectively. All input arrays for the streaming kernels have a size of 1000 elements to reflect the small amount of molecules. As a consequence, all kernels will run out of the CPU cache.

Table 4.1 presents the obtained results on different hardware architectures. The distance computation parameter is found to be the smallest and the potential parameter the largest. This can be explained by the fact that the potential evaluation uses a division operation, which is more expensive to be performed by the hardware than simple additions or multiplications. For example, on the Intel Haswell the SSE/AVX division operation VDIVPD alone has a reciprocal throughput of $8-14$ cycles [39]. The also large binning parameter is caused by expensive rounding operations in the index calculation. Despite the lower frequency of the SX-ACE CPU, which is running at only 1 GHz, it shows comparable performance to the x86 processors. Hence, it makes effective use of its vector architecture even for this small vector length of 1000 elements [34].

	Intel Haswell	Intel Skylake	NEC SX-ACE	
	gcc 8.2.0	gcc 8.2.1	sxcc 1.0r112	
	`-O3 -mavx2`	`-O3 -mavx2`		
c_{dist} in ns	18.7	1.9	1.0	1.6
c_{bin} in ns	71.0	3.8	5.0	8.9
c_{pot} in ns	77.8	3.8	7.0	7.4

Table 4.1: Cost parameters for MD algorithm cost models obtained from streaming kernel runs. In each streaming kernel run the average time for 1000 distance computations, binning operations, and potential evaluations is determined. The final cost parameters are chosen as the minimal value found over a set of 100 runs.

Based on the obtained cost parameters for the Intel Skylake, a first impression for the cost models is given in Figure 4.1 using the typical factor $k = 1.3$ and update frequency $p = 1/20$.

Figure 4.1a shows the runtime prediction from the models for a scenario of density $\rho = 0.8$ using a cut off $r_\mathrm{c} = 2.5$. It can be seen that the naïve algorithm is supposed to be faster than the link cell for less than 340 molecules and faster than the neighbour list algorithm for less than 150 molecules. Also, the neighbour list algorithm is expected to be faster than the linked-cell algorithm for all numbers of molecules.

Figure 4.1b shows the predicted runtime for scenarios with $N = 150$ and $N = 1000$ molecules using a cut off $r_\mathrm{c} = 2.5$. For the former, the model suggests the naïve algorithm to be more efficient than the linked-cell and neighbour list algorithm for densities below 0.3 and 0.8, respectively. For the latter, the neighbour list algorithm is predicted to be the best choice.

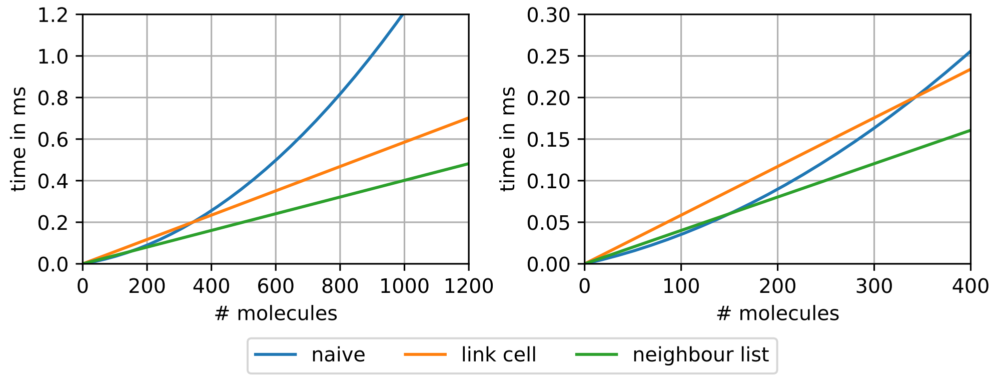
Figure 4.1c shows the predicted runtime for scenarios with $N = 150$, $N = 340$, and $N = 1000$ molecules using a density $\rho = 0.8$. Here the crossover point for the linked-cell and neighbour list algorithm to be more efficient than the naïve algorithm moves to larger cut off values for higher number of molecules. According to the model, the naïve algorithm can compete for $N = 1000$ with the two other algorithms down to a cut off $r_\mathrm{c} = 5$.

4.1.4 Algorithm selection

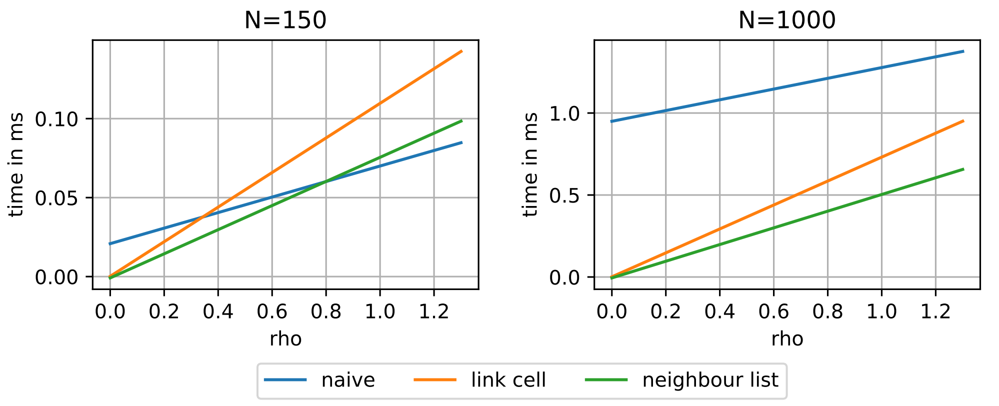
With the mathematical cost models to describe the performance of the algorithms, it is now possible to make predictions which algorithm should be best for a given scenario. The theoretical efficiency crossover function between the basic algorithms can be determined from the computational costs per molecule. To make the decision easier, the difference of the cost models is calculated and assigned to functions. Here, the cost parameters are assumed to be the same for the algorithms. Depending on the function value being positive or negative, the one or the other algorithm is more efficient.

Crossover between NA and LC Subtracting the per molecule costs of the link cell algorithm from the naïve one gives

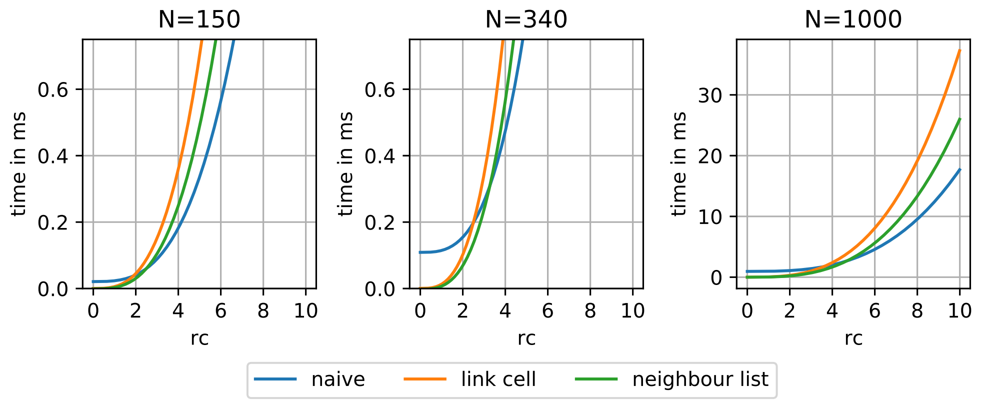
$$f_{\mathrm{NA}\leftrightarrow\mathrm{LC}}(N, \rho, r_\mathrm{c}) = \frac{1}{N}\left\{C^{(\mathrm{NA})} - C^{(\mathrm{LC})}\right\} = c_\mathrm{dist}(N - 27\rho r_\mathrm{c}^3) - c_\mathrm{bin}\ . \tag{4.12}$$



(a) #molecule dependence ($\rho = 0.8$, $r_c = 2.5$, $k = 1.3$, and $p = 1/20$)



(b) density dependence ($r_c = 2.5$, $k = 1.3$, and $p = 1/20$)



(c) cutoff dependence ($\rho = 0.8$, $k = 1.3$, and $p = 1/20$)

Figure 4.1: Predicted single time step runtime from the algorithm cost models using parameters obtained from the streaming kernels $t_{\text{dist}} = 0.95\,\text{ns}$, $t_{\text{bin}} = 5.01\,\text{ns}$, and $t_{\text{pot}} = 7.15\,\text{ns}$.

So, the linked-cell algorithm is more efficient for

$$N - 27\rho r_\mathrm{c}^3 > \frac{c_\mathrm{bin}}{c_\mathrm{dist}} \; . \tag{4.13}$$

Crossover between NA and NL Subtracting the per molecule costs of the neighbour list algorithm from the naïve one gives

$$\begin{aligned} f_{\mathrm{NA}\leftrightarrow\mathrm{NL}}(N,\rho,r_\mathrm{c}) &= \frac{1}{N}\left\{C^{(\mathrm{NA})} - C^{(\mathrm{NL})}\right\} \\ &= c_\mathrm{dist}\left(N - \left[\left(\frac{4}{3}\pi + 27p\right)\rho(kr_c)^3 - p\right]\right) - pc_\mathrm{bin} \end{aligned} \tag{4.14}$$

So, the neighbour list algorithm is more efficient for

$$N - \left(\frac{4}{3}\pi + 27p\right)\rho(kr_c)^3 - p > p\frac{c_\mathrm{bin}}{c_\mathrm{dist}} \; . \tag{4.15}$$

Crossover between LC and NL Subtracting the per molecule costs of the neighbour list algorithm from the link cell one gives

$$\begin{aligned} f_{\mathrm{LC}\leftrightarrow\mathrm{NL}}(\rho,k,p) &= \frac{1}{N}\left\{C^{(\mathrm{LC})} - C^{(\mathrm{NL})}\right\} \\ &= (1-p)c_\mathrm{bin} + c_\mathrm{dist}\left[27\rho r_c^3\left(1 - k^3(p + \frac{4}{3}\pi)\right) + p\right] \; . \end{aligned} \tag{4.16}$$

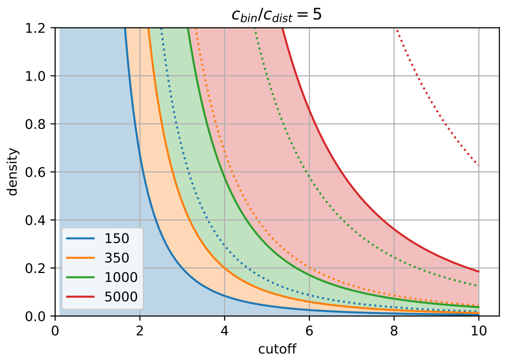
So, the neighbour list algorithm is more efficient for

$$27\rho r_c^3\left(k^3(p + \frac{4}{3}\pi - 1)\right) + p > (1-p)\frac{c_\mathrm{bin}}{c_\mathrm{dist}} \; . \tag{4.17}$$

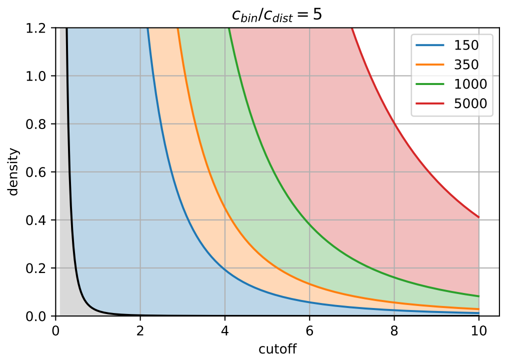
Interestingly, this crossover is independent of the number of particles N. Also, the update frequency $1/p$ turns out to play only a minor role as p is typically much smaller than one.

Algorithm selection diagrams From Equations (4.13) and (4.15), it is obvious that the crossover function for best performance between the linked-cell and naïve algorithm as well as the one between the neighbour list and naïve algorithm depend on the number of molecules N and the fraction c_{bin}/c_{dist}. However, looking at Equation (4.17), the crossover function between the linked-cell and the neighbour list algorithm depends only on the fraction c_{bin}/c_{dist}.

The three Equations (4.13), (4.15) and (4.17) are summarized for the common values $c_{\mathrm{dist}}/c_{\mathrm{bin}} = 5$, $k = 1.3$, and $p = 1/20$ in the algorithm selection diagrams shown in Figure 4.2. Herein, the solid lines represent the crossover condition $f_*(\rho, k, p) = 0$ for different number of molecules N. To the right of the crossover line, the naïve algorithm is favoured and to the left of the line, the linked-cell and neighbour list algorithm are advantageous. In Figure 4.2a, the dotted lines mark the end of the applicability of the linked-cell algorithm. To the right of this, the cutoff radius becomes larger than the diameter of the simulation volume and therefore a single cell used in the link-cell algorithm will cover the entire volume. In Figure 4.2b, the grey area on the left marks the region, where the linked-cell algorithm is advantageous over the neighbour-list algorithm—independently of the number of molecules according to Equation (4.17).



(a) Linked-cell to naïve algorithm performance crossover for different number of molecules. Linked-cell is faster in the filled areas left of the crossover line (solid lines). The dashed lines mark the upper limit up to which the link cell algorithm can be used with a cell length of r_c.



(b) Neighbour list to naïve algorithm performance crossover for different number of molecules. Neighbour list is faster in the filled areas left of the crossover line (solid line). The black line marks the crossover between the link cell and the naïve algorithm, in the grey area link cell is favoured.

Figure 4.2: Algorithm selection diagrams for $c_{\mathrm{dist}}/c_{\mathrm{bin}} = 5$, $k = 1.3$, $p = 1/20$

Two interesting results can be found from the performance models. First, the linked-cell algorithm is found to be advantageous over the neighbour list algorithm for small cutoffs and lower densities, independently of the number of molecules. Conversely, the neighbour-list algorithm is favoured for larger cutoffs and higher densities. Second, the naïve algorithm may be of interest as well for scenarios with a small number of molecules. Thinking of highly parallel simulations running with many processes, often less than 1000 molecules per process are used. With a typical cutoff of $r_c = 5$

for higher precision simulations, all of a sudden the naïve algorithm is expected for densities above around 0.6.

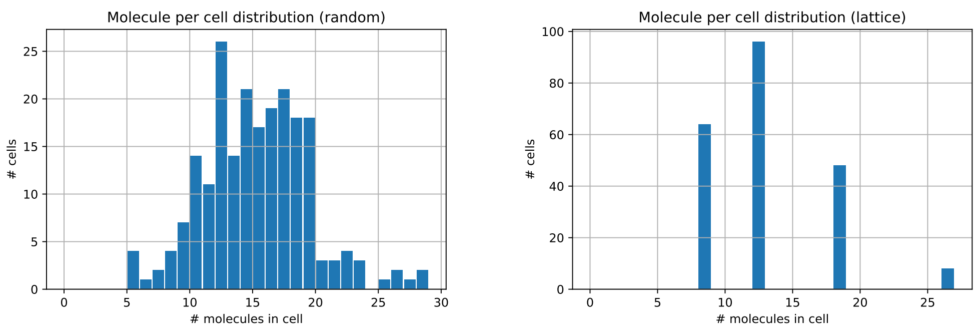
4.1.5 Cost model validation

To validate the cost models, the basic algorithms were implemented as benchmark kernels in C++, which were then used to compute different scenarios. To make the benchmark kernels good representatives of real applications, the linked-cell and neighbour list benchmark kernels were implemented according to [80].

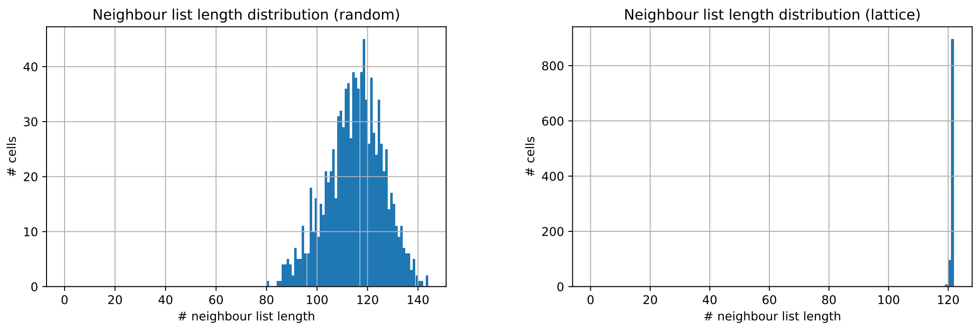
As scenarios, two different particle distributions are used with the initial state: (1) random placement, which represents gases and liquids, and (2) placement on a regular lattice, which represents solids. The different spatial particle distributions affect the distribution of particles to the cells in the linked-cell algorithm parts as well as the length of the neighbour lists. Examples for the distribution of particles to cells are shown in Figure 4.3a. The corresponding neighbour list length distribution is shown in Figure 4.3b. The randomly placed distribution shows a Gaussian distribution-like population of the cells as well as neighbour list lengths. The lattice based distribution shows a clear structure with some distinct values being more likely.

The time to perform the computation of those scenarios with different densities ρ, particle numbers N, cut-off radii r_c, and the common algorithms is determined and fitted to the obtained cost models using the cost-parameters c_{dist}, c_{bin}, and c_{pot} as fitting parameters. As a starting point for the fitting parameters, the values obtained from the streaming kernels are used. However, these values are obviously not perfect as they do not take into account the interaction of the different code parts.

Runtimes obtained with the benchmark kernels for the scenarios for different numbers of molecules and densities using a cut-off $r_c = 2.5$ are provided in Table 4.2 and Table 4.3. In case of the lattice based initialization, the initial distance between molecules end up larger than the cutoff as long as the cutoff is not chosen to be larger than $r_c = 10$. Because it is not common to use larger cutoff radii than this with the studied algorithms in real applications, these values are omitted for this scenario.



(a) Particles per cell distribution



(b) Neighbour list length distribution

Figure 4.3: Characteristics for the linked-cell and neighbour list algorithms when used with a random and lattice based particle distribution in a scenario with $N = 1000$, $\rho = 0.8$, $r_c = 2.5$.

Comparing the runtime of the two scenarios, the computations for the scenarios using lattice initialisation are overall faster than the ones using random placement. There are two reasons for this. First, the overall number of interactions to be computed is on average around 5 % higher for the random placed initial configurations, e.g., for 1000 molecules 29 290 interactions versus 28 000 interactions. Second, there are more distance calculations to be performed. Both is the result of the particle distribution and the fact that the cell population and neighbour list length contribute quadratically to the computational costs.

The runtime data from Table 4.2 are used as input for the fitting of the algorithm models. The fit is performed using the `lmfit` python module [68], which implements an extended version of the Levenberg–Marquardt algorithm [79], to achieve a least-squares fit. The fit parameters are constrained to positive values below 1 millisecond. An additional constant c_0 was added to the model to take into account any constant overhead from the benchmark setup. While not relevant for real applications, data

N	0.9	0.8	0.7	0.003	0.002	0.001
1	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$1.07 \cdot 10^{-6}$	0	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$
8	$4.05 \cdot 10^{-6}$	$4.05 \cdot 10^{-6}$	$7.03 \cdot 10^{-6}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$
27	$3.7 \cdot 10^{-5}$	$3.6 \cdot 10^{-5}$	$3.46 \cdot 10^{-5}$	$3.1 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$2.15 \cdot 10^{-6}$
64	$1.2 \cdot 10^{-4}$	$1.16 \cdot 10^{-4}$	$1.09 \cdot 10^{-4}$	$1.91 \cdot 10^{-5}$	$8.11 \cdot 10^{-6}$	$1.79 \cdot 10^{-5}$
125	$2.84 \cdot 10^{-4}$	$2.74 \cdot 10^{-4}$	$2.43 \cdot 10^{-4}$	$5.15 \cdot 10^{-5}$	$3.6 \cdot 10^{-5}$	$3.39 \cdot 10^{-5}$
216	$5.8 \cdot 10^{-4}$	$5.18 \cdot 10^{-4}$	$5.12 \cdot 10^{-4}$	$7.89 \cdot 10^{-5}$	$7.81 \cdot 10^{-5}$	$8.5 \cdot 10^{-5}$
512	$1.86 \cdot 10^{-3}$	$1.72 \cdot 10^{-3}$	$1.63 \cdot 10^{-3}$	$3.61 \cdot 10^{-4}$	$3.44 \cdot 10^{-4}$	$3.04 \cdot 10^{-4}$
1,000	$4.92 \cdot 10^{-3}$	$4.51 \cdot 10^{-3}$	$4.27 \cdot 10^{-3}$	$1.09 \cdot 10^{-3}$	$1.04 \cdot 10^{-3}$	$9.88 \cdot 10^{-4}$

(a) naïve algorithm

N	0.9	0.8	0.7	0.003	0.002	0.001
1	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$
8	$1.91 \cdot 10^{-6}$	$1.55 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$	$4.05 \cdot 10^{-6}$	$4.05 \cdot 10^{-6}$	$1.36 \cdot 10^{-5}$
27	$4.89 \cdot 10^{-5}$	$5.04 \cdot 10^{-5}$	$4.8 \cdot 10^{-5}$	$1.88 \cdot 10^{-5}$	$1.98 \cdot 10^{-5}$	$2.19 \cdot 10^{-5}$
64	$1.66 \cdot 10^{-4}$	$1.63 \cdot 10^{-4}$	$1.55 \cdot 10^{-4}$	$2.91 \cdot 10^{-5}$	$2.98 \cdot 10^{-5}$	$4.49 \cdot 10^{-5}$
125	$2.92 \cdot 10^{-4}$	$2.53 \cdot 10^{-4}$	$2.44 \cdot 10^{-4}$	$4.41 \cdot 10^{-5}$	$5.01 \cdot 10^{-5}$	$6.81 \cdot 10^{-5}$
216	$5.46 \cdot 10^{-4}$	$5.04 \cdot 10^{-4}$	$4.97 \cdot 10^{-4}$	$6.89 \cdot 10^{-5}$	$7.81 \cdot 10^{-5}$	$1.21 \cdot 10^{-4}$
512	$1.13 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$9.42 \cdot 10^{-4}$	$1.57 \cdot 10^{-4}$	$1.97 \cdot 10^{-4}$	$2.59 \cdot 10^{-4}$
1,000	$1.93 \cdot 10^{-3}$	$1.85 \cdot 10^{-3}$	$1.75 \cdot 10^{-3}$	$2.97 \cdot 10^{-4}$	$3.42 \cdot 10^{-4}$	$4.92 \cdot 10^{-4}$

(b) linked-cell algorithm

N	0.9	0.8	0.7	0.003	0.002	0.001
1	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$2.15 \cdot 10^{-6}$	$2.5 \cdot 10^{-6}$	$1.44 \cdot 10^{-5}$
8	$1.91 \cdot 10^{-6}$	$2.15 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$	$1.94 \cdot 10^{-5}$	$2 \cdot 10^{-5}$	$3.9 \cdot 10^{-5}$
27	$6.91 \cdot 10^{-6}$	$8.58 \cdot 10^{-6}$	$7.7 \cdot 10^{-5}$	$4.65 \cdot 10^{-5}$	$5.7 \cdot 10^{-5}$	$8.65 \cdot 10^{-5}$
64	$3.19 \cdot 10^{-4}$	$3.1 \cdot 10^{-4}$	$2.95 \cdot 10^{-4}$	$7.56 \cdot 10^{-5}$	$9.89 \cdot 10^{-5}$	$1.83 \cdot 10^{-4}$
125	$7.77 \cdot 10^{-4}$	$7.15 \cdot 10^{-4}$	$6.65 \cdot 10^{-4}$	$1.28 \cdot 10^{-4}$	$1.87 \cdot 10^{-4}$	$3.51 \cdot 10^{-4}$
216	$1.46 \cdot 10^{-3}$	$1.37 \cdot 10^{-3}$	$8.62 \cdot 10^{-4}$	$2.03 \cdot 10^{-4}$	$2.99 \cdot 10^{-4}$	$5.63 \cdot 10^{-4}$
512	$3.15 \cdot 10^{-3}$	$2.94 \cdot 10^{-3}$	$2.79 \cdot 10^{-3}$	$5.29 \cdot 10^{-4}$	$6.71 \cdot 10^{-4}$	$1.26 \cdot 10^{-3}$
1,000	$5.02 \cdot 10^{-3}$	$4.7 \cdot 10^{-3}$	$4.44 \cdot 10^{-3}$	$9.64 \cdot 10^{-4}$	$1.33 \cdot 10^{-3}$	$2.62 \cdot 10^{-3}$

(c) neighbour list algorithm

Table 4.2: MD benchmark kernel runtimes for different densities and numbers of molecules N using the random initialisation method and a cut-off of $r_c = 2.5$. Values are the median of at least five measurements given in seconds.

N	0.9	0.8	0.7
1	$9.54 \cdot 10^{-7}$	0	$1.19 \cdot 10^{-6}$
8	$4.05 \cdot 10^{-6}$	$5.01 \cdot 10^{-6}$	$8.11 \cdot 10^{-6}$
27	$3.7 \cdot 10^{-5}$	$3.7 \cdot 10^{-5}$	$3.1 \cdot 10^{-5}$
64	$1.07 \cdot 10^{-4}$	$1.04 \cdot 10^{-4}$	$8.2 \cdot 10^{-5}$
125	$2.06 \cdot 10^{-4}$	$2.29 \cdot 10^{-4}$	$1.84 \cdot 10^{-4}$
216	$4.46 \cdot 10^{-4}$	$4.46 \cdot 10^{-4}$	$3.72 \cdot 10^{-4}$
512	$1.42 \cdot 10^{-3}$	$1.4 \cdot 10^{-3}$	$1.24 \cdot 10^{-3}$
1,000	$3.65 \cdot 10^{-3}$	$3.71 \cdot 10^{-3}$	$3.44 \cdot 10^{-3}$

(a) naïve algorithm

N	0.9	0.8	0.7
1	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$
8	$2.15 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$	$9.54 \cdot 10^{-7}$
27	$4.41 \cdot 10^{-5}$	$4.91 \cdot 10^{-5}$	$4.01 \cdot 10^{-5}$
64	$1.46 \cdot 10^{-4}$	$1.48 \cdot 10^{-4}$	$1.27 \cdot 10^{-4}$
125	$2.2 \cdot 10^{-4}$	$2.47 \cdot 10^{-4}$	$2.04 \cdot 10^{-4}$
216	$4.58 \cdot 10^{-4}$	$4.54 \cdot 10^{-4}$	$3.63 \cdot 10^{-4}$
512	$8.62 \cdot 10^{-4}$	$8.86 \cdot 10^{-4}$	$6.85 \cdot 10^{-4}$
1,000	$1.65 \cdot 10^{-3}$	$1.68 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$

(b) linked-cell algorithm

N	0.9	0.8	0.7
1	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$	$9.54 \cdot 10^{-7}$
8	$1.19 \cdot 10^{-6}$	$1.91 \cdot 10^{-6}$	$1.19 \cdot 10^{-6}$
27	$4.05 \cdot 10^{-6}$	$6.91 \cdot 10^{-6}$	$6.48 \cdot 10^{-5}$
64	$2.65 \cdot 10^{-4}$	$2.63 \cdot 10^{-4}$	$2.26 \cdot 10^{-4}$
125	$4.91 \cdot 10^{-4}$	$5.41 \cdot 10^{-4}$	$4.9 \cdot 10^{-4}$
216	$9.87 \cdot 10^{-4}$	$9.73 \cdot 10^{-4}$	$6.3 \cdot 10^{-4}$
512	$2.38 \cdot 10^{-3}$	$2.37 \cdot 10^{-3}$	$2.14 \cdot 10^{-3}$
1,000	$4.24 \cdot 10^{-3}$	$4.23 \cdot 10^{-3}$	$3.74 \cdot 10^{-3}$

(c) neighbour list algorithm

Table 4.3: MD benchmark kernel runtimes for different densities and number of molecules N using the lattice initialisation method and a cut-off radius of $r_\mathrm{c} = 2.5$. Values are the median of at least five measurements given in seconds.

measurements for a single molecule are included to help with the fit of the c_0 parameter here. The resulting parameters are listed in the Tables in Figure 4.4 and Figure 4.5.

The fitted models are in good accordance with the measured data, especially for the dense scenarios. However, the obtained fitting parameters differ noticeably between the dense and sparse scenarios.

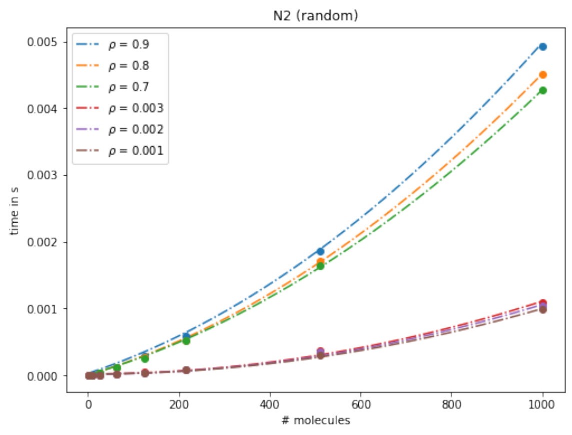
For the scenarios with higher density, the fits for the random and the lattice based state are good. Especially the naïve algorithm is modelled very well in its behaviour. The distance computational cost parameter c_{dist} for all algorithms is found to be in the range 1.3 ns to 4.5 ns. This is close to the 1.9 ns obtained from the streaming kernel in Table 4.1. The binning cost parameter c_{bin} from the fits is in the range 0.3 µs to 5.7 µs. This is around two orders of magnitude higher than expected when comparing to the 3.8 ns from the streaming kernel version. The potential cost parameter c_{pot} is in the range 4.8 ns to 44.5 ns and slightly higher than the expected 3.8 ns from the streaming kernel. The additional overhead constant c_0 is relatively constant for all models and in the range 1.1 µs to 41.3 µs.

The scenarios with low density and random initial positioning show a very flat runtime behaviour compared to the higher density scenarios. This is explained by the fact that there are now very few interactions within the cut-off radius. Again, the naïve algorithm is modelled very well and the cost models from the fits are close to the streaming kernel results. However, the data for the linked-cell and neighbour list algorithm are not as well in accordance with the model. The distance computational cost parameter c_{dist} for them is found to be in the range 1.9 ns to 41.0 ns and the binning cost parameter c_{bin} is in the range 0.8 µs to 3112.3 µs. So these values exhibit a very high uncertainty with a trend towards high values. The overhead constant c_0 is found to be in the range 6.7 µs to 21.4 µs, and therefore fluctuates less than for the higher density scenarios.

In summary, the cost parameters resulting from the fits are overall higher than the once obtained from the streaming kernels. One reason is the increased complexity of the benchmark code compared to the streaming kernels. This complexity comes along with some minor additional computational overheads. Also the increased complexity prevents some compiler optimizations, e.g., vectorization.

ρ	c_0 in µs	c_{dist} in ns	c_{pot} in ns
0.900	27.0	2.7	39.1
0.800	1.1	2.5	40.1
0.700	2.0	2.3	44.5
0.003	20.0	1.1	1.2
0.002	16.0	1.0	0.8
0.001	16.0	1.0	0.4

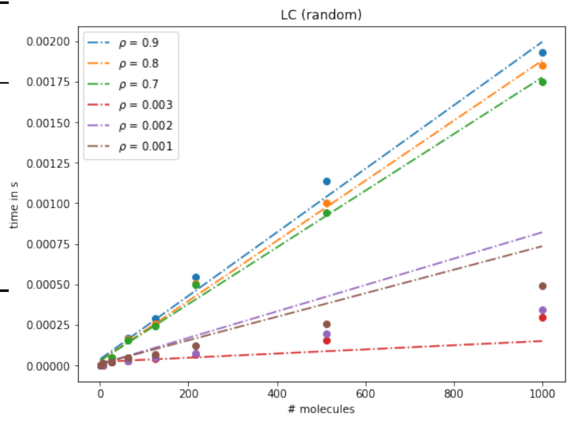
(a) Fitting parameters



(b) Original data and fit

ρ	c_0 in µs	c_{dist} in ns	c_{bin} in µs	c_{pot} in ns
0.900	36.4	2.2	0.7	7.9
0.800	25.6	4.0	0.3	4.8
0.700	27.5	2.4	0.5	12.1
0.003	21.4	30.8	0.1	26.9
0.002	6.7	1.9	0.8	0.8
0.001	9.3	38.8	3.0	2,374.1

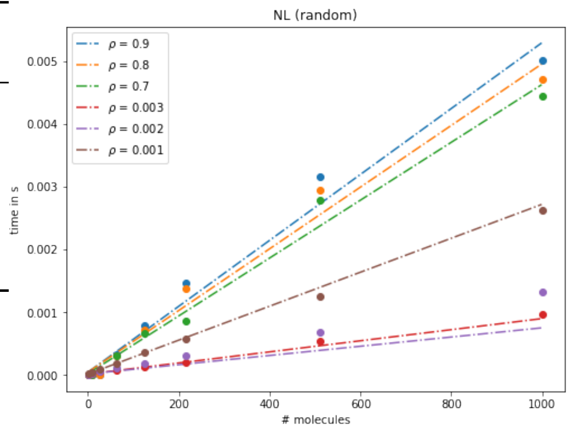
(c) Fitting parameters



(d) Original data and fit

ρ	c_0 in µs	c_{dist} in ns	c_{bin} in µs	c_{pot} in ns
0.900	40.9	2.7	1.4	22.9
0.800	41.3	2.8	1.3	24.1
0.700	15.8	4.5	0.5	15.7
0.003	10.9	20.2	2.0	1,473.5
0.002	13.2	26.8	2.0	1,447.5
0.001	10.2	41.0	5.7	3,112.3

(e) Fitting parameters

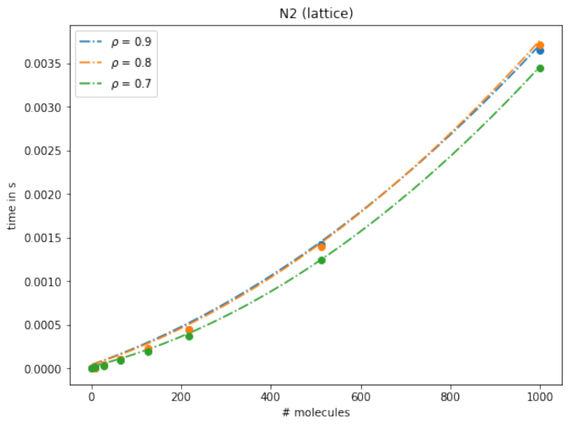


(f) Original data and fit

Figure 4.4: Fit of algorithm cost models for different scenarios initialized with random particle positions and using a cut-off of radius $r_c = 2.5$

ρ	c_0 in µs	c_{dist} in ns	c_{pot} in ns
0.900	37.3	1.9	31.1
0.800	36.6	2.0	33.0
0.700	14.9	2.1	29.0

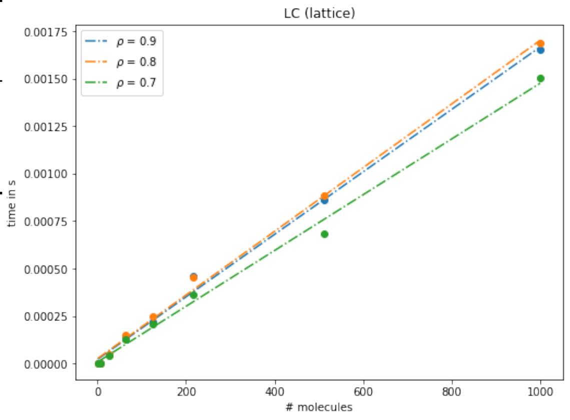
(a) Fitting parameters



(b) Original data and fit

ρ	c_0 in µs	c_{dist} in ns	c_{bin} in µs	c_{pot} in ns
0.900	21.1	2.8	0.3	5.3
0.800	24.5	2.3	0.4	9.2
0.700	6.8	2.5	0.3	8.8

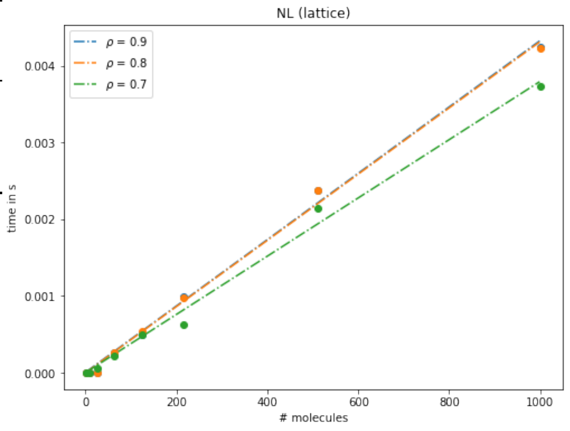
(c) Fitting parameters



(d) Original data and fit

ρ	c_0 in µs	c_{dist} in ns	c_{bin} in µs	c_{pot} in ns
0.900	5.9	1.3	2.3	14.0
0.800	0.2	3.3	0.7	15.5
0.700	1.2	2.3	0.6	33.3

(e) Fitting parameters



(f) Original data and fit

Figure 4.5: Fit of algorithm models for different scenarios initialized with lattice based particle positions and using a cut-off radius of $r_c = 2.5$

4.2 Analysis of the naïve algorithm

The cost model developed in Section 4.1 showed that the naïve algorithm is more efficient for small number of molecules and higher molecule densities, compared to the linked-cell and neighbour list algorithms. Also, the distance and force computation in the linked-cell algorithm itself is often implemented by calculating all interactions between molecules in two cells using the naïve algorithm. Therefore, it is of importance to have a very efficient implementation of it. To find such an efficient implementation for a specific system is the target of this section.

4.2.1 Naïve algorithm implementations

To compute intermolecular forces for all pair interactions, the naïve algorithm in its basic form uses two loops over all molecules. For short range potentials, it skips interactions based on a cutoff distance as described in Section 2.3.3. However, it can be implemented in many different ways.

In general, the computation of the forces between N molecules with the naïve algorithm can be interpreted as a force matrix $\boldsymbol{F}$ with dimensions $N \times N$, where columns and rows correspond to the molecules and the matrix elements F_{ij} to the interaction forces between pairs of them. Depending on the association of the matrix dimensions to molecules in the force calculation, a matrix element F_{ij} is either the force on molecule i or j. In the following, the matrix elements F_{ij} represent the force on molecule i caused by molecule j. Due to Newton's 3rd law, the force matrix is antisymmetric and so $F_{ij} = -F_{ji}$. Figure 4.6 shows the force matrix and the different directions used to loop through the interactions in the following algorithm implementations. The total force F_i onto a molecule i is then the sum of all contributions in a row $F_i = \sum_j F_{ij}$ or alternatively the sum in a column $F_j = \sum_i F_{ij}$.

The following list contains the different variants of naïve kernel implementations evaluated in this work. In the code listings `interaction(i, j)` computes the interaction between molecule i and j and adds the result to the total value for molecule i. The `apply_newton(i, j)` method is used as a placeholder for the application of Newton's 3rd law to reuse an already computed force contribution from molecule i and add it to the total value for molecule j. For each implementation that does not

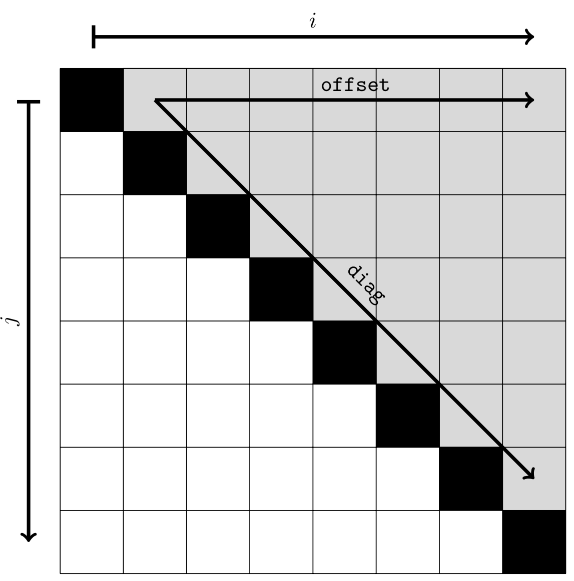


Figure 4.6: Visualization of the force matrix (F_{ij}) in the naïve algorithm. Cells represent the inter-molecular forces F_{ij}. Black cells are excluded as they represent self-interactions. Gray cells represent the upper part of the force matrix. Arrows indicate the directions of the loops used in the different algorithm implementations.

use Newton's 3rd law, two versions are implemented, from which one updates the forces for F_i and the other for F_j. This influences the memory access pattern and is therefore likely to have an influence on the performance.

The following implementations were studied:

1. **ij-loop with if `i!=j`:** This version is a straightforward way of the implementation using two loops over all molecules and an if statement to compute contributions only when $i \neq j$.

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    if(i != j)
      interaction(i, j);
```

2. **ij-loop with continue:** This version is a straightforward way of the implementation using two loops over all molecules and an if statement to skip computations with a continue in the case of $i = j$.

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    if(i == j)
      continue;
    interaction(i, j);
```

3. **ij-loop with Newton:** This version computes only the elements in the upper half of the force matrix and uses Newton's 3rd law to fill the lower half.

```
for(i = 0; i < N-1; i++)
  for(j = i+1; j < N; j++) {
    interaction(i, j);
    apply_newton(j, i);
  }
```

4. **two ij-loops:** This version computes the upper and lower triangular part of the force matrix separately, skipping the diagonal elements with $i = j$.

```
for(i = 0; i < N-1; i++)
  for(j = i+1; j < N; j++)
    interaction(i, j);
for(i = 1; i < N; i++)
  for(j = 0; j < i; j++)
    interaction(i, j);
```

5. **two ji-loops:** This version computes the upper and lower triangular part of the force matrix separately, skipping the diagonal elements with $i = j$. The loop order is interchanged compared to the two ij-loops version.

```
for(j = 0; j < N-1; j++)
  for(i = j+1; i < N; i++)
    interaction(i, j);
for(j = 1; j < N; j++)
  for(i = 0; i < j; i++)
    interaction(i, j);
```

6. **ij-loop + ji-loop:** This version computes the upper and lower triangular part of the force matrix separately, skipping the diagonal elements with $i = j$. The loop order is interchanged between the two parts: One part using ij and the other ji ordering.

```
for(j = 0; j < N-1; j++)
  for(i = j+1; i < N; i++)
    interaction(i, j);
for(i = 0; i < N-1; i++)
  for(j = i+1; j < N; j++)
    interaction(i, j);
```

7. **ij-loop from diagonal:** This version computes the elements of the force matrix row-wise. Computation starts from the diagonal of the force matrix to the right and continues periodically on the left, when reaching the right end. The $i = j$ diagonal is skipped by starting from offset one and ending at offset $N - 1$.

```
for(i = 0; i < N; i++)
  for(long offset = 1; offset < N; offset++) {
    j = (i + offset) % N;
    interaction(i, j);
  }
```

8. **diagonal with offset:** This version computes the force matrix along diagonals. The diagonals are shifted cyclically from left to right. Self-interactions are left out by skipping the matrix diagonal itself.

```
for(long offset = 1; offset < N; offset++) {
  for(i = 0; i < N; i++)
    j = (i + offset) % N;
    interaction(i, j);
  }
```

9. **diagonal with offset Newton:** This version computes the upper triangular part of the force matrix looping diagonally over the upper triangular force matrix part by increasing the offset to the diagonal. Newton's 3rd law is used to obtain the lower triangular part.

```
for(offset = 1; offset < N-1; offset++)
  for(i = 0; i < N-offset; i++) {
    j = i + offset;
    interaction(i, j);
    apply_newton(j, i);
  }
```

10. **diagonal two loop blocks:** This version uses two loop blocks to compute the upper and lower triangular part of the force matrix looping diagonally over the force matrix parts by increasing the offset to the diagonal.

```
for(offset = 1; offset < N; offset++)
  for(i = 0; i < N-offset; i++) {
    j = i + offset;
    interaction(i, j);
  }
for(offset = 1; offset < N; offset++)
  for(i = 0; i < N-offset; i++) {
    j = i + offset;
    interaction(j, i);
  }
```

11. **diagonal using two loop blocks ji:** This version uses two loop blocks to compute the upper and lower triangular part of the force matrix looping diagonally over the force matrix parts by increasing offset to the diagonal. Loop order is interchanged compared to the diagonal using two loop blocks ij version.

```
for(offset = 1; offset < N; offset++)
  for(j = 0; j < N-offset; j++) {
    i = j + offset;
    interaction(i, j);
  }
for(offset = 1; offset < N; offset++)
  for(j = 0; j < N-offset; j++) {
    i = j + offset;
    interaction(j, i);
  }
```

4.2.2 Naïve algorithm implementation performance and numerical differences

The naïve algorithm implementations described in the previous section were implemented as benchmark kernels with the C++ programming language. The molecules' positions, forces, and potential energies are stored in a structure of arrays (SoA) layout using multiple `std::vector` with elements of type `double`. Elements are accessed via the index operator `std::vector::operator[]` as other access methods, e.g., `std::vector::at()`, introduce overheads due to bounds checking. The interaction kernel computes the distance and LJ potential with cut-off for the molecule pairs. The number of interactions is checked to be the same for all kernels as a sanity measure.[1]

The results from the `ij-loop with if i!=j` implementation are used as a reference to identify influences of numerical rounding effects, which can, e.g., be caused by different hardware instruction execution orders. As a metric for the comparison, the normalized root mean square deviation (NRMSD) for the molecular forces is used.

For the following tests, a homogeneous scenario with $r_c = 3.0$, $\rho = 0.9$, and $N = 125$ were used. This set of parameters represents a typical scenario for which the pre-

[1]The counting of computed interactions is preformed in a separate run so it does not disturb the actual performance measurement due to the additional overhead.

viously derived algorithm selection diagram in Figure 4.2 favours the naïve algorithm.

The algorithm implementations are run on different systems with a variety of different compilers. The used systems include different nodes from the HLRS Laki/Vulcan cluster as well as HLRS Kabuki. The specifications of the CPUs on those nodes are listed in Table 4.4. Compilers include gcc, Intel, PGI and NEC. For the compiler options, the following optimization levels are used: `-O3` for the gcc and Intel, `-O3 -Minline=level:20` for the PGI compiler and `-Caopt -pi,auto` for the NEC SX compiler. The obtained execution times for the kernels are listed in Table 4.5 and are shown together with the NRMSD values for the forces in Figure 4.7a. The results show clear differences between the various implementations and platforms in terms of performance and exhibit smaller differences when it comes to the numerical result.

CPU	f in GHz	$f_{\max}$ in GHz	cores
Intel Sandy Bridge (Intel Xeon E5-2670)	2.6	3.3	8
Intel Haswell (Intel Xeon E5-2680 v3)	2.5	3.3	12
Intel Skylake (Intel Xeon Gold 6138)	2.0	3.7	20
NEC SX-ACE	1.0	1.0	4

Table 4.4: Specifications of CPUs used in the naïve algorithm implementation comparison. f: nominal frequency for all cores, $f_{\max}$: maximal turbo frequency for a single core, cores: number of physical cores

The x86 based Intel Sandy Bridge, Haswell and Skylake systems show identical behaviour when it comes to the different algorithms. All results are close together in terms of performance and accuracy. As expected, the newer CPU generations are slightly faster than their predecessors. The best implementation for the scenario here is `diagonal with offset Newton`. Looking at the results obtained with the gcc 8.2.0 compiler, Haswell is 21 % faster than Sandy Bridge, and Skylake is 21 % faster than Haswell. However, the kernels are executed on a single core and therefore, the CPUs run at their maximum turbo speed. Skylake has an 6 % performance advantage against Haswell due to its higher turbo speed. Comparing the results for the different compilers on Skylake, the Intel compiler achieves the best results. On average over all implementations, its results are around 6 % faster than the gcc once. Though, the fastest implementation `diagonal with offset Newton` is only marginally faster with an 0.7 % advantage for Intel over gcc. On the older Haswell, the gcc compiler turns out to provide the best results with an average performance advantage of 4 % over the Intel compiler. For the fastest implementation `diagonal with offset Newton` gcc achieves even a 7.6 % lead here.

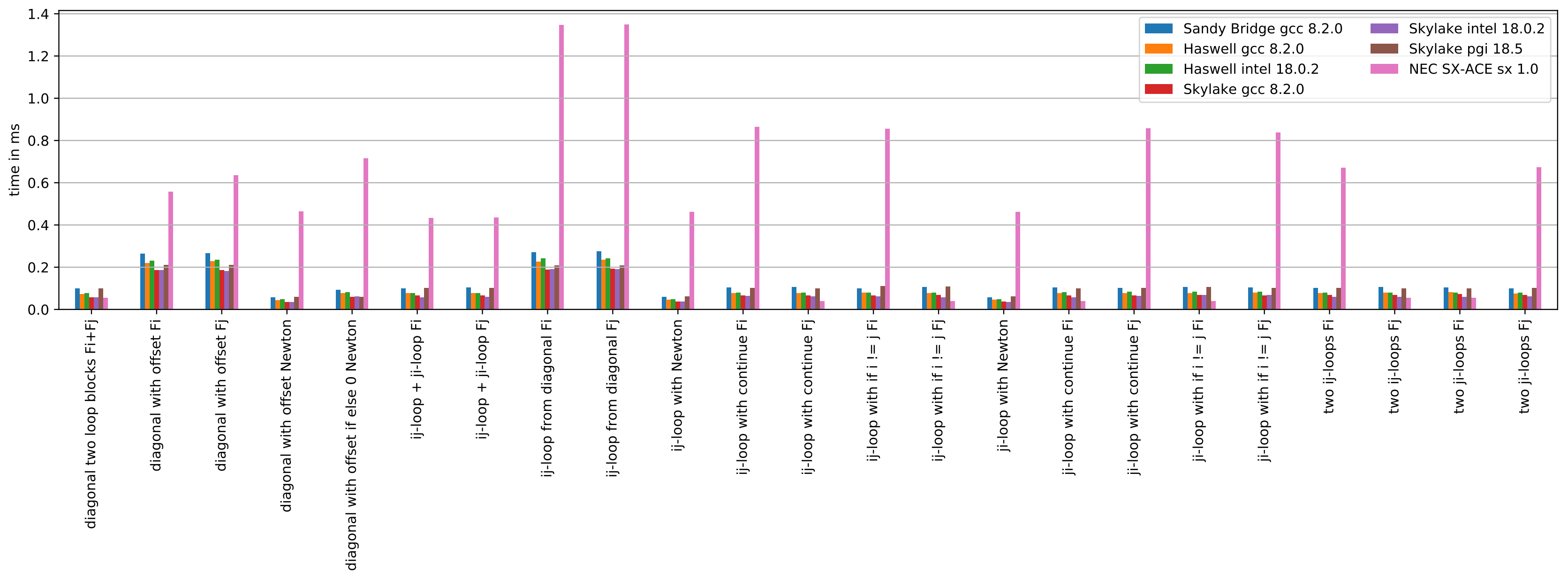
The times for the PGI compiler are overall much slower than the other compilers, even though its inline level was increased from the default 10 to 20 to help with the optimization of the C++ vectors. Without this adjustment, the used PGI compiler could not inline and optimize the standard vector element access operator `operator[]` at all.

The NEC SX-ACE results exhibit huge differences in performance across the algorithm implementations. The best algorithm implementations for the SX-ACE turn out to be the `ij and ji loop` based ones without Newton. These implementations are 35 times faster than the slowest implementation `ji loop from diagonal`. This can be explained by looking at the optimization report. It turns out that the SX compiler vectorized the fast versions, while was not able to vectorize the slow ones. Despite the SX-ACE low operational frequency and the disadvantages of the vector architecture for short loops, its per core performance is 14 % better than the Intel Haswell core. However, it is 10 % worse than the Intel Skylake core.

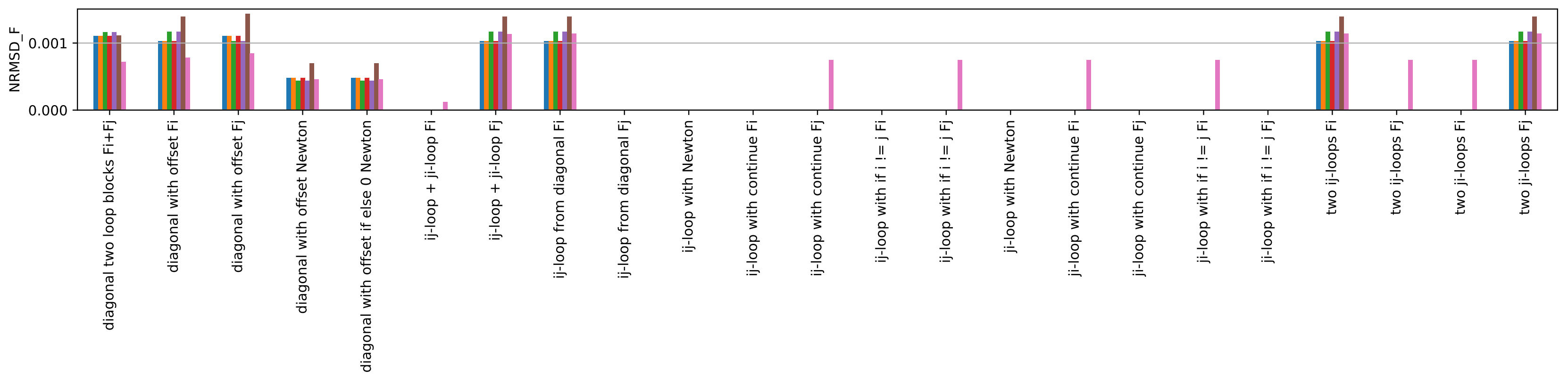
Looking at the NRMSD for the forces in Figure 4.7, around half of the algorithms produce slightly different results compared to the chosen reference. The NRMSD values are between 0.000 410 and 0.001 440. All the algorithms iterating diagonally through the force matrix show here numerical differences to the row wise iterating ij-loop with if `i!=j` reference implementation. Also, some algorithms iterating over the force matrix in two blocks deviate numerically. The SX-ACE results differ to the reference for the fastest implementations. However, the small differences are not critical for most MD simulations, as they use correction methods for small numerical rounding errors, e.g., in form of thermostats, anyway.

implementation	Sandy Bridge gcc 8.2.0	Haswell gcc 8.2.0	Haswell intel 18.0.2	Skylake gcc 8.2.0	Skylake intel 18.0.2	Skylake pgi 18.5	NEC SX-ACE sx 1.0
ij-loop with if i != j Fi	0.099	0.079	0.079	0.066	0.062	0.110	0.856
ij-loop with if i != j Fj	0.105	0.078	0.079	0.067	0.057	0.108	0.038
ji-loop with if i != j Fi	0.105	0.077	0.084	0.067	0.068	0.106	0.038
ji-loop with if i != j Fj	0.104	0.079	0.084	0.066	0.069	0.101	0.837
ij-loop with continue Fi	0.104	0.076	0.079	0.066	0.063	0.101	0.863
ij-loop with continue Fj	0.107	0.077	0.079	0.066	0.062	0.099	0.038
ji-loop with continue Fi	0.103	0.076	0.082	0.065	0.058	0.099	0.038
ji-loop with continue Fj	0.100	0.077	0.083	0.066	0.063	0.101	0.857
ij-loop with Newton	0.058	0.045	0.048	0.037	0.036	0.061	0.462
ji-loop with Newton	0.056	0.045	0.049	0.037	0.035	0.060	0.462
two ij-loops Fi	0.102	0.077	0.079	0.067	0.059	0.102	0.670
two ij-loops Fj	0.105	0.078	0.080	0.068	0.058	0.100	0.054
two ji-loops Fi	0.104	0.082	0.079	0.071	0.059	0.100	0.054
two ji-loops Fj	0.098	0.076	0.079	0.068	0.060	0.102	0.672
ij-loop + ji-loop Fi	0.100	0.076	0.077	0.066	0.057	0.101	0.432
ij-loop + ji-loop Fj	0.103	0.077	0.077	0.065	0.059	0.100	0.434
ij-loop from diagonal Fi	0.271	0.226	0.241	0.189	0.191	0.208	1.347
ij-loop from diagonal Fj	0.275	0.235	0.240	0.193	0.191	0.208	1.348
diagonal with offset Fi	0.264	0.219	0.231	0.185	0.185	0.211	0.557
diagonal with offset Fj	0.266	0.228	0.234	0.185	0.181	0.210	0.635
diagonal with offset Newton	0.056	0.044	0.047	0.035	0.035	0.060	0.464
diagonal with offset if else 0 Newton	0.093	0.076	0.082	0.060	0.061	0.060	0.715
diagonal two loop blocks Fi+Fj	0.099	0.072	0.077	0.058	0.056	0.100	0.054

Table 4.5: Execution times for the Naïve algorithm implementations on various platforms using different compilers for a scenario with $\rho = 0.9$, $r_c = 3.0$, and $N = 150$. Molecules are initialized on a lattice. Minimum times for each system are marked in grey. Times are in milliseconds and the average over 100 kernel invocations.



(a) Runtime



(b) NRMSD for the forces using ij-loop with if $i! = j$ as reference.

Figure 4.7: Comparison of naïve algorithm implementations on different platforms using different compilers for a scenario with $\rho = 0.9$, $r_\mathrm{c} = 3.0$, and $N = 150$ with molecules initialized on a lattice.

Section summary

In this section, different implementations for the naïve algorithm were evaluated with respect to their performance with small number of molecules. The obtained results show that there can be huge differences in the performance. The fastest implementation on x86 was found to be 5 times faster than the slowest one. On the NEC SX-ACE vector system, even a factor 35 was between the performance of the best and worst implementation. On cache based x86 systems, the `diagonal with offset Newton` implementation was found to be the best. However, the standard implementations with loops over the rows and columns of the force matrix were close. On vector systems, the row and column based algorithms with conditionals in the computation were the best. With respect to the numerical differences between the different implementations, the NRMSD was mostly below 0.1 %. Differences are mostly caused by the different directions to iterating over the force matrix or by splitting loops into two blocks.

4.3 Task graph based parallelization of the linked-cell algorithms

The number of cores in high performance computing (HPC) systems is steadily increasing and current HPC systems already provide millions of them. This leads to problems parallelizing with pure Message Passing Interface (MPI) due to memory and communication overheads. Therefore, shared memory parallelization is typically applied at the node level to decrease the number of MPI processes. The most common combination here is MPI + OpenMP. However, achieving a good parallelization with classical loop based OpenMP constructs is becoming increasingly difficult with the higher core counts of the nodes due to the limited number of sufficiently long loops in the codes. Task dependency based parallelization models as described in Section 2.2.3 provide a potential solution here.

In the following, the applicability of the task graph based programming model is studied for the parallelization of the linked-cell algorithm at the node level. The task graph based programming model presented in Section 2.2.3 provides an easy way to parallelize a serial program or to convert an already MPI parallel code into a hybrid MPI + threads code. Assuming infinite resources, the scalability of this model is limited by the critical path of the task directed acyclic graph (DAG). The maximal achievable speedup S of the program is determined by the length of the critical path t_{cp}, as the tasks in this path have to be executed in sequential order:

$$S = \frac{t_{\text{s}}}{t_{\text{cp}}} \tag{4.18}$$

where t_{s} is the sequential execution time of the program.

The SMPSs [75] and OmpSs [31] runtime systems implement the task graph based programming model using simple pragma based code annotations that identify functions as tasks, which may be run in parallel. For dependency tracking between the tasks they use the memory addresses of the specified input and output parameters. SMPSs supports C and Fortran, OmpSs C, C++, and Fortran. Listing 1 shows the basic syntax of the code annotations to be used in a C program with the SMPSs runtime. The annotations for OmpSs use the marker `omp` instead of `css`, but are identical otherwise.

```c
#pragma css task input(...) output(...) inout(...)
void function(...) { ... }
```

Listing 1: Syntax of the SMPSs code annotations for a C program.

4.3.1 Taskification approach

To parallelize the linked-cell algorithm used in MD codes with the SMPSs/OmpSs runtime, tasks have to be defined. The runtime of the tasks plays an important role for the performance as described in Section 2.2.3: Too short tasks lead to high overheads while too long tasks do not allow to extract enough parallelism by the runtime system. In the linked-cell algorithm, tasks can be selected at several levels. One potential, very fine-grained task level is the calculation of a single molecule pair interaction including the distance and potential calculation. The input and output parameters for these tasks are the molecule data of the involved molecules as well as the potential parameters.

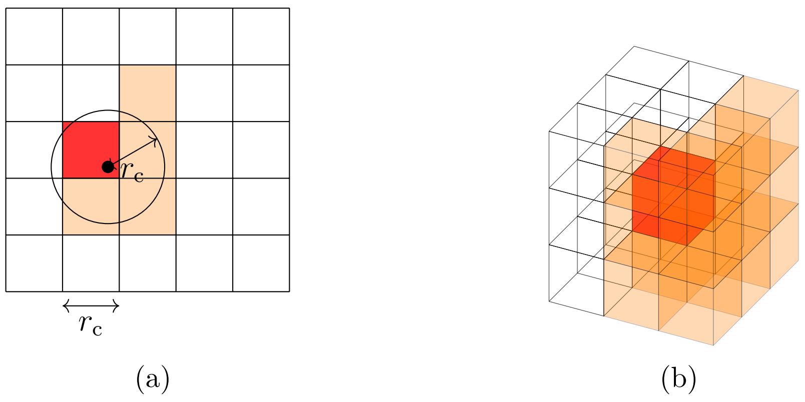


Figure 4.8: Stencils for 2D (a) and 3D (b) as they are used in the linked-cell algorithm with *actio est reactio*. The cell dimensions are equal to the cutoff radius r_c of the interaction potential. The red cell is the centre of the stencil, the orange cells are the neighbour cells. Red and orange cells are both updated.

However, a single pair interaction calculation takes around 10 ns on a recent Intel processor, as shown in Section 4.1.3. This is too short to be scheduled efficiently by the runtime systems, which require typically tasks of at least 250 µs to achieve good performance [75].

Therefore, longer running tasks are created by grouping together all computations related to interactions of molecules within one cell or between two neighbouring cells. In the following, those tasks will be referred to as cell interaction tasks. The input and output parameters of those cell interaction tasks are the molecules in the involved cells. For a typical cut-off of 5 and a density of 0.9, tasks at this level will include more than 100 molecules and take around 10 ms. This is enough for the efficient scheduling of the runtime system.

The link cell algorithm includes inter- and intra-cell calculations for the neighbour cells with a fixed stencil pattern. Therefore, the basic algorithm to be parallelized is a 9 or 27 point stencil operation for 2D or 3D, respectively. With the application of Newton's 3rd law, they reduce to a 5 or 14 point stencil as shown in Figure 4.8. The stencil is executed for each cell. To iterate over all cells, typically loops moving over the cells, first in x, then in y, and last in z direction are used. The same loop ordering is normally used for the inter-cell interactions in the stencil. This geometrically most appealing approach will be referred to as the naïve version in the following.

For the following considerations, it is assumed that all tasks have equal execution time. This simplification can be made, when the computational costs for the potential calculations are not much higher than the distance calculation, such as for the LJ potential. With this assumption, the critical path length can be given by the number of executed tasks along the path instead of the overall duration of their execution.

4.3.2 Benchmark implementation

The basic linked-cell algorithm with taskified cell interaction kernels was implemented in a benchmark for 2D and 3D using the SMPSs/OmpSs runtime [71]. The kernels consist of a busy loop to allow arbitrary execution times with a µs resolution. The kernels include also counters for correctness checks, which allow for the verification of the result against the original serial code version.

As the SMPSs/OmpSs programming model uses memory addresses to identify task dependencies, larger structured data are not supported directly as input or output parameters for a task. Thus, single memory addresses have to be used as a representative for larger structured data or arrays in these cases [75]. For the kernels, the memory address pointing to the cell data structure is used as a representative for each cell and its related data, i.e., all the molecules in the cell, here. Listing 2 shows the actual task definitions from the benchmark code.

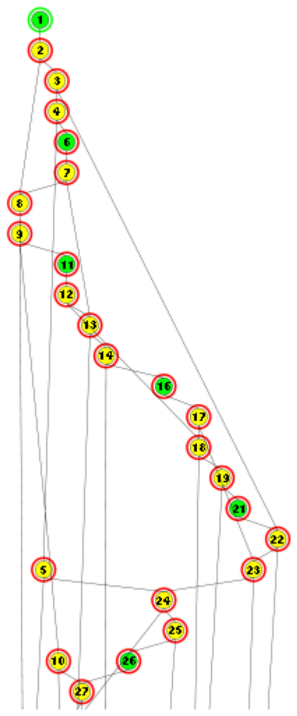
```
#pragma css task inout(cell)
void process_cell(Cell *cell);

#pragma css task inout(cell1, cell2)
void process_cell_pair(Cell *cell1, Cell *cell2);
```

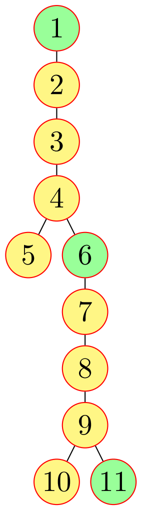
Listing 2: SMPSs task definitions in the link cell algorithm benchmark for intra- and inter-cell interaction calculations.

4.3.3 Task dependence analysis

Figure 4.9a shows the initial part of the task DAG, which was extracted from a benchmark run directly from the runtime using the Temanejo debugger [23]. In the task DAG, vertices represent tasks and arcs represent the dependencies between them. Some of the dependencies shown by Temanejo can be neglected as they are redundant with other, longer dependency chains. For example the dependence between task 3 and 22 is redundant to the dependency chain along tasks 4–6–7–13–14–16–17–18–19–21–22 and can therefore be omitted. Removing all redundant edges gives the so-called transitive reduction of the graph [6]. Taking into account the periodic boundary conditions, the basic structure of the transitive reduction of the benchmark's task DAG is shown in Figure 4.9b.



(a) Task DAG extracted during program execution with Temanejo



(b) Simplified Task DAG where redundant dependencies are removed

Figure 4.9: Task DAG for the basic 2D link cell algorithm with Newton's 3rd law using the naïve stencil implementation (see Figure 4.10a). Intra-cell interaction tasks are depicted in green, these for inter-cell in yellow.

From the transitive reduction of the Task DAG in Figure 4.9b, the critical path is much easier to identify. The critical path exhibits nearly sequential execution order of all tasks here, which limits the maximal achievable speedup. The origin for this graph structure lies in the order how the cell interactions are processed and thereby the inout dependencies are added to the graph. In the naïve linked-cell version, the stencil is moved in x direction over cells first and the interaction partners in the stencil itself are processes in a counter-clockwise-like way as shown in Figure 4.10a. Hereby, the dependency caused by the interaction between the central cell with its nearest neighbour in x direction is added second to last. This dependency is the critical one. Due to its inout character, it blocks the execution of all tasks for the stencil shifted in the x direction.

Fortunately, the ordering of the naïve linked-cell version is not the only possible one. In 2D there are four inter-cell interaction tasks to be computed for each stencil. Therefore, there is a total $4! = 24$ different permutations for the execution order of the tasks for this case. In the 3D case, there are 13! possible execution orders, respectively.

The maximal achievable speedup S for the different execution orders in the stencils can be expressed using three numbers: The number of total stencil executions k, the number of tasks in the stencil n, and the stencil displacement Δ, which is the number of cell-interactions to be calculated within the stencil before the computation of interactions for the next stencil can be started

$$S(\Delta, k, n) = \frac{nk}{n + \Delta(k-1)} \ . \tag{4.19}$$

In the limit $k \to \infty$ for an infinite number of cells, the speedup becomes

$$S_{\max}(\Delta, n) = \lim_{k \to \infty} S(\Delta, k, n) = \frac{n}{\Delta} \ . \tag{4.20}$$

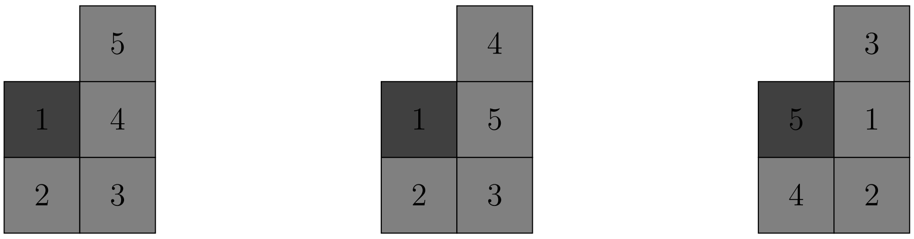
In 2D the stencil displacement Δ varies between 2 and 5 for the different execution orders of the inter-cell interaction tasks. Figure 4.10a and Figure 4.10b show possible task execution orders for the 2D-stencil and the corresponding achievable speedup according to Equation (4.20). The displacement becomes $\Delta = 2$, if the task computing interactions with the neighbour in x direction is computed directly after the computation of the intra-cell interaction task, and $\Delta = 5$, if it is the last interaction task being evaluated in the stencil. For the 3D case, Δ is within the interval $[2, 14]$, respectively. Therefore, in both cases the worst case is thus $\Delta = n$ and so by Equation (4.20) no speedup is achieved at all! The best case is $S_{\max}(2, 5) = 2.5$ for 2D and $S_{\max}(2, 14) = 7$ for 3D.

However, this can be improved further. If the task interaction with the neighbour in x direction is evaluated before the intra-cell interaction as shown in Figure 4.10c, then

$\Delta = 1$. The resulting task DAG then has the form of a pipelined execution and Equation (4.19) becomes the formula for the speedup of a pipeline

$$S = \frac{nk}{n + k - 1} \tag{4.21}$$

wherein n is the number of operations to perform and k is the pipeline depth. This improved version for the stencil implementation will achieve at most a speedup of k. In 2D this is $k = 5$, in 3D it is $k = 14$. Therefore, no suitable parallelization across an entire many-core node with more than 14 cores can be achieved in this way of parallelizing the link-cell algorithm.



(a) naïve: $S_{\max} = 1.25$ (b) worst case: $S_{\max} = 1$ (c) optimal stencil: $S_{\max} = 5$

Figure 4.10: Different possible execution orders for cell interactions in the 2D-stencil and their theoretical speedups

4.3.4 Task DAG optimization approaches

As shown, the straightforward taskified stencil implementation, with an outer loop over the cells and an inner loop over the stencil, runs into severe speedup limitations due to the inout-dependency sequence. To overcome these limitations, different approaches can be applied.

The first approach to improve the scalability of the task parallel version is to change the order of the loops. Instead of using an outer loop over all cells and an inner loop over the stencil, the loops are exchanged and therefore the outer loop iterates over the stencil and the inner loop over the cells. This reduces the effect of dependencies between the tasks dramatically and allows the parallel execution of one row for all cases, but the one of inter-cell interaction in the x direction. The latter and some more bad dependencies in the DAG can be circumvented by the addition of a colouring scheme with two-strided loops for the loops over all cells. This prevents blocking of

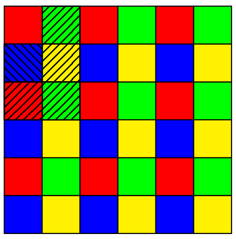


Figure 4.11: 2D colouring scheme for the loop exchange implementation of the link-cell algorithm: Executing computations for one stencil direction at a time for the cells of one colour does not introduce dependencies between tasks.

task execution due to dependencies in the forward cell interaction steps for each of the directions x, y and z.[2] Figure 4.11 shows an example of the applied colouring scheme in 2D.

A second approach to improve the scalability is task nesting. Here, the original loop over all intra-cell interactions is used, but instead of using a static loop creating the inter-cell interaction tasks, nesting is used: Each intra-cell interaction task creates an inter-cell task with one of its neighbours after it has completed its computation. Once this inter-cell task has finished its computation, it creates the next inter-cell task in the stencil. This is continued until all neighbours in the stencil are evaluated. With this approach, the dependencies in the DAG are introduced only when needed during execution and therefore reduces the number of dependencies at runtime, allowing more tasks to be executed in parallel. Unfortunately, this is currently only supported by OmpSs [32].

4.3.5 Evaluation

To evaluate the made theoretical predictions and the task DAG optimization strategies, the benchmark kernel introduced in Section 4.3.2 was used. It was modified accordingly for the different optimization approaches. Measurements were performed on the Laki and Hermit systems at HLRS using SMPSs 2.4 and the OmpSs version based on Mercurium 1.3.5.8. Laki is a cluster with 8 core dual socket Xeon X5560 nodes, Hermit a Cray XE6 system with 32 core dual socket Opteron 6276 nodes.

[2]Note that a simple colouring scheme without loop reordering cannot extract the same amount of parallelism.

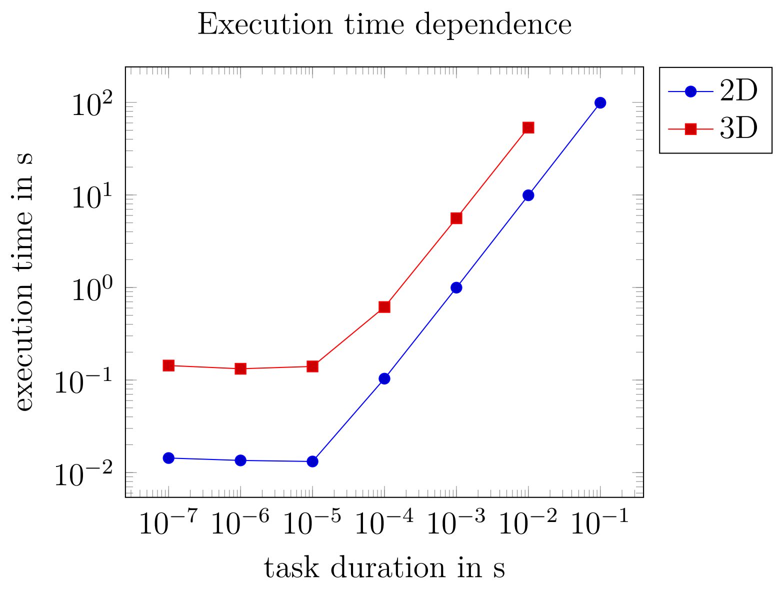


Figure 4.12: Effect of the single task duration on the overall program runtime using OmpSs

At first, the influence of the runtime scheduling system overhead is determined. This is necessary to ensure that the chosen task duration has no noticeable influence on the actual study of task dependency effects in the task DAG. Therefore, the execution time of the single tasks in the benchmark was varied from 100 ns to 100 ms for a run with 1 thread. The scenario consists out of 50×50 cells for the 2D and a $30 \times 10 \times 10$ cells for the 3D experiments. Having more cells along the x direction helps to reduce the influence of dependencies due to the periodic boundary conditions in the task DAG later.

The results in Figure 4.12 show a more or less constant execution time for task durations smaller than 10 µs. So the runtime overhead dominates here over the actual task execution time. For task durations above 100 µs, the execution time then is dominated by the task duration. This is in good agreement with the advice of the runtime developers that the task duration should be around or more than 250 µs [75].

Based on this, a task duration of 5 ms is selected for the 3D and 10 ms for the 2D in the following experiments. These values lie well within the region in which the runtime overhead can be neglected.

First, the taskified linked-cell implementation with different execution orders for the inter-cell interactions is tested. Out of all the possible orderings, one was selected for each displacement Δ between the optimal ($\Delta = 1$) and the worst ($\Delta = 5$) theoretical case. Figure 4.13 shows the achieved speedups for them in 2D using different number of threads. As can be seen, for every displacement the speedup increases nearly perfectly linear until it is limited from some number of threads on. The experimentally found limit for the speedup of each ordering is in excellent agreement with the theoretical value from Equation (4.19).

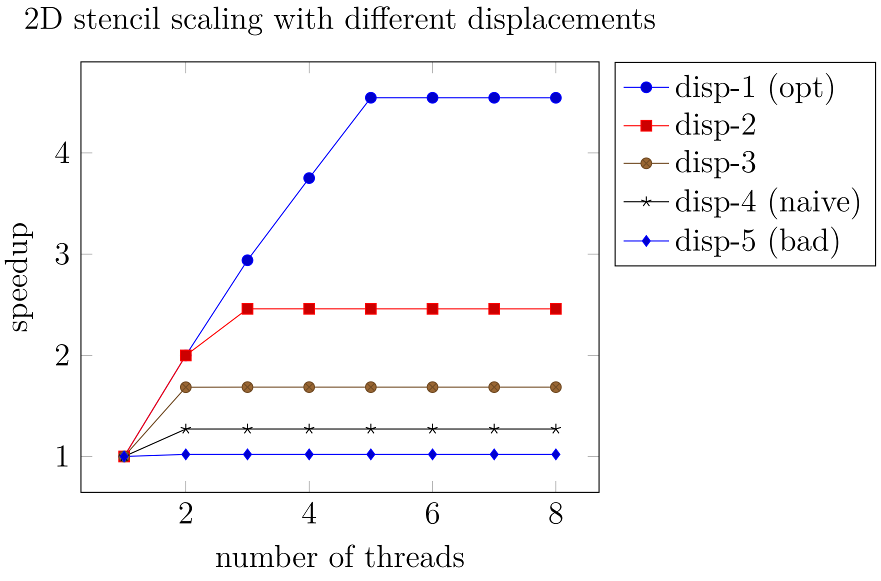


Figure 4.13: Scaling for different displacements in the 2D stencil obtained with the SMPSs-2.4 runtime on a single Laki node.

To evaluate the optimization strategies the loop exchange with colouring and nested versions have been implemented for the 2D and 3D stencil. The corresponding speedup results are shown in Figure 4.14.

Figure 4.14 shows the speedup results for the 3D case obtained on Hermit for the worst stencil, optimal stencil as well as the improved approaches loop reordering with colouring and nesting. Again perfect agreement with the predicted speedups for the stencil implementations can be seen. In contrast, the loop exchanged version shows perfect scaling up to the full node with 32 threads. Up to 23 threads nearly perfect scaling is achieved with the nested version. Above, the behaviour becomes jittery and is found to differ between different runs. The parallel efficiency is not perfect any more. The detailed cause for this is unknown, but is not found to be caused directly from the task DAG and seems to be related to the runtime implementation.

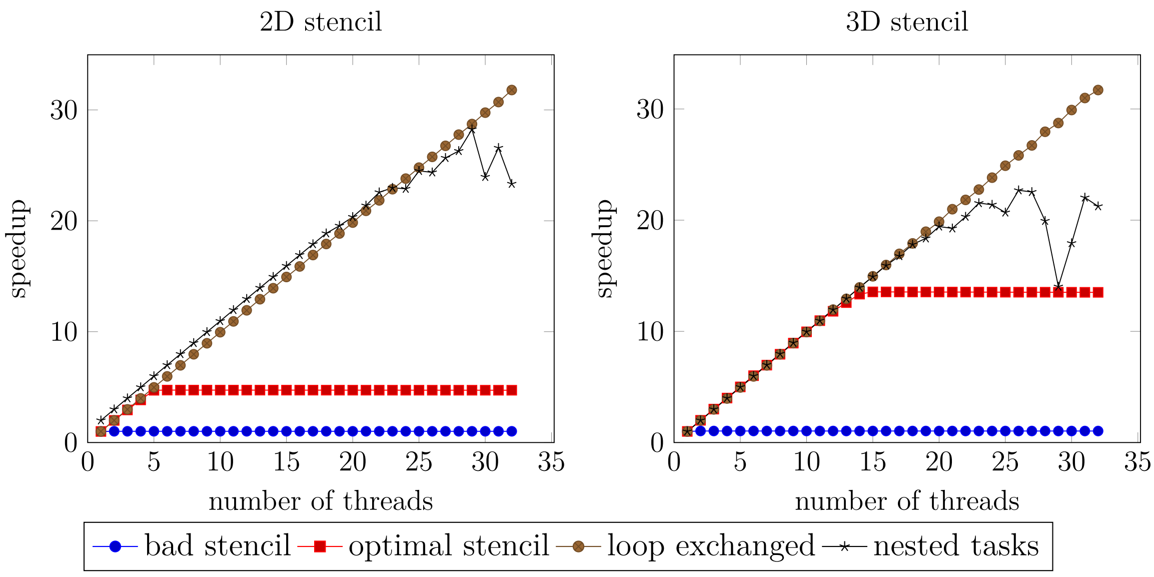


Figure 4.14: Scaling of the four different taskification approaches for the 2D and 3D stencil on a single Hermit node: serializing stencil (bad), optimal stencil (optimal), loop reordered (loop exchanged), and nested tasks.

Hence, it is not further analyzed in this work.

4.3.6 Section summary

In this section the applicability of the task graph based programming model to parallelize the link cell algorithm was studied. Therefore, a benchmark reproducing the algorithmic structure of the linked-cell algorithm was implemented and parallelized with the SMPSs and OmpSs runtime. It uses concurrent cell updates as it is found in the most common linked-cell implementation. The SPMSs/OMPSs tasks have been defined at the level of cell interactions. For this implementation, serialization effects were observed leading to a limited speedup. The cause for the serialization was identified as the order in which the tasks are created. Here, the intuitive order leads to a serializing dependency chain because of the inout nature of the task parameters. A detailed analysis of the dependencies resulted in a theoretical limit for the speedup of the basic parallelization approach, which was confirmed in experiments.

Two optimization approaches were made to overcome this limitation: a loop and colouring based version as well as a version using nested tasks. Both versions are not limited in their theoretical speedup as long as the number of tasks is large enough. The applicability of those two techniques was proven by experiments and the results showed their good scalability. The version breaking dependencies based on loop exchange and colouring showed perfect scaling. The task nesting based approach scaled well, but the task scheduling of the runtime showed effects for higher thread counts. However, the nesting based approach was found to be much simpler to implement as it required less code changes.

As a conclusion, the task graph based parallelization of the linked-cell algorithm showed that it can be applied very easily to a stencil based code selecting the right level of task granularity. However, care has to be taken when it comes to the resulting task dependencies, otherwise the achievable speedup may be limited. The obtained results were presented to the SPMSs/OMPSs runtime developers, who now added a `concurrent` statement in OMPSs specifically for such use cases.

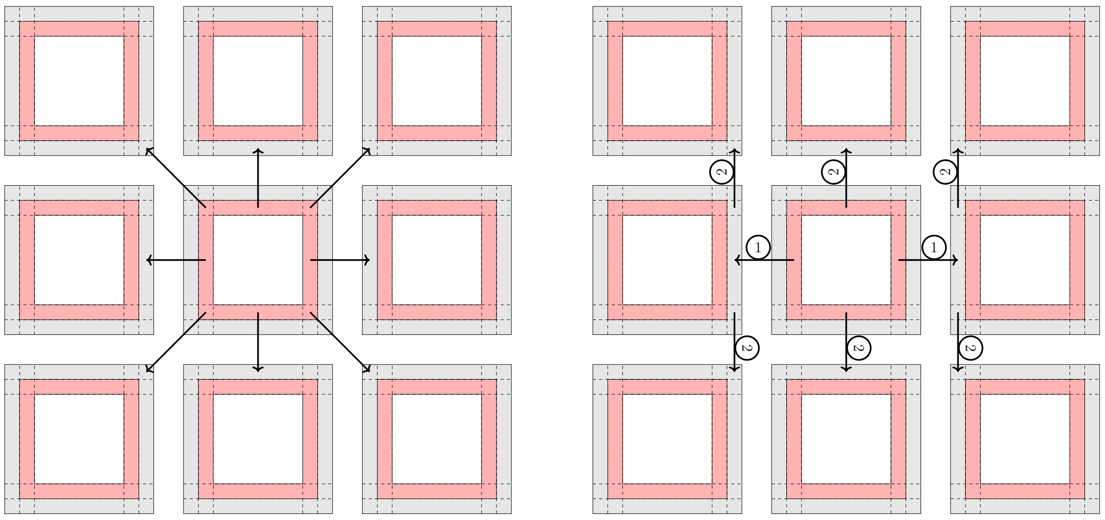
5 Inter node level performance engineering/analysis

The parallelization across multiple nodes of Molecular Dynamics (MD) codes for large systems and short range potentials is commonly based on domain decomposition. With the increasing number of nodes of HPC systems, the necessary degree of parallelism at the inter-node level is rising. Thus, the efficiency of the communication is becoming more important than it already is. While the domain decomposition approach is used since the beginnings of MD simulations, there are many implementation details. These details are mostly based around different communication patterns and the overlapping of communication and computation. In the following, these two aspects are investigated on the basis of a synthetic benchmark as well as the ls1-MarDyn simulation code.

5.1 Direct corner vs. indirect corner exchange

In a distributed application that uses domain decomposition for its parallelization, each process has to communicate halo information with neighbouring processes that are responsible for the adjacent volumes as described in Section 2.3.4. In the standard domain decomposition, where each process has 26 neighbours in 3D, two approaches to implement the communication are common: Direct communication with all neighbours and indirect corner communication via folding.

In the direct communication approach, each process exchanges necessary halo data directly with its neighbours using 26 send/recv operations. Instead, the indirect corner communication via the folding approach sends data only along the spatial axes in x, y, and z direction in a coordinated manner. Corner halo data are exchanged by flipping them over neighbour processes by adding them to the halo data for this direction. This reduces the number of send/recv operations to 6, but comes with the disadvantage of synchronization between the different directions x, y, and z as well as a slightly higher overall communication volume due to the repeated sending of the corners and edges. Both patterns are shown in Figure 5.1 for two dimensions.



(a) direct communication

(b) indirect corner communication via folding, first in x (1), then y (2) direction

Figure 5.1: Communication schemes used in the standard domain decomposition for the 2D case. Data from the border areas (red) of a process are transferred to the corresponding halo areas (grey) of the neighbouring processes.

5.1.1 Benchmark implementation

To test the efficiency of the direct and corner communication via a folding pattern, a benchmark was created. The benchmark is implemented as a C++ program and uses MPI point to point communication for the data transfer.

Each process is assumed to hold a cubic volume with a homogeneous particle density of $\rho = 1$. The diameter of the volume and the width of the halo can be changed. The amount of data to be communicated per molecule in the halo shall include values for position, velocity, orientation and angular momentum as well as a unique molecule ID and a component identifier. This results in the exchange of twelve double precision floating point and two 64 bit integer numbers per molecule, which corresponds to a total of 112 B to be transferred per molecule.

Communication buffers are allocated as contiguous memory and transferred as a single byte stream using the `MPI_CHAR` datatype. Note that a real application is likely to use a derived datatype constructed using vector and struct types. This is likely to introduce additional overhead compared to the benchmark implementation.

The two communication patterns are implemented on the basis of MPI's Cartesian communicator interface. The dimensions of the process grid are determined with the `MPI_Dims_create` function, which selects the grid dimensions to be as close to each other as possible. The Cartesian communicator is created using the `MPI_Cart_create` function. For both patterns a blocking and a non-blocking version is implemented.

The communication pattern details of the implementations using MPI are shown in Figure 5.2. The actual data transfer for the patterns is performed with `MPI_Sendrecv` for the blocking and `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` for the non-blocking versions. In the non-blocking case, for the direct communication, all communication requests are synchronized at the end of an iteration, for the indirect corner communication scheme, communication is synchronized for each spatial direction within the iteration.

Each communication pattern is repeated several times and the overall time for all iterations is reported for each rank.

5.1.2 Benchmark results

With the benchmark, a weak scaling experiment was performed on the HLRS Hazel-Hen system. For the experiment, different numbers of particles per process were used, including typical values for large simulations of $20^3 = 8000$ and $50^3 = 125\,000$ particles. Scaling up to 2048 nodes / 49 152 cores of Hazel Hen, this corresponds to scenarios with up to 393.2 million and 6.144 billion particles. The halo width was chosen to be 5, matching the commonly used cut-off radius $r_c = 5$.

```
for( neighbour in neighbours) {
  MPI_Sendrecv( ... );
}
```

(a) direct, blocking communication

```
for( neighbour in neighbours) {
  MPI_Irecv( ... );
  MPI_Isend( ... );
}
MPI_Waitall( ... )
```

(b) direct, non-blocking communication

```
for( dim in {x,y,z} ) {
  for( disp in {-1, +1} ) {
    MPI_Sendrecv( ... );
  }
}
```

(c) indirect, blocking communication

```
for( dim in {x,y,z} ) {
  for( disp in {-1, +1} ) {
    MPI_Irecv( ... );
    MPI_Isend( ... );
  }
  MPI_Waitall( ... )
}
```

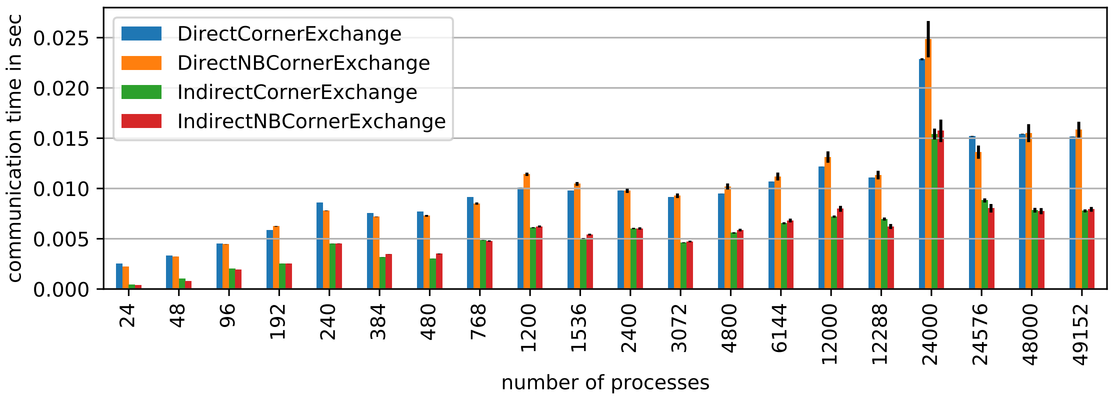
(d) indirect, non-blocking communication

Figure 5.2: MPI call sequences of the communication patterns in the neighbour communication benchmark

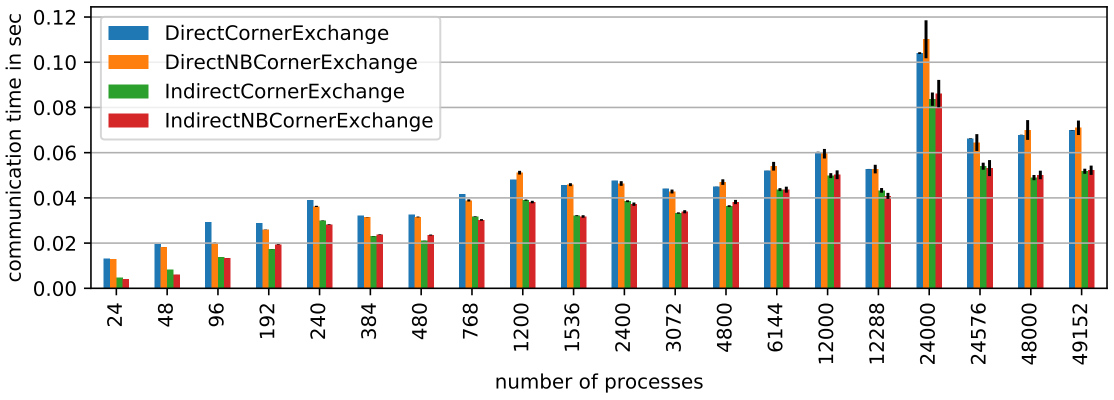
Figure 5.3 shows the average communication time for one iteration. As can be seen, the indirect corner communication based approach is always faster than the direct communication approach. The blocking and non-blocking versions of a pattern are mostly of comparable performance within the error margins, even though the non-blocking versions seem overall a bit slower. This is expected, as there is no potential for overlapping of communication and computation due to the lack of computation in this implementation of the benchmark. So, the pattern follows exactly the self-consistent MPI performance requirements [90].

Overall, the communication time increases with the number of cores. Comparing the time of a single node and the largest runs with 2048 nodes, the time increases for the direct communication pattern by a factor of roughly 6 and 20 for 20^3 and 50^3 molecules per process, respectively. For the indirect pattern, time increases by a factor of roughly 20 and 11 for 20^3 and 50^3 molecules per process, respectively.

The main cause for this behaviour lies in the inter-node communication. Figure 5.4 shows the message distribution for inter-node, inter-socket, and intra-socket messages for the two approaches. As can be seen, the fraction of slower inter-node messages increases with more cores. Consequently, communication time increases for both



(a) 20^3 molecules per process

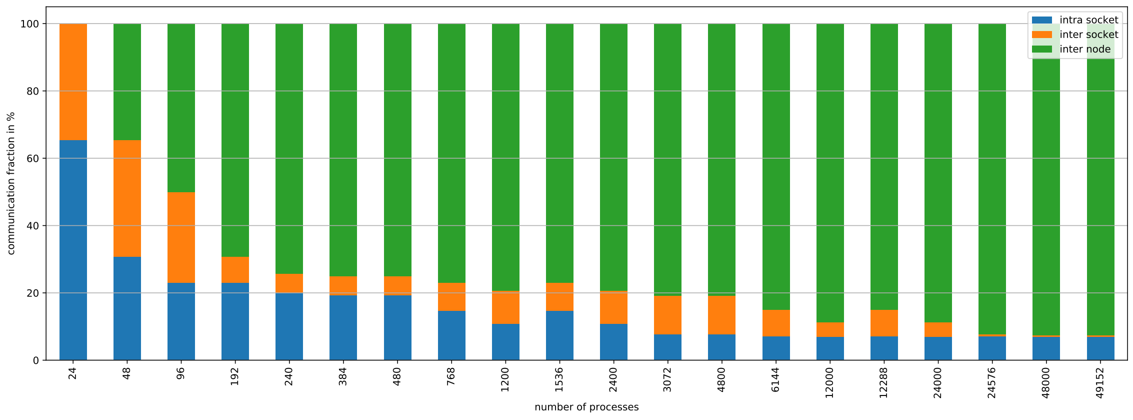


(b) 50^3 molecules per process

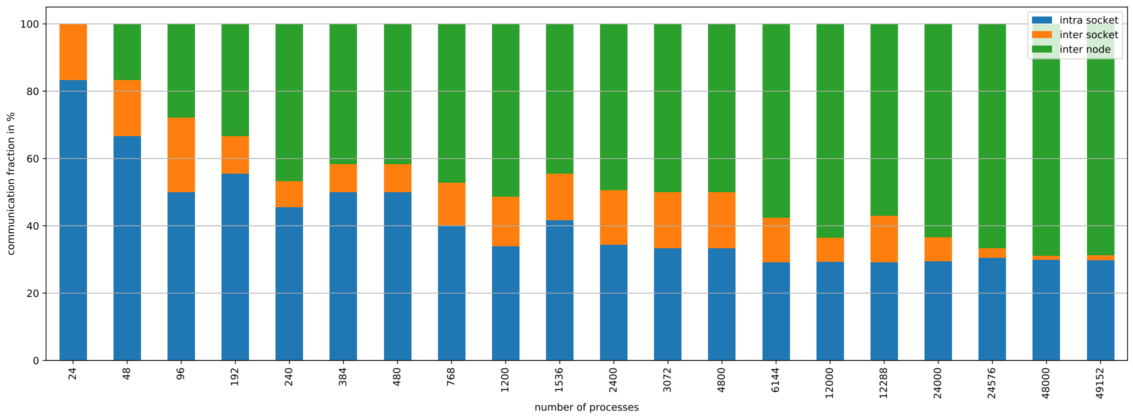
Figure 5.3: Weak scaling experiment of communication time for the direct communication and indirect corner communication patterns using blocking and non-blocking MPI communication from the neighbour communication benchmark on HLRS HazelHen for two different per process scenario sizes. Times are the average halo exchange time over 100 iterations. Black lines mark the variance across multiple measurements based on min and max times. The used halo width is 5 molecules.

approaches with larger number of processes.

Further, it can be seen from Figure 5.4 that for runs with more than 2400 nodes, the number of inter-node messages in the direct communication approach accounts for more than 92 % of the total message count. In contrast, in the indirect corner communication approach, the inter-node messages make up only a fraction of around 70 % here. So, despite the overall increased communication volume, the indirect corner communication via folding is faster than the direct approach, as the faster intra-socket communication overcompensates for this. This advantage is even more pronounced for the smaller per process volume case. Here, the communication volume along edges is smaller and therefore latency plays a more important role, which is even better for intra-node communication.



(a) direct



(b) indirect corner communication via folding

Figure 5.4: Message distribution for inter-node, inter-socket and intra-socket communication for two communication approaches

5.2 Overlapping halo communication in ls1-MarDyn

As shown in the previous section, the communication pattern for the indirect corner communication via folding is still favourable for the particle exchange in domain decomposition based MD simulations on current HPC systems. However, no clear result for the choice between blocking and non-blocking communication could be given with the simple synthetic benchmark. So, the use of non-blocking communication with MPI and potential overlapping opportunities will be studied in the following based on the real world code ls1-MarDyn MD.

5.2.1 Particle exchange implementation in ls1-MarDyn

The original ls1-MarDyn code uses the indirect corner communication pattern in the version with two consecutive blocking `MPI_Sendrecv` calls for each exchange direction. In contrast to the neighbour communication benchmark, ls1-MarDyn exhibits potential for overlapping of communication and computation. There are two main parts in a simulation step, which can be overlapped: The computation of all inner molecule interactions and the preparation of the send buffers after updating molecule properties. Due to the structure of the code, the first approach is left out because this requires a major rewrite of the computational kernels. So, the second approach will be studied in the following.

The ls1-MarDyn code implements for every folding step the following substeps for the particle exchange in order: (1) Allocation of send buffers of appropriate size, (2) packing of particles to be exchanged from the data structure used during the force calculation into the send buffers, (3) exchanging the number of particles to be exchanged with the other process via `MPI_Sendrecv` , (4) allocation of receive buffers with appropriate size based on the exchanged molecule count, (5) sending/receiving the actual particle data via `MPI_Sendrecv` , and (6) unpacking the received particles into the data structures used for the force calculation and freeing the buffers.

While this implementation is straightforward, it has two major drawbacks: First, it requires two communication steps, one to exchange the buffer sizes and one to transmit the actual data. Second, the used blocking MPI calls do not allow to overlap communication and computation and are therefore also not capable to hide load balancing issues, if the numbers of particles to be sent in the different directions x, y, and z differ.

5.2.2 Alternative implementations

In the following, three alternative implementations for the particle exchange in ls1-MarDyn are described: Isend-probe-recv, Isend-probe-irecv, and Out-of-order.

Isend-probe-recv

The first drawback—the separate communication of the number of sent molecules and actual data—can be omitted by using `MPI_Probe` and `MPI_Get_count` , which allow to extract the number of received data elements from the message header. So, no extra communication step is needed to obtain the appropriate size for the receive buffer. To prevent deadlocks, the `MPI_Isend` operation is used. The exchange steps become then `MPI_Isend` , `MPI_Probe` , `MPI_Get_count` , allocate memory, `MPI_Recv` .

Isend-probe-irecv

The second issue—overlapping of communication and computation—can be addressed by the use of non blocking operations for the receive step. So, the first send and receive in each direction x, y and z can be overlapped with the second send, receive buffer allocation and receive. The necessary steps for the communication with one neighbour are `MPI_Isend` , `MPI_Probe` , `MPI_Get_count` , allocate memory, `MPI_Irecv` . These steps are repeated for the second neighbour in the same direction and is followed by `MPI_Wait` to synchronize the exchanges before unpacking the received data.

Out-of-order

Additional potential lies in the improvement of load balancing for the message exchange. If the second message in a direction is smaller and therefore arrives earlier than the first one at the destination, the unpacking order can be swapped. So the unpacking of the earlier message can be overlapped with the receive-operation for the other message. This out of order approach is implemented by checking the message status with `MPI_Test` . If a message is received, it starts immediately with the unpacking and insertion of data into the data structures used for the computational kernel. The steps to perform here start with buffer packing and `MPI_Isend` for both neighbours in a direction. This is followed by `MPI_Iprobe` , `MPI_Get_count` , receive buffer allocation, and the posting of the receives with `MPI_Irecv` . After this, the processes check for the first message to arrive via `MPI_Test` .

5.2.3 Implementation performance comparison and summary

In the following, a comparison of the performance for the four different implementations in ls1-MarDyn is presented. The test scenario is in this case a homogeneous Lennard-Jones (LJ) system with reduced temperature $T = 0.95$ and reduced density $\rho = 0.6223$ consisting of 5 million molecules. This is a typical problem to be computed with the standard domain decomposition approach. The test is run with 1200 processes on HazelHen. The simulation is executed for 1000 time steps, to obtain statistically sound data for the communication costs.

The results are shown in Figure 5.5. The three alternative implementations prove all to be better than the original send-recv based version. The best one of them is the out-of-order implementation, which saves more than 8 % of communication time for this scenario. So, the communication time per simulation step goes down from 14.7 ms for the original send-recv to 13.5 ms for the out-of-order implementation. The standard deviations for all reported times is around 0.15 ms.

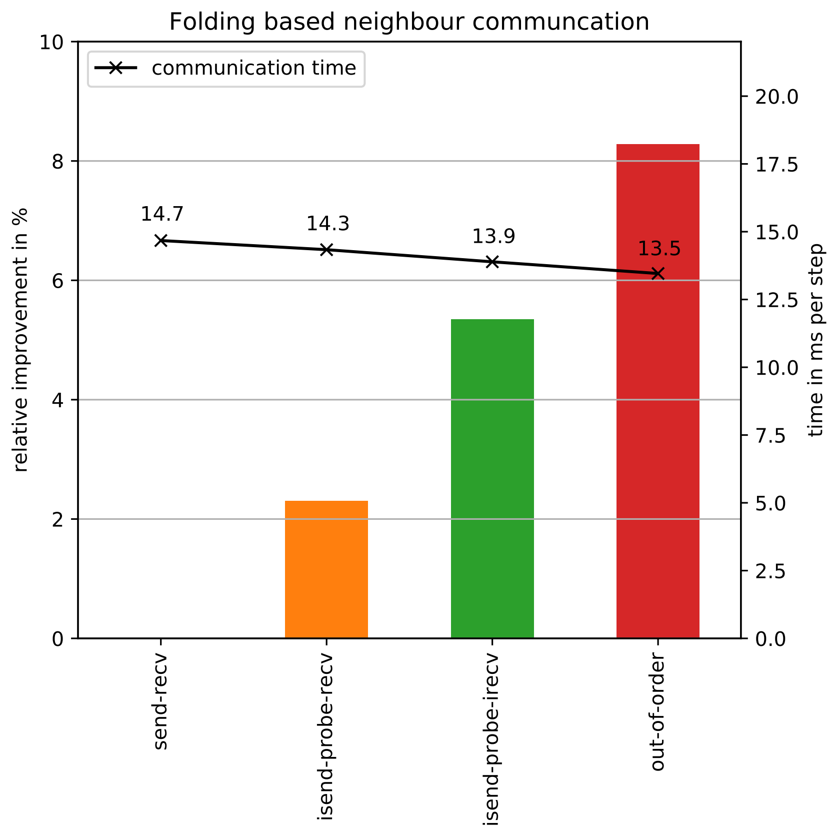


Figure 5.5: Average communication times and relative improvement of different implementations for the folding based neighbour communication in ls1-MarDyn. Bars show the performance improvement in comparison to the original send-recv implementation. The black line marks the average communication time per time step.

6 Fault tolerant MD

The Molecular Dynamics (MD) simulations in this work target large systems and long time spans to be simulated. These simulations require extensive compute resources to handle the large systems and long runtimes for the high number of necessary time integration steps. With simulations using systems with hundreds of thousands of CPUs, the mean time between failure (MTBF) is expected to drop to the order of one day as described in Section 2.5. Therefore, it is important that MD simulations can cope with the problem of system failures.

This section starts with a short system failure study for a current HPC system. Then resilience for MD codes is addressed. As the classical Checkpoint/restart (C/R) approach becomes inefficient for large systems [29], an Application-based fault tolerance (ABFT) approach for MD simulations is developed in this part of the work. An implementation of the approach is realized in ls1-MarDyn, which is then used to evaluate the approach with a real world scenario.

6.1 System failure survey

In order to handle fault tolerance efficiently, it is important to characterize the frequency of failures as well as the hardware levels at which they occur. Therefore, an initial overview of the current failures in actual HPC systems is obtained using available data for the Hornet/Hazel Hen systems at HLRS in the time between November 2014 and April 2019. A history of recorded hardware replacement events in each month is shown in Figure 6.1. As can be seen, there is some variance in the number of events between months, but no general trend of decrease or increase in failure rate for this time period. In total, 1665 hardware replacement events were recorded that were related to faulty behaviour of system components and overall 2261 hardware parts had to be replaced. This corresponds to an MTBF of 24.9 hours for the

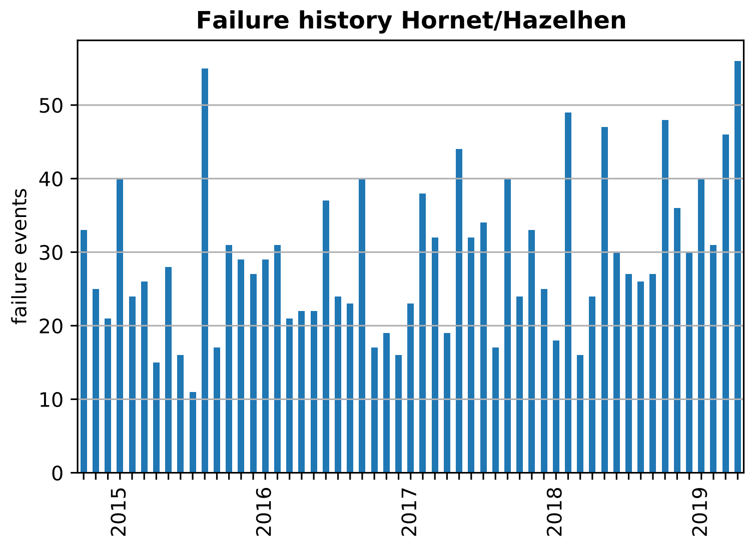


Figure 6.1: Number of hardware replacement events per month for Hornet/Hazel Hen in the time from November 2014 to April 2019.

system and an MTBF of 17.7 hours for hardware parts. These values confirm the estimated MTBF for current HPC systems of 24 hours described in Section 2.5 very well.

Not all the recorded hardware issues resulted in an abnormal abortion of jobs with data loss in running applications. For example, power supplies are multiple times redundant in the system, so the job can in most cases finish, and the failed power supply can be replaced later during a maintenance window. So, it is of interest to know which parts of the system are affected the most.

Figure 6.2 shows the distribution of failed components across the different parts of the system. Obviously, half of the hardware replacements were related to the memory modules (RAM). Failures in the memory system affect in most cases a running job and will result in crashes and data loss or silent data corruption in a node. The same applies for the less frequent problems with CPUs. The 26 percent of failures caused by power supplies are of less importance as stated above. Network, board, and other failures that can bring down multiple nodes or the entire system at a time are below 10 percent. So, it is important for applications to focus on resilience at the single node level in first place.

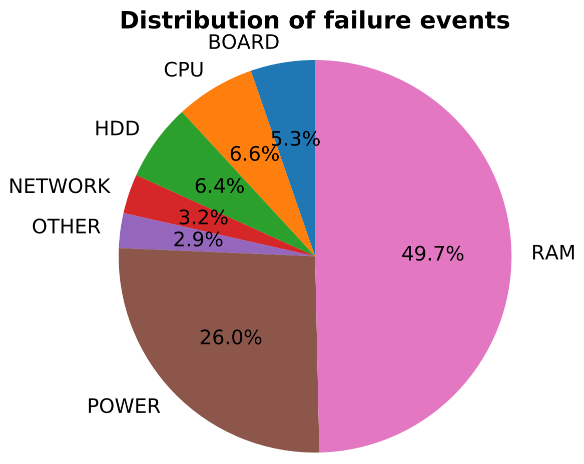


Figure 6.2: Origin of failure events on Hornet/Hazel Hen by categories. The category "other" includes failures in the cooling system. Values are based on recorded hardware replacements in the time from November 2014 to April 2019.

6.2 Failure scenario

Today's HPC system architecture is based on many shared-memory nodes. Large simulations, as the targeted MD simulations, run distributed across many of these nodes via MPI. A typical setup here is to use one MPI process per node and to use shared memory parallelization, e.g., OpenMP, inside the node. As seen in the previous failure study, the normal failure scenario is the loss of such a compute-node. For the typical setup, this means the loss of one MPI processes with all its related data.

So, for the study in this work, the following failure scenario was assumed: A simulation is running with N processes P_i. At some point of the simulation process P_X fails. The failure causes all local data of process P_X to be lost. No more than one failure is assumed to happen at the same time. However, multiple failures can occur at different times within one simulation run. After failure, the system provides a new process P'_X, e.g., from a pool of spare processes, which takes over the place of the failed one. Then the recovery mechanism for the simulation is started. The failure scenario is depicted in Figure 6.3 for a 2D simulation.

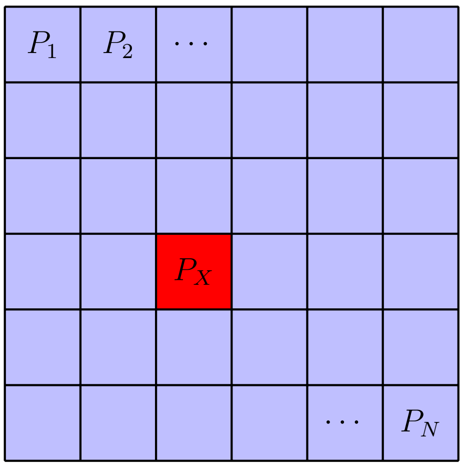


Figure 6.3: 2D simulation failure scenario: The 2D data are distributed via a regular domain decomposition over N processes P_i. Process P_X fails and all its local data for the subdomain are lost. The failed process P_X will be replaced by a new process P'_X.

6.3 Current fault tolerance approach of ls1-MarDyn

The current approach of failure handling in ls1-MarDyn implements a standard application level C/R scheme, which is common practice also in other MD codes such as LAMMPS [81]. The checkpoint interval can be chosen via the configuration file. Phase space data are written out after the specified number of timesteps at the end of a timestep. Checkpoints can be written either as an ASCII file or using a binary file format.

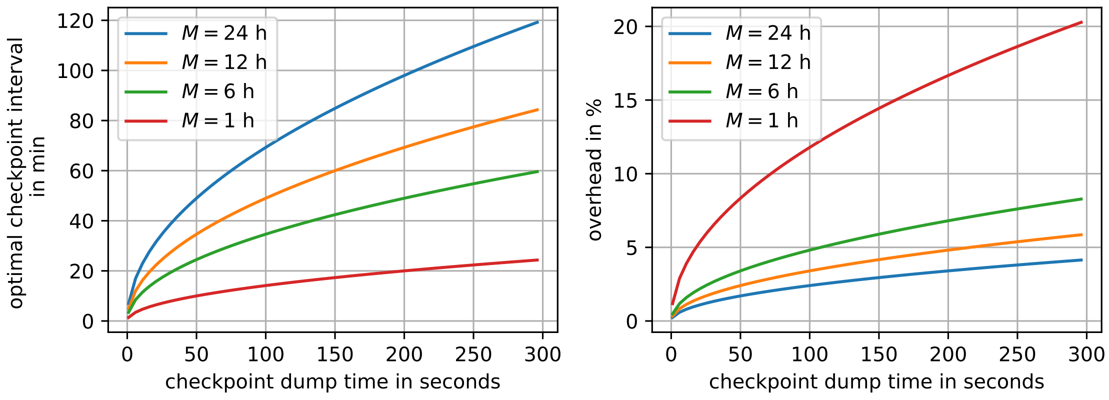


Figure 6.4: Optimal checkpoint interval and corresponding checkpointing overhead for dump time $\delta = 100\,\mathrm{s}$, restart time $R = 100\,\mathrm{s}$, and different MTBF values based on Equation (2.11).

The analysis and optimization of the I/O part of ls1-MarDyn was already presented in Section 3.1. From there the typical I/O time for large multi-million molecule scenarios is on the order of 100 seconds. Figure 6.4 shows the optimal checkpoint interval for different MTBF values based on Equation (2.11) assuming an average checkpoint dump time $\delta = 100\,\mathrm{s}$ and restart time $R = 100\,\mathrm{s}$. Using the current MTBF of $M = 24\,\mathrm{h}$, the optimal checkpoint time becomes $\tau_{\mathrm{opt}} = 69\,\mathrm{min}$ and the overhead for checkpoint writing is 2.4 %. While this might still be acceptable for the moment, it gets worse with decreasing MTBF. So, for an MTBF of 1 hour, the optimal checkpoint interval becomes 14 min and the overhead increases to 12 %. Hence, there is a clear necessity for improvement targeting large MD simulations on future HPC systems.

6.4 A new resilience approach for MD

6.4.1 Concept

The execution time model of Daly, outlined in Section 2.5.3, takes into account several overheads and models them with several parameters. Two of the most important parameters in it, are the dump time δ, that represents the time to write necessary restart data to persistent storage, and the restart time R. To increase the efficiency of a fault tolerant application, the dump time and the restart time have to be minimized. Both goals can be achieved by shrinking the restart dump size, i.e., the amount of checkpoint data.

In the failure case, the classical C/R approach restores all processes to the state of the last checkpoint and re-computes all simulation steps from thereon. However, in the studied failure scenario for MD simulations, not all data are lost from the last computed time step: Only simulation data residing in the failed component are lost, data on the other components are still present and usable.

One idea to take advantage of this fact, is to use all available compute resources after failure to speed up the re-computation of the lost part. However, to exactly re-compute the lost data in an MD simulation, data from intermediate steps between the moment of the checkpoint dump and the failure event are necessary, but these are not kept during the simulation, as this would obviously equal a per step checkpoint. So this idea does not work.

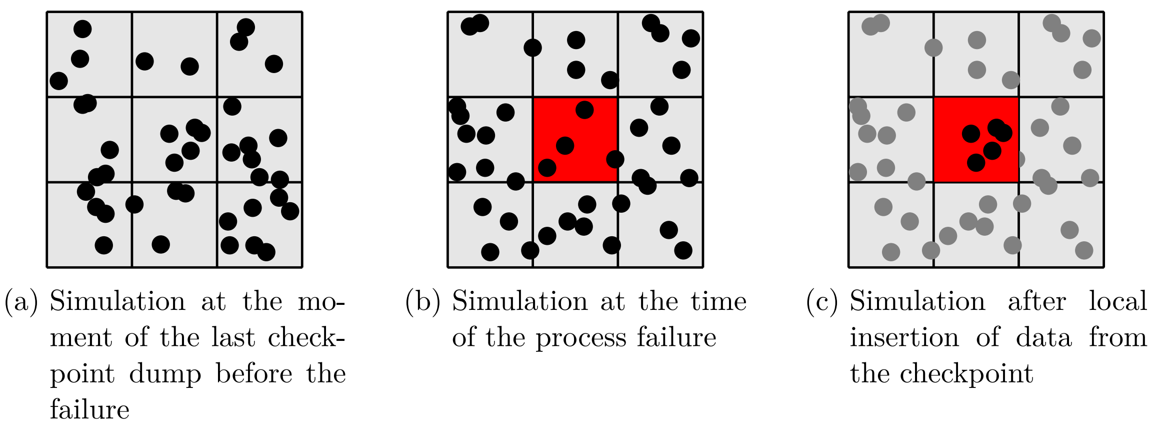
However, one can take another approach, which takes advantage of the not-lost data. It is based on the idea to omit the expensive exact re-computation by creating approximate local data that fit the lost ones as good as possible. The simulation shall then continue after the insertion of the approximate local data. For this approach to work, the simulation must be stable under a small perturbation, allowing to converge towards the same result at the end. MD simulations are based on this assumption already, as they start from an inexact physical state, which undergoes an initial equilibration phase before collecting results in a following production phase. So, the problem of exact re-computation is now transformed into the problem of finding good approximate data that keep the disturbance of the simulated system as small as possible and then applying a short re-equilibration phase before continuing with the production phase.

For the first evaluation of this approach, the following techniques to replace the lost local data are studied in the following: (1) The reuse of historic data without re-computation for the intermediate simulation, (2) on the fly regeneration of data based on local properties collected and saved during the simulation, e.g., the number of particles and temperature within the subdomains of the processes.

6.4.2 Historic data

The historic data strategy is based on the C/R approach. The classical C/R approach performs a full rollback recovery from a previously saved checkpoint. In contrast, the historic data approach replaces only the missing data of the failed processes with checkpoint data while keeping all other current data and continues the computation without performing a rollback, i.e., re-computing the lost steps.

Figure 6.5 shows the essential parts based on a 2D example with 9 processes: The phase space data are stored at point 6.5a. Then at point 6.5b the data for the central subvolume are lost due to a failure. Data from 6.5a are then inserted as a replacement, while the rest of the data are kept and the simulation is continued from there on. The corresponding state after the recovery is shown in 6.5c.


(a) Simulation at the moment of the last checkpoint dump before the failure

(b) Simulation at the time of the process failure

(c) Simulation after local insertion of data from the checkpoint

Figure 6.5: 2D example of the new resilience strategy using the historic data approach. The simulation uses a domain decomposition with 9 processes. The red area in the centre marks the subvolume lost due to a failure. Gray points where not subject to the failure and therefore kept as is.

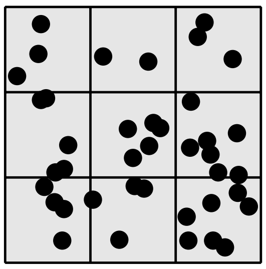
This approach requires some care with molecules close to the subvolume boundary of the lost part. So, the insertion of the old data may result in molecules being very close to each other at the boundary between the processes' subdomains resulting in very high molecule accelerations. This can be addressed to some degree by using a thermostat that takes care of such cases by limiting the maximal acceleration for the affected molecules.

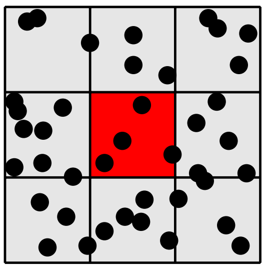
Also duplication or loss of molecules may occur with this approach in cases where a molecule crossed a process's subdomain boundary between the last checkpoint and the failure. Here it depends on the scenario if the global or local increase of the number of molecules is important. However, this is not an issue in most cases for the here targeted large systems as will be shown in the evaluation part.

6.4.3 Data regeneration

The second approach to create approximate local phase space data for the lost subvolume is the generation from ensemble values. Well suited values here are the number of particles N and temperature T within each subvolume. These two values are normally already known or computed in every MD simulation step and therefore do not require additional computational overhead. Also, having only two values to be stored reduces the amount of necessary restart data drastically—from $\mathcal{O}(N)$ for a full checkpoint to $\mathcal{O}(1)$. These two facts together now allow to save restart data per simulation step.

The corresponding steps for the on the data regeneration approach are shown exemplarily in Figure 6.6: The number of particles is saved in 6.6a—here $N = 5$ for the central subvolume. The simulation continues until the failure occurs in 6.6b. Now the lost data are replaced by inserting the appropriate number of particles—here 5. The insertion follows the same method as for the creation of an initial state with a face centered cubic (fcc) lattice, resulting in the system shown in 6.6c. The simulation continues then from there on.


(a) Simulation at the moment of the last restore data collection point before the failure


(b) Simulation at the time of the process failure

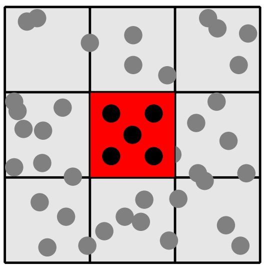

(c) Simulation after local insertion of generated data

Figure 6.6: 2D example of the new resilience strategy using the on the regeneration approach. The simulation uses a domain decomposition with 9 processes. The red area in the centre marks the subvolume lost due to a failure. Gray points where not subject to the failure and therefore kept as is.

This approach has one downside: The inserted data are not equilibrated and are likely to need some more re-equilibration time. However, there are several advantages that make up for this: The possibility to store restart data per simulation step helps to reduce errors made in the creation of approximate data for the lost ones. For example, with per simulation step coverage, the number of restored molecules will always be correct. Further, the regeneration approach automatically ensures the appropriate distances between the inserted molecules at the boundaries preventing the problems of too high particle accelerations from the historic data approach.

The biggest advantage of this approach is the minimal amount of data to be stored. While this itself reduces the dump time to persistent storage already, it now allows also In-Memory Checkpointing for large scenarios. Here, full In-Memory checkpoints waste substantial amounts of the memory that are necessary for computations, as at least two checkpoints have to be present at the moment of checkpoint creation [77]. In contrast, the here presented approach requires only to store a few double precision floating point values and thus does not require as much memory. The data can therefore also be distributed easily to multiple processes on other nodes, reducing potential problems with successive failures of nodes that keep necessary recovery data. This helps to mitigate the hard problem of optimized checkpoint placement in future exascale systems [52].

The recovery data transfer to other nodes in this approach can, e.g., make use of the existing neighbour communication infrastructure from the domain decomposition in MD codes, by appending the additional values to the per step particle exchange. This also helps this approach to keep overheads low.

6.5 Implementation in ls1-MarDyn

In order to test the proposed ABFT method, it was implemented into ls1-MarDyn. Two main functionalities have to be provided: failure detection, and process and communication recovery. These are tightly connected to the used parallel runtime system. The parallelization of ls1-MarDyn is based on the Message Passing Interface (MPI). Unfortunately, the current MPI standard is not prepared for continuation of a parallel program subject to failures. The default error handler in MPI is `MPI_ERRORS_ARE_FATAL`[1] that will terminate the application in case of errors immediately and the alternative `MPI_ERRORS_RETURN` leaves the problem of an unrecoverable `MPI_COMM_WORLD` communicator. There are discussions over the past years about the addition of an MPI fault tolerance interface on the basis of the User Level Failure Mitigation (ULFM) proposal [4, 18]. However, the usage of ULFM for large MD applications has been shown to be extremely difficult and the application of it was limited to programs with a small code base so far [58]. So, to work around this problem, the essential failure detection and recovery parts have to be built inside ls1-MarDyn for now.

[1]except for I/O operations

A general problem for testing the new ABFT approach is the relatively low frequency and unpredictability of encountered failures in real systems. This would make a systematic study of the developed ABFT approaches challenging and time-consuming. Therefore, it is necessary to have a controlled way to generate and execute well-defined failure scenarios. So, failure simulation capabilities have to be an essential part of the implementation.

In order to address these points, a generalized failure infrastructure is introduced into the ls1-MarDyn code. This infrastructure handles all fault tolerance related tasks. Its interfaces are designed in a way that allows to use different external fault tolerance systems, including an upcoming fault tolerant MPI, as well as other failure detection tools. To achieve the necessary flexibility, the fault tolerance infrastructure is divided into two parts: failure detection and failure handling as shown in Figure 6.7. Both parts use a plugin architecture for easy extendability. The details of the different parts and plugins implemented for this work are described in the following.

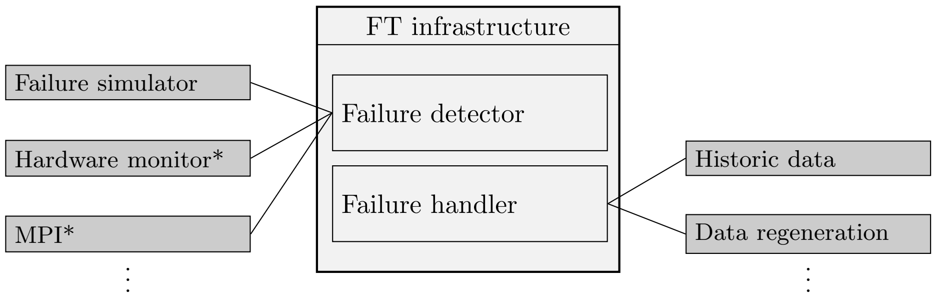


Figure 6.7: Overview of the failure infrastructure implemented in ls1-MarDyn. Failure detection and failure handling are implemented as plugins. (Plugins marked with * show possible plugins that are not implemented yet.)

6.5.1 Failure detector

The basic computational unit in ls1-MarDyn is a simulation step. The simulation should be in a consistent state after each step. Therefore, failures should be detectable with at least this granularity. Further, one may want to change or combine multiple failure detection mechanisms.

The failure detection mechanism in ls1-MarDyn is therefore implemented via a plugin infrastructure, where each detection mechanism is put into its own failure detector plugin. Each failure detector plugin provides a method with the signature `bool failure(int simstep)`, which returns `true` if it detected failures during the simulation step provided as an argument. In this way, also the failure history of the simulation may be queried later.

The plugin infrastructure allows different methods of failure identification to be used easily. For example, a failure detector could identify failures during communication, e.g., using features provided by a future fault tolerant MPI standard. Another failure detector could take into account lower-level health information obtained from HPC system monitoring services. And, for the purpose of testing, the plugin infrastructure allows for a simple way for failure injection via a simulator plugin. For the studies in this work, such a failure simulator was implemented and used. The details of it are described in the following.

Failure simulator

For testing the developed fault tolerance techniques it is not practicable to wait for real system failures: First, it may take a long time to see a failure and second, it would require to have a large target system with failure detection capability and fault resistant MPI. Therefore, ls1-MarDyn is extended with a failure simulator, which is implemented as a plugin using the failure detector interface.

A simulated failure means the loss of one or multiple processes. The processes affected by the failure are specified using the MPI ranks of the processes. The moments at which failures appear are defined by the simulation step at which they shall occur. The failure simulator provides three different failure scenarios:

Arbitrary failures: In the arbitrary failure scenario, one or multiple processes are subject to failures at arbitrary points of the simulation. Multiple failures can occur for a single process or multiple failures at multiple processes or any combination of both. Also, the failures at different processes may overlap in time. Each single failure event is defined by the MPI rank and the simulation step, at which it shall occur. The necessary failure event definitions are read from a configuration file.

Single failure: The single failure scenario is a special case where only one process fails once. This can be accomplished by the arbitrary failure scenario with only one failure definition.

Recurring failures: In the recurring failure scenario, one process is subject to recurring failures with a fixed interval. The process is specified by the MPI rank and the recurrence interval by the number of time steps between failures. In addition, a time can be provided up to which no failures are emitted, e.g., to allow undisturbed equilibration of the system at the beginning of the simulation. All necessary parameters are read from corresponding entries in the ls1-MarDyn configuration file.

6.5.2 Failure handler

The failure handler is responsible to recover the simulation from a failure. As it is currently not possible to recover the MPI environment from failures, its task for now is only to set up data in replacement for the lost ones while reusing a still running MPI process. Therefore, it collects necessary data for the chosen recovery method during simulation. When the recovery is triggered due to a failure, it identifies the affected subvolume and recovers this based on the chosen recovery method. The two recovery methods described above are implemented as handler plugins.

Historic data insertion

The historic data insertion handler uses fast in-memory checkpointing to keep a copy of all molecules for each process. For the purpose of the study in this work, the checkpoint data are stored in the processes' local memory. So, this approach does not work with real process failures for now. In order to add this functionality, the data have to be stored in neighbouring processes on other nodes as described in [77].

Data regeneration

The data regeneration handler saves, for each process, the average temperature and number of molecules at a specified interval. In case of a failure, it takes the most recent values to generate phase space data for the failed process. For generation, the handler uses the infrastructure from the flexible in memory configuration generator developed in Section 3.2. As a creation method, it uses a FCC lattice and a simple basis to insert the molecules. Velocities are assigned with the equal velocity assignment strategy.

6.6 Evaluation

In the following, the fault tolerance approach shall be evaluated with respect to its applicability to standard MD simulations.

6.6.1 Evaluation criteria

Simple thermodynamic variables

The thermodynamic properties of an ensemble can be described by the extensive variables NVE and the intensive variables μpT. These values are all accessible in MD simulations and can be computed from the phase space data [42]. For the following study, the pressure and potential energy in the system are used. Both are easy to compute and in most cases already available in the simulation. Also, they are common indicators used to check for the end of the equilibration phase and, therefore, make them perfect candidates to study the effects of disturbance in the approximate checkpoint approach.

Radial distribution function

The radial distribution function (RDF) is a central construct from which many thermodynamic variables can be determined [8]. It is accessible in simulations by computing particle pair correlations and in experiments via diffraction experiments. The

RDF $g(r)$ describes the local density around an atom or molecule with respect to the distance to a reference molecule or atom and connects it to the average system density $\rho = N/V$. It is defined such that the local density ρ_l in the distance r to the reference atom or molecule is given by [42]

$$\rho_l(r) = \rho g(r) \ . \tag{6.1}$$

The analytical form for the RDF in the canonical ensemble NVT is

$$g(r) = \frac{N(N-1)}{\rho^2 Z_{NVT}} \int \cdots \int \exp(-\beta U_N)\, dr_3 \ldots dr_N \ . \tag{6.2}$$

Here, Z_{NVT} is the canonical partition function and U_N the potential energy of all particles in the system. In MD simulations, it is computed as a time average via

$$g(r) = \frac{V}{N^2} \left\langle \sum_i \sum_{j \neq i} \delta(\vec{r} - \vec{r}_{ij}) \right\rangle \ . \tag{6.3}$$

The RDF is a relatively sensitive value that depends strongly on the state of a material, which allows to differentiate between solid, liquid, and gas. This makes it useful to track the state of system equilibration and identify disturbances in MD simulations.

6.6.2 Recovery behaviour

As the proposed new ABFT approach uses approximate data, it will disturb the system at the recovery point. Hence, it is of interest to see how the simulation reacts after the recovery point. For the evaluation of the recovery behaviour, two questions are here of interest: What are the failure conditions under which the new ABFT approach allows to achieve the correct final result and how efficient is this approach compared to the classical C/R.

As scenario for the study, a homogeneous Lennard-Jones (LJ) system consisting out of 5000 molecules at a reduced density $\rho = 0.9$ and a reduced temperature $T = 0.85$ is equilibrated with ls1-MarDyn. It is simulated using 8 processes. With 625 molecules per process, this is a scaled down scenario of typical long-running simulations with many time steps. The system is equilibrated for 10 000 time steps and is followed by a production phase of 40 000 time steps. Failures are injected during the simulation run with the previously described failure simulator according to the single failure and recurring failure scenario.

Historic data

First, the historic data approach is studied in more detail. Figure 6.8 shows the time evolution of the potential energy and pressure of the entire system around the failure event. The failure is corrected immediately after its occurrence using the historic data approach. The resolution used for the monitoring of the pressure and potential energy is 10 time steps. Shown values are the average over the preceding 10 time steps. The age of the historic data inserted in place of the lost data is varied, here. The most recent data inserted are 10 time steps behind, the oldest 200 time steps.

The results show that after the insertion of the old data, the system is disturbed heavily as the peaks in the pressure and potential energy indicate. However, the system recovers fast to its equilibrated production values. After roughly 100 time steps, the values are already within the uncertainty from a single simulation. Both, potential energy and pressure are recovered then.

Table 6.1 lists the pressure and potential energy values of the system 1000 time steps after the failure and recovery for the different data ages. The differences between the results are within the order of the standard deviation of the failure-free simulation. So, the age of the inserted data plays a minor role after the initial equilibration.

age	p	U_{pot}
reference	2.268 ± 0.015	-6.2326 ± 0.0028
10	2.269	-6.2256
20	2.282	-6.2257
50	2.286	-6.2213
200	2.255	-6.2266

Table 6.1: Pressure p and potential energy U_{pot} 1000 time steps after failure correction with the historic data approach. For the reference simulation without errors, the standard deviation for the entire run is provided.

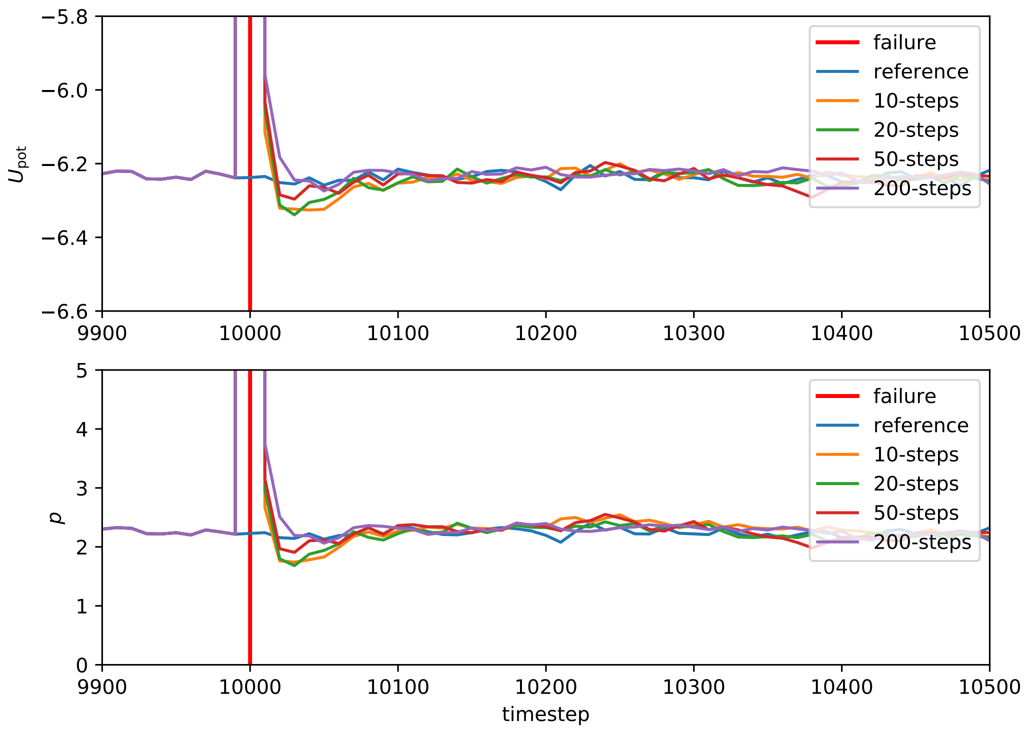


Figure 6.8: Evolution of the potential energy U_{pot} and pressure p of a LJ system simulation after failure recovery. The equilibrated system is subject to a failure at time step 10 000 (vertical red line). Here, 1 of 8 processes fails. The lost data are replaced by historic data from 10, 20, 50 or 200 time steps before. The system consists of 5000 molecules and has a reduced density $\rho = 0.9$ and a reduced temperature $T = 0.85$.

Regeneration

The same evaluation is performed for the regeneration based approach. Figure 6.9 shows the time evolution of the potential energy and pressure of the entire system after a failure event, which was corrected by inserting regenerated data. The regeneration is based on the number of molecules and average temperature 10 time steps before the failure. It is compared to the previous results from the historic data approach.

The results show that the regeneration based approach takes longer to recover than the historic data approach. The potential energy returns back to the shape of the reference curve after around 200 time steps and the pressure after around 400 time steps.

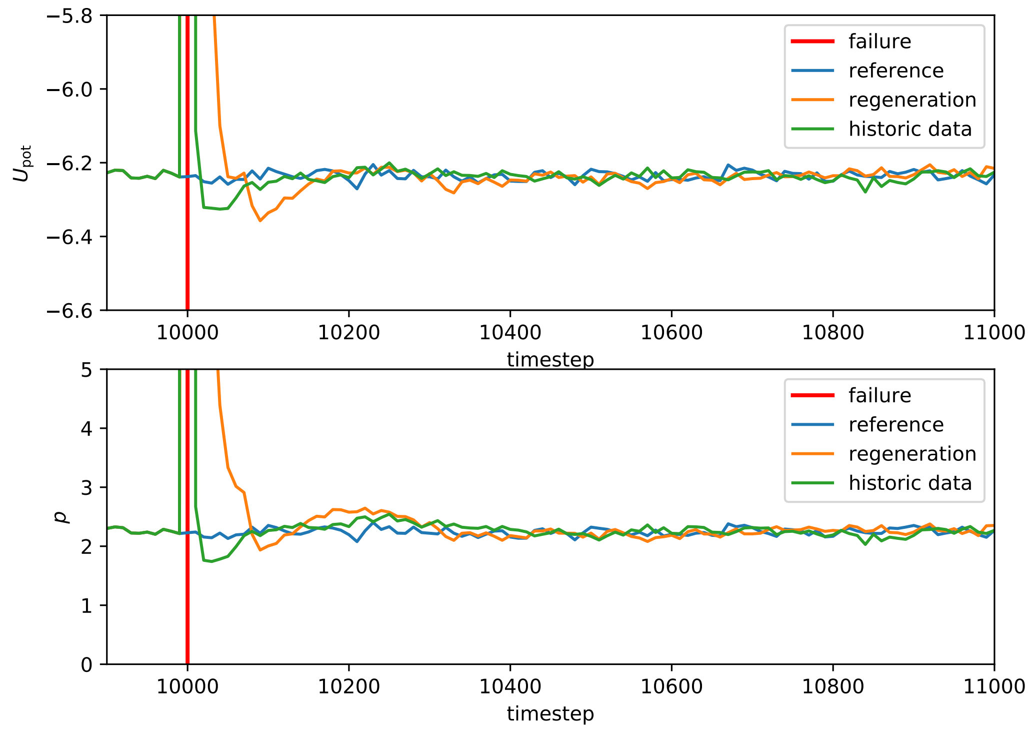


Figure 6.9: Evolution of the potential energy U_{pot} and pressure p of a LJ system simulation after failure recovery. The equilibrated system is subject to a failure at time step 10 000 (vertical red line). Here, 1 of 8 processes fails. The lost data are replaced by data, which are regenerated based on the failed process local number of molecules and temperature 10 time steps before. The regeneration is performed with a FCC lattice and equal velocity with random orientation. For comparison the historic data insertion approach with 10 time step old data is included. The system consists out of 5000 molecules and has a reduced density $\rho = 0.9$ and a reduced temperature $T = 0.85$.

6.6.3 Tolerance to recurring failures

So far both approaches showed that they can be used to recover from a single failure within relatively short time. However, it is not clear, if continued application of the recovery approaches leads to systematic errors in the simulation. Hence, in this part, the influence of recurring failures onto the simulation will be explored. The regeneration based recovery approach is selected for the following study. Due to the reduced amount of necessary restore data, this approach is especially well suited for more frequent recoveries. Also, its slightly longer recovery time has a higher probability to introduce correlations between successive applications.

The same system as for the single failure study before was used. The difference is that now one process constantly fails in a regular interval starting from time step 10 000 on. Figure 6.10 shows the time evolution for the potential energy and pressure of the simulated system for 30 000 time steps after the initial failure. Two different failure intervals are applied: One of 1000, which is larger than the time observed to be necessary to recover to a good state in the single failure scenario before, and one of 200 time steps, which is slightly shorter than this time. Clearly visible are the failures. Each time the potential energy and pressure of the entire system are disturbed.

One interesting observation here is that the disturbance for the shorter failure interval becomes smaller after around 23 000 time steps. At the same time, the values for the potential energy and pressure shift away from the reference value. The differences in the obtained values from the end of the simulations are listed in Table 6.2. For a failure interval of 200 time steps the pressure deviates more than 75 % from the reference value. So, the frequent failures and recoveries lead in this case to a change in the observed system, which cannot be corrected by the thermostat of the MD simulation. For the longer failure interval of 1000 time steps, the system has enough time for recovery in-between failure events. Hence, there are hardly any correlations leading to systematic errors in this case.

failure frequency	p	U_{pot}
reference	2.236 ± 0.067	-6.239 ± 0.013
200	0.558	-5.906
1000	2.125	-6.171

Table 6.2: Average pressure p and potential energy U_{pot} over time steps 39 990 to 39 999 under the influence of contiguous failures of different failure frequencies. For recovery the new ABFT approach with regenerated data was used.

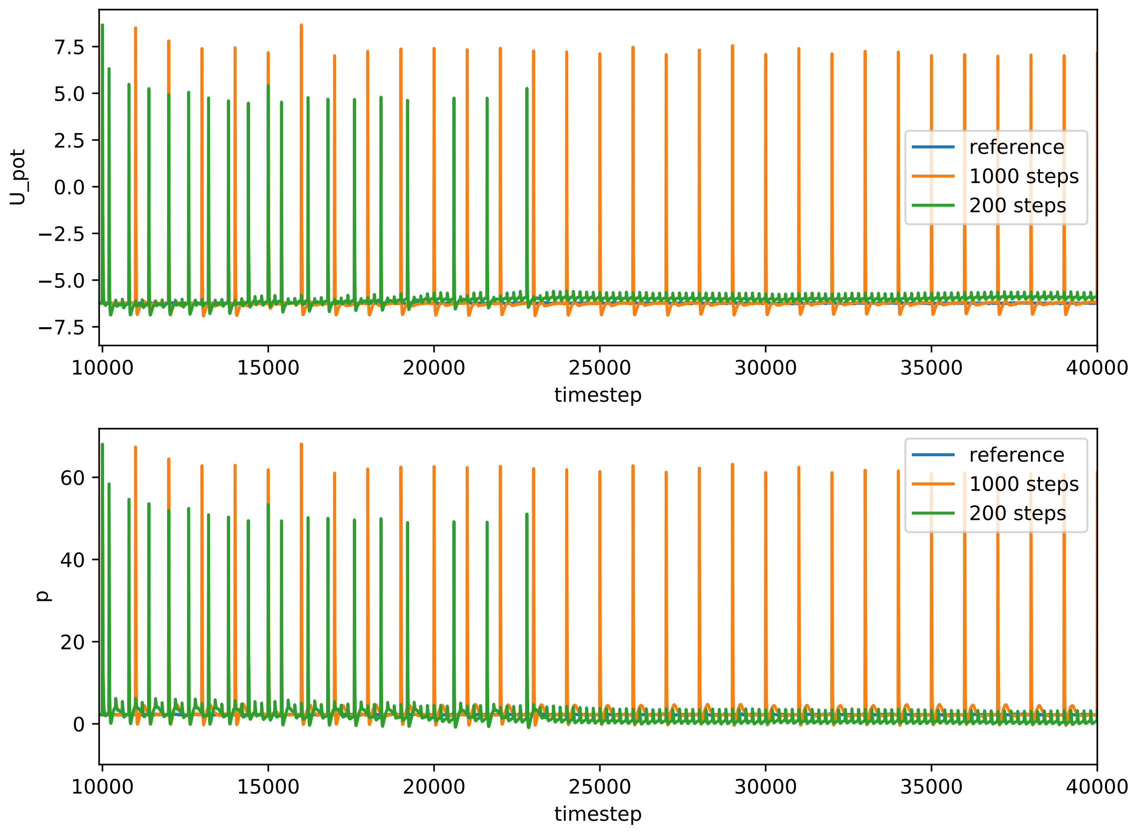


Figure 6.10: Time evolution of the potential energy U_{pot} and pressure p of a LJ system simulation subject to recurring failures. Here, 1 of 8 processes fails regularly. The lost data are replaced by data, which are regenerated based on the failed process local data. For the regeneration strategy, the local number of molecules and temperature is used. The regeneration is performed with an FCC lattice and equal velocity with random orientation. The system consists of 5000 molecules and has a reduced density $\rho = 0.9$ and a reduced temperature $T = 0.85$.

6.6.4 Masking

So far errors were corrected by the new ABFT approach and the application went on computing. While the investigated thermodynamic values recovered fast and the generated averages were well within the error margins at the end of the simulation, there is still a difference. To see the effect of the inserted data, we have a look at the more sensitive RDF. Especially the approach inserting re-generated data on the basis of a lattice will be visible here in the result. The original RDF pattern of the lattice used for the particle placement in this approach is visible in the RDF average later on. This effect can be seen in Figure 6.11 as small peaks in the RDF, e.g., at 0.9.

One strategy to minimize the influence of the inserted data on the final result is to exclude them for some time steps after the failure from the computation of these results. Figure 6.11 shows the resulting RDF when masking them out for the next 200, 400 or 500 time steps after the failure event and regeneration of the data. With increased length of the masking window, the peaks vanish and the resulting RDF curve gets closer to the reference one. This effect can be seen even better in Figure 6.12, which shows the difference between the RDF obtained in the runs with failures and the failure free-reference run. To quantify the difference, the normalized root mean square deviation (NRMSD) was computed for the difference of the RDFs and the reference. The results in Figure 6.12b show that the error decreases by up to a factor of 6 when the 500 time steps after the insertion are masked. This is in accordance with the results from the recovery behaviour analysis in Section 6.6.2, where around 400 time steps were necessary to recover the temperature and pressure values in this scenario.

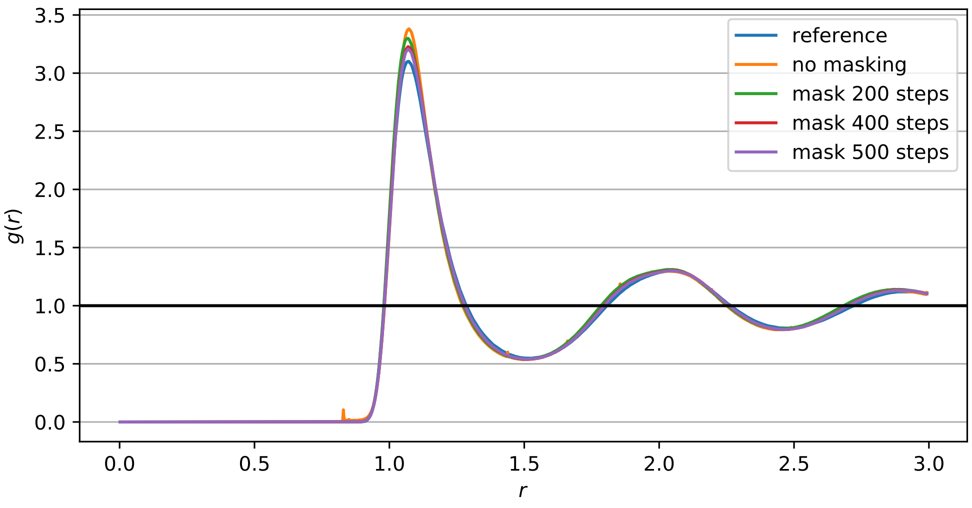
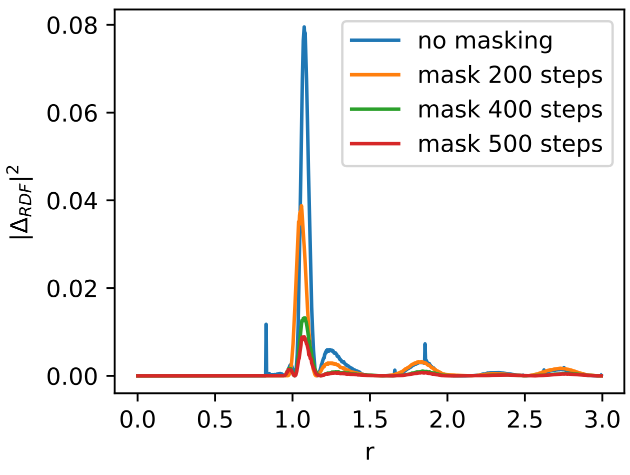


Figure 6.11: Radial distribution function obtained as average over 30 000 time steps while the simulation is subject to recurring failures of one of 8 processes for two different failure intervals.



(a) Error to reference curve

	NRMSD
reference	0.000%
no masking	0.067%
mask 200 steps	0.045%
mask 400 steps	0.016%
mask 500 steps	0.011%

(b) NRMSD

Figure 6.12: Error of the RDF for different masking interval lengths.

6.6.5 Failures during the initial equilibration phase

So far failures were studied that occur after the equilibration phase in MD simulations. However, there is the open question, wether the new ABFT approach can also be applied successfully within the initial equilibration phase of the simulation, where often special techniques are applied to prepare the state. Figure 6.13 shows the time evolution of the potential energy and pressure with recurring failures every 1000 time steps in one process from the start of a simulation. The ls1-MarDyn code uses here the method of equilibration at increased temperature [47]. In the shown scenario, the simulation follows a protocol that linearly increases and decreases the temperature during the first 5000 time steps of the simulation. As can be seen, the disturbance of the system caused by the inserted data shows the same behaviour as in the equilibrated state. The system is able to recover quickly enough here too, allowing it to follow the equilibration protocol in the same way as the reference curve.

6.6.6 Performance comparison to classical C/R

Now, after showing that the new ABFT approach can be applied to MD simulation scenarios, a short comparison in terms of its efficiency with respect to the classical C/R shall be made.

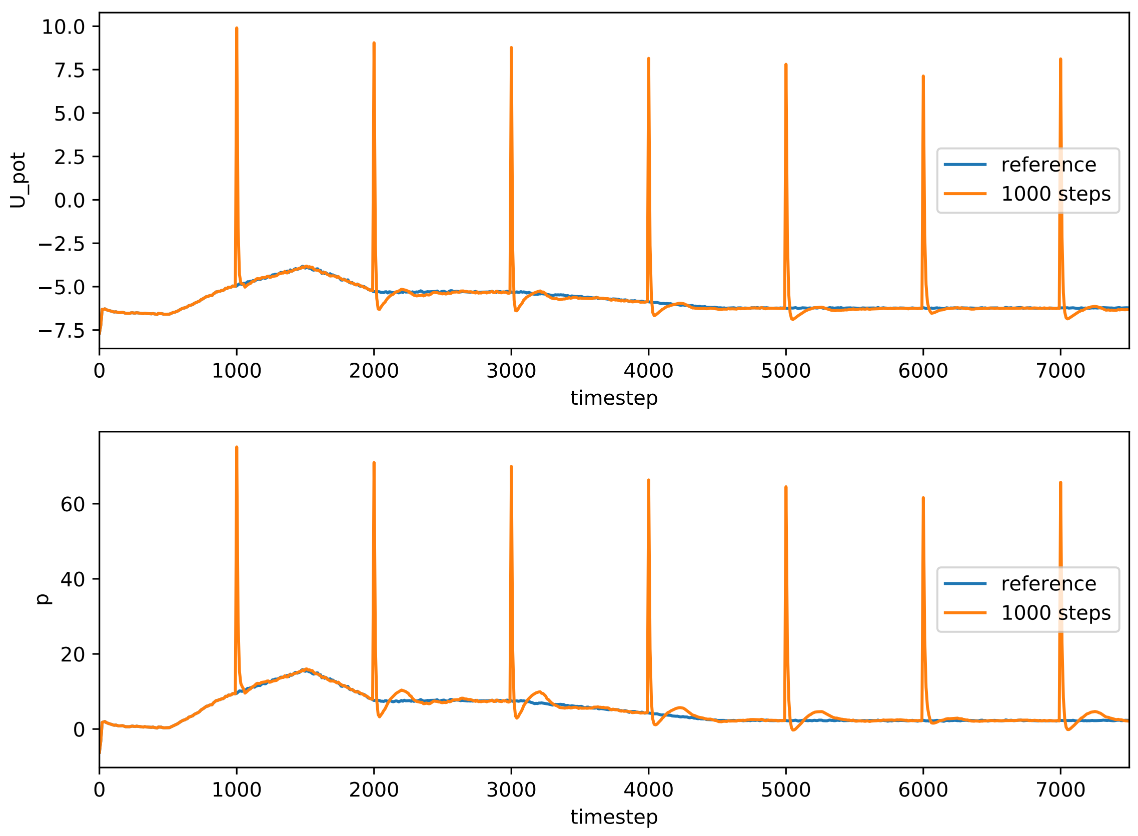


Figure 6.13: Time evolution of the potential energy U_{pot} and pressure p of a LJ system simulation subject to recurring failures during the equilibration phase.

For the studied small scenario, the median checkpoint dump time on the Vulcan system is found to be $\delta = 0.12\,\text{s}$. With an average computational time of 4.8 ms per time step, this corresponds to a time needed for the computation of 25 time steps. Under the assumption that reading the checkpoint takes the same time, i.e., $R = \delta$ and an MTBF $M = 1\,\text{d} = 18\,000\,000$ timesteps, one gets with Equation (2.11) an optimal checkpoint interval $\tau_{\text{opt}} = 30\,000$ timesteps. Hence, for the classical C/R, 600 checkpoints have to be dumped on average between failure events. The total overhead of the classical C/R approach is therefore $M/\tau_{\text{opt}} * \delta + R = 15\,025$ timesteps. In comparison, the overhead of the new ABFT method using the regeneration approach with masking loses around 500 timesteps during the recovery process, but has a negligible dump time. Therefore, the presented new ABFT method can save more than 96 % of the time necessary for fault tolerance handling in this case.

6.7 Chapter summary

In this section, a new ABFT approach for MD simulations was developed. The strategy in the new approach includes the replacement of lost simulation data on the fly. Two replacement strategies were presented: The recovery based on historic data and the regeneration based on the average system temperature and local particle count. In comparison to the classical C/R approach, no roll-back of the system is performed after the replacement step. The main question for the applicability of this new approach was the convergence behaviour of the simulation under the presence of one or multiple failure events.

To test the applicability of the new ABFT approach, it was implemented in ls1-MarDyn. The implementation is split into two functional components: the failure detector and the failure handler. The failure handler is used to simulate failures in a controlled way, while the failure handler implements the different recovery strategies. Different failure scenarios were used during the evaluation: Single failure, recurring failure after equilibration, and recurring failure during the initial equilibration phase of the simulation. The pressure, the potential energy, and the radial distribution function were used as criteria for the evaluation of the equivalence of the simulation results as well as the study of the recovery behaviour. Results obtained with the new ABFT approach under simulated failures were compared against results from a reference simulation without failures.

In the case of single event failures, the chosen scenario was found to require not more than 400 time steps to recover from the failure. The overall error was shown to be within the standard deviation of the regular simulation 1000 time steps after the failure event.

For recurring failures, it was shown that the time between failures can change the outcome, if the time interval between failures is shorter than the time necessary for the recovery to a good state. However, the approach worked fine for the chosen scenario when the time between failures was larger than the needed recovery time.

A masking strategy was introduced to improve the accuracy of simulations with failures corrected by the new ABFT approach. The masking strategy excludes the simulation steps after the failure event. For the evaluation, the more sensitive RDF was used as metric here. The obtained results show that masking can help increase the accuracy of the simulation results in this case by up to a factor of 6.

Errors during the initial equilibration phase of MD simulations were studied and the reported results show that even in the case of special equilibration techniques—as used in ls1-MarDyn—the system will achieve equilibration.

Finally, it was shown that the new ABFT approach can reduce the fault tolerance overhead by around 96 % for the chosen scenario.

To conclude, the new fault tolerance approach is capable to recover MD simulations successfully for the chosen scenario in nearly all studied failure scenarios, while the overhead is much lower than for classical C/R.

7 Summary and Outlook

7.1 Summary

In this work, various concepts for Molecular Dynamics (MD) simulations were studied with respect to the advancement of their scaling on future HPC systems. Therefore, numerous existing approaches in different areas were evaluated. Based on the results, improvements were made or new approaches introduced to overcome identified limitations. Experiments using benchmark kernels and the real-world code ls1-MarDyn were used to show the applicability and to quantify their performance gains.

Aspects of initial configuration creation involved the analysis of the current workflow for the ls1-MarDyn code. The practice there was found to be based heavily on input and output (I/O) with an ASCII file format. This was identified as a clear bottleneck preventing runs at larger scale. The I/O part was therefore first tuned with existing broadcasting techniques making use of the fast network over MPI communication for the I/O. The achieved improvement allowed to run user scenarios with up to 100 000 processes, which was not possible before. However, the used approach does still not scale with the scenario size itself. Therefore, a new concept of scenario generation was developed. This new approach introduces the possibility of distributed scenario generation in a transparent way for the users. It was implemented in ls1-MarDyn and evaluated with a real world scenario. The results show that the approach achieves its goal of an user-friendly interface that allows scalability up to thousands of processes even for complex scenarios.

As far as parallel computations are concerned, the necessary computation has to be distributed. Due to the increasing parallelism, subproblem sizes become smaller. Therefore the most important algorithms for the time consuming force calculation in

MD simulations were investigated in detail. Performance models for the commonly used algorithms in MD simulations were created to allow runtime predictions for them with low number of molecules. The models were evaluated using various benchmarks. Based on the models, an algorithm selection diagram was introduced. This shows that the naïve algorithm for a typical cut-off of $r_\mathrm{c} = 5$ is comparable to or better than the commonly used linked-cell and neighbour list algorithm in the region around and below 1000 molecules. The particle density was found to be relevant as a criterion for the choice of the best algorithm.

For the most important basic force calculation algorithm, a wide variety of different implementations was studied with respect to performance and comparability of the results. The influence of the compiler and target architecture was evaluated in this context. As a result, it turned out that particularly algorithms looping diagonally over the elements of the force matrix perform well when compiled with Intel or GCC on the studied cache based architectures, i.e., Intel Xeon systems. The traditional implementation approaches using loops over the rows and columns of the force matrix were found to be a bit slower here. However, the latter performed better on the NEC SX vector system. Concerning the equality of the results of the different implementations, the normalized root mean square deviation (NRMSD) values for forces and energy were compared. Here, the algorithms were found to differ slightly, but not critically.

Regarding parallelization at the node level, the increasingly important task-based parallelization approach was applied to the linked-cell algorithm. Shared Memory parallelization was studied using the SmpSs and OmpSs runtime systems. A pitfall leading to limited speedup caused by dependencies was investigated here. Solutions for code developers werde presented and shown to provide perfect scaling. Additionally, suggestions for an extension of the programming model to overcome the underlying problem were made.

In the context of Distributed Memory parallelization, the two main communication strategies for the necessary particle exchange in the domain decomposition based parallelization were re-evaluated: Direct and folding based data exchange. The recommendation of the past to use the folding approach was found to be still valid for current HPC systems. However, process placement was identified to be a big issue for the future. As an improvement to mitigate the negative effects on current systems, potential for overlapping of communication and computation were identified in the ls1-MarDyn code. This allowed for an improvement of the communication time by up to 8 % running with more than a thousand processes.

Fault tolerance as one of the aspects becoming more and more important was addressed. A study of failures in the HAWK system at HLRS confirmed the existing predictions of failure frequencies of more than one failure event per day. The limits of the classical checkpoint/restart (C/R) approach currently used in the codes was examined using the state of the art model for the optimal checkpointing time. The optimal C/R parameters for the ls1-MarDyn code resulted in overheads larger than 2 percent. One of the important parameters to reduce overheads was identified to be the recovery-data dump-time. Therefore, a new application-based fault tolerance (ABFT) approach for MD simulations was invented that minimizes the amount of needed recovery-data. This approach was implemented in the ls1-MarDyn code and successfully applied to common homogeneous simulation scenarios. It was shown that the overheads necessary for fault tolerance handling can be reduced by more than 96 percent based on the studied scenarios for current systems.

All these different aspects are important to scale up MD simulations in the future in terms of the system size as well as runtime. The results from this work help to improve each one of them. The modifications applied to ls1-MarDyn were helping to scale up the application to more than 100 000 cores and were the starting point for further optimizations, which resulted lastly in the simulation of around 20 trillion atoms [89].

7.2 Outlook

While the results in this work already showed ways to overcome limitations of current MD codes, there is still room for further research and improvements in each of the aspects presented.

For example, the new in memory scenario generation system is not aware of load balancing across processes. For specific systems, this may lead to memory issues if some processes have to deal with many molecules that exceed the memory of a single node. Also an excessive and expensive re-distribution of molecules across processes directly after the initialisation should be prevented for such cases. Beside this, methods to simplify the in-memory scenario generation process for the user with the help of graphical tools instead of the current XML input file based approach would be a useful addition.

More interesting research questions arise especially in the topics of the automatic algorithm selection, new task-based parallelization models, process mapping, and the introduced new ABFT approach, which all seem interesting for follow-up work to this thesis.

Based on the performance model from this work, applications may dynamically adjust their behaviour during runtime. They may even decide to select different algorithms for different subdomains of the simulation domain based on the criteria from this work, helping to improve the overall efficiency. However, this requires applications to implement those different algorithms as well as the algorithm-exchange at runtime, which is not the case for ls1-MarDyn at the moment. One interesting new question coming up then is the question how the automatic algorithm selection for a subdomain of the simulated system interacts with the load balancing schemes used across processes.

At this point, the task-based parallelization approach studied in this work will become even more of interest as it allows for a much better and dynamic load balancing. Here, the focus will be clearly on methods to simplify dependencies to allow the runtime to extract more parallelism. Also, the task distribution may be applied across processes as suggested in, e.g., HPX [54]. In this case, the tuning of the scheduling strategies and task locality will become important.

For the MPI based Distributed Memory parallelization, the folding based neighbour communication scheme in MD codes was found to be still the best choice. Some improvements by overlapping communication and computation were made. However, the results of this work showed that communication suffers from the heavy inter-node communication on current and future systems, which makes application to hardware topology mapping important. The resulting performance degradation, especially in NUMA systems, was already subject to recent research [72] and an extension for a new API to the MPI standard [73]. Therefore, further studies related to specialized methods for the process placement and locality optimization are a good candidate for improvements and are likely to have an impact also on stencil based code outside the MD simulation community.

The new ABFT approach developed in this work was applied to homogeneous scenarios. A lot of questions may be posed around its applicability to more sophisticated scenarios that react sensitively to local properties of the system, e.g., the local parts of crack propagation or thin interfaces. To handle such situations, the presented fault tolerance approach may be extended, e.g., to take into account a more detailed particle distribution by linear or higher order approximations.

To conclude, there are many ideas that could be further explored when preparing MD simulations for even larger systems based on the results of this work. However new systems will bring new challenges making this a never ending cycle.

Bibliography

[1] NEC Cluster Disk Storage. (accessed 4th November 2018).

[2] NEC SX-ACE. (accessed 4th November 2018).

[3] TOP500 List (November 2020). `https://www.top500.org/list/2018/11/`. Accessed: 2020-12-27.

[4] ULFM Specification, February 2017.

[5] The Open Group Base Specifications Issue 7. IEEE Std, IEEE, 2018. Last accessed December 27, 2020.

[6] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. SIAM Journal on Computing, 1(2):131–137, 1972.

[7] B. J. Alder and T. E. Wainwright. Studies in Molecular Dynamics. I. General Method. J. Chem. Phys., 31:459, 1959.

[8] M. P. Allen and D. J. Tildesley. Computer Simulation of Liquids. Clarendon Press, New York, NY, USA, 1989.

[9] Robert Alverson, Duncan Roweth, and Larry Kaplan. The Gemini System Interconnect. In 2010 18th IEEE Symposium on High Performance Interconnects, pages 83–87. IEEE, August 2010.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In AFIPS Spring Joint Computing Conference, volume 30 of AFIPS Conference Proceedings, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

[12] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In Proceedings of 2011 international conference for high performance computing, networking, storage and analysis, pages 1–32, 2011.

[13] Stefan Becker, Maximilian Kohns, Herbert M. Urbassek, Martin Horsch, and Hans Hasse. Static and Dynamic Wetting Behavior of Drops on Impregnated Structured Walls by Molecular Dynamics Simulation. The Journal of Physical Chemistry C, 121(23):12669–12683, 2017.

[14] Gordon Bell, David H Bailey, Jack Dongarra, Alan H Karp, and Kevin Walsh. A look back on 30 years of the Gordon Bell Prize. The International Journal of High Performance Computing Applications, 31(6):469–484, 2017.

[15] Jon Bentley and Bob Floyd. Programming Pearls: A Sample of Brilliance. Commun. ACM, 30(9):754–757, September 1987.

[16] Herman JC Berendsen, David van der Spoel, and Rudi van Drunen. GROMACS: a message-passing parallel molecular dynamics implementation. Computer physics communications, 91(1-3):43–56, 1995.

[17] Wesley Bland, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. A proposal for User-Level Failure Mitigation in the MPI-3 standard. Department of Electrical Engineering and Computer Science, University of Tennessee, 945, 2012.

[18] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. The International Journal of High Performance Computing Applications, 27(3):244–254, 2013.

[19] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.0, October 1197.

[20] OpenMP Architecture Review Board. OpenMP Application Programming Interface, November 2018.

[21] Peter J Braam and Philip Schwan. Lustre: The intergalactic file system. In Ottawa Linux Symposium, volume 8, pages 3429–3441, 2002.

[22] Ron Brightwell, Kevin T. Pedretti, and Keith D. Underwood. Initial Performance Evaluation of the Cray SeaStar Interconnect. In Hot Interconnects, pages 51–57. IEEE Computer Society, 2005.

[23] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. TEMANEJO - a debugger for task based parallel programming models. CoRR, 2011.

[24] Franck Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. IJHPCA, 23(3):212–226, 2009.

[25] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In Karsten M. Decker and René M. Rehmann, editors, Programming Environments for Massively Parallel Distributed Systems, pages 213–218, Basel, 1994. Birkhäuser Basel.

[26] Intel Corporation. Lustre Software Release 2.x Operations Manual, 2017.

[27] John Daly. A Model for Predicting the Optimum Checkpoint Interval for Restart Dumps. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, International Conference on Computational Science, volume 2660 of Lecture Notes in Computer Science, pages 3–12. Springer, 2003.

[28] W. Demtröder. Experimentalphysik 1: Mechanik und Wärme. Springer Verlag, 4. edition, 2006.

[29] Jack Dongarra, Thomas Herault, and Yves Robert. Fault Tolerance Techniques for High-Performance Computing, pages 3–85. Springer International Publishing, Cham, 2015.

[30] Jack Dongarra, Hans W. Meuer, and Erich Strohmaier. TOP500 supercomputer sites. Supercomputer, 13, 01 2000.

[31] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters, 21(02):173–193, 2011.

[32] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters, 21(2):173–193, 2011.

[33] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, Martin Bernreuther, Colin W. Glass, Christoph Niethammer, Arndt Bode, and Hans-Joachim Bungartz. 591 TFLOPS Multi-trillion Particles Simulation on SuperMUC. In Julian M. Kunkel, Thomas Ludwig, and Hans Werner Meuer, editors, ISC, volume 7905 of Lecture Notes in Computer Science, pages 1–12. Springer, 2013.

[34] Ryusuke Egawa, Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Akihiro Musa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing, 73(9):3948–3976, Sep 2017.

[35] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv., 34(3):375–408, 2002.

[36] David J Evans. Parallel SOR iterative methods. Parallel computing, 1(1):3–18, 1984.

[37] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray cascade: a scalable HPC system based on a Dragonfly network. In SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–9. IEEE, 2012.

[38] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold.

Evaluating the viability of process replication reliability for exascale systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 44:1–44:12. ACM, 2011.

[39] Agner Fog. Instruction tables, 1996-2019.

[40] James Foley. Computer graphics : principles and practice. Addison-Wesley, Reading, Mass, 1995.

[41] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1, 2015.

[42] Daan Frenkel and Berend Smit. Understanding Molecular Simulation. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2001.

[43] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. A Prototype Implementation of OpenMP Task Dependency Support. volume 8122, pages 128–140, 09 2013.

[44] Pedro Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. Journal of Computational Chemistry, 28(2):570–573, 2007.

[45] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications. Springer Publishing Company, Incorporated, 1st edition, 2007.

[46] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In Journal of Physics: Conference Series, volume 46, page 067. IOP Publishing, 2006.

[47] Darrall Henderson, Sheldon Jacobson, and Alan Johnson. The Theory and Practice of Simulated Annealing, pages 287–319. 04 2006.

[48] Thomas Herault and Yves Robert. Fault-Tolerance Techniques for High-Performance Computing. Springer Publishing Company, Incorporated, 1st edition, 2015.

[49] Martin Horsch, Martina Heitzig, Calin Dan, Jens Harting, Hans Hasse, and Jadran Vrabec. Contact Angle Dependence on the Fluid- Wall Dispersive Energy. Langmuir, 26(13):10913–10917, 2010.

[50] Martin Horsch, Christoph Niethammer, Jadran Vrabec, and Hans Hasse. Computational Molecular Engineering as an Emerging Technology in Process Engineering. Informat. Technol., 55, 05 2013.

[51] Martin Horsch, Jadran Vrabec, Martin Bernreuther, Sebastian Grottel, Guido Reina, Andrea Wix, Karlheinz Schaber, and Hans Hasse. Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. Journal of Chemical Physics, 128(16):164510, 2008.

[52] Z. Hussain, T. Znati, and R. Melhem. Optimal Placement of In-memory Checkpoints Under Heterogeneous Failure Likelihoods. In 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 900–910, 2019.

[53] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, E Weinan, and Linfeng Zhang. Pushing the Limit of Molecular Dynamics with Ab Initio Accuracy to 100 Million Atoms with Machine Learning. In SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–14. IEEE, November 2020.

[54] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, page 6. ACM, 2014.

[55] M Karplus. Molecular dynamics of biological macromolecules: A brief history and perspective. BIOPOLYMERS, 68(3):350–358, MAR 2003. Molecular dynamics of biological macromolecules: A brief history and perspective Biopolymers Volume 68, Issue 3, Date: March 2003, Pages: 350-358 Martin Karplus.

[56] John Kim, William Dally, Steve Scott, and Dennis Abts. Cost-efficient dragonfly topology for large-scale systems. IEEE micro, 29(1):33–40, 2009.

[57] Charles Kittel. Introduction to Solid State Physics. Wiley, 8 edition, 2004.

[58] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in MPI applications. Int. J. High Perform. Comput. Appl., 30(3):305–319, 2016.

[59] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. IEEE transactions on Computers, 100(10):892–901, 1985.

[60] Zhen Li, Rohit Atre, Zia Ul Huda, Ali Jannesari, and Felix Wolf. DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities. 08 2015.

[61] Xavier Martorell, Rosa M. Badia, and Xavier Teruel. Introduction to the OmpSs Programming Model, 2018.

[62] D. R. Mason. Faster neighbour list generation using a novel lattice vector representation. Computer Physics Communications, 170(1):31–41, 2005.

[63] William Mattson and Betsy M Rice. Near-neighbor calculations using a modified cell-linked list method. Computer Physics Communications, 119(2-3):135–148, 1999.

[64] M I Mendelev, M J Rahman, J J Hoyt, and M Asta. Molecular-dynamics study of solid–liquid interface migration in fcc metals. Modelling and Simulation in Materials Science and Engineering, 18(7):074002, sep 2010.

[65] Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. The TOP500: History, Trends, and Future Directions in High Performance Computing. Chapman & Hall/CRC, 1st edition, 2014.

[66] David L. Millman, D. Griesheimer, Brian R. Nease, and J. Snoeyink. Computing Numerically-Optimal Bounding Boxes for Constructive Solid Geometry (CSG) Components in Monte Carlo Particle Transport Calculations. In ICS 2014, 2014.

[67] Gordon E. Moore. Cramming more components onto integrated circuits. Electronics, 38(8), April 1965.

[68] Matthew Newville, Till Stensitzki, Daniel B. Allen, and Antonino Ingargiola. LMFIT: Non-Linear Least-Square Minimization and Curve-Fitting for Python, September 2014.

[69] Christoph Niethammer. Performance Evaluation and Optimization of the ls1-MarDyn Molecular Dynamics code on the Cray XE6. 2012.

[70] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans Hasse, et al. ls1 mardyn: The massively parallel molecular dynamics code for large systems. Journal of chemical theory and computation, 10(10):4455–4464, 2014.

[71] Christoph Niethammer, Colin W. Glass, and Jose Gracia. Avoiding Serialization Effects in Data / Dependency Aware Task Parallel Algorithms for Spatial Decomposition. In Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12, pages 743–748. IEEE Computer Society, 2012.

[72] Christoph Niethammer and Rolf Rabenseifner. An MPI Interface for Application and Hardware Aware Cartesian Topology Optimization. In Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19, New York, NY, USA, 2019. Association for Computing Machinery.

[73] Christoph Niethammer and Rolf Rabenseifner. Proposal to the MPI standard for the addition of Topology aware Cartesian communicators, 2019. (online https://github.com/mpi-forum/mpi-issues/issues/120).

[74] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.

[75] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In CLUSTER, pages 142–151. IEEE Computer Society, 2008.

[76] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical Task-Based Programming With StarSs. IJHPCA, 23(3):284–299, 2009.

[77] James S. Plank, Kai Li, and Michael A. Puening. Diskless Checkpointing. IEEE Trans. Parallel Distributed Syst., 9(10):972–986, 1998.

[78] Steve Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. Journal of Computational Physics, 117(1):1–19, 1995.

[79] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. Second Edition. 1992.

[80] Gary S. Grest, Burkhard Dünweg, and Kurt Kremer. Vectorized link cell Fortran code for molecular dynamics simulations for a large number of particles. Computer Physics Communications, 55:269–285, 10 1989.

[81] Sandia Corporation. LAMMPS Users Manual, 3 2017.

[82] P. Schofield. Computer simulation studies of the liquid state. Computer Physics Communications, 5(1):17–23, 1973.

[83] Tom Shanley. InfiniBand network architecture. Addison-Wesley Professional, 2003.

[84] F. Siperstein, A. L. Meyers, and O. Talu. Long range corrections for computer simulations of adsorption. Molecular Physics, 100(13):2025–2030, 2002.

[85] J Stadler, R Mikulla, and H-R Trebin. IMD: a software package for molecular dynamics studies on parallel computers. International Journal of Modern Physics C, 8(05):1131–1140, 1997.

[86] Martin Steinhauser and Stefan Hiermaier. A Review of Computational Methods in Materials Science: Examples from Shock-Wave and Polymer Physics. International journal of molecular sciences, 10:5135–216, 12 2009.

[87] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. ACM Trans. Comput. Syst., 3(3):204–226, 1985.

[88] Vladimir Subotic, Arturo Campos, Alejandro Velasco, Eduard Ayguade, Jesús Labarta, and Mateo Valero. Tareador: The Unbearable Lightness of Exploring Parallelism, pages 55–79. 01 2015.

[89] Nikola Tchipev, Steffen Seckler, Matthias Heinen, Jadran Vrabec, Fabio Gratl, Martin Horsch, Martin Bernreuther, Colin W Glass, Christoph Niethammer, Nicolay Hammer, Bernd Krischok, Michael Resch, Dieter Kranzlmüller, Hans Hasse, Hans-Joachim Bungartz, and Philipp Neumann. TweTriS: Twenty

trillion-atom simulation. The International Journal of High Performance Computing Applications, 33(5):838–854, 2019.

[90] Jesper Larsson Träff, William Gropp, and Rajeev Thakur. Self-consistent MPI performance requirements. In European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pages 36–45. Springer, 2007.

[91] Hans-Rainer Trebin, Ralph Mikulla, Jörg Stadler, Gunther Schaaf, and Peter Gumbsch. Molecular dynamics simulations of crack propagation in quasicrystals. Computer Physics Communications, 121-122:536–539, 1999. Proceedings of the Europhysics Conference on Computational Physics CCP 1998.

[92] Andrij Trokhymchuk and José Alejandre. Computer simulations of liquid/vapor interface in Lennard-Jones fluids: Some questions and answers. The Journal of Chemical Physics, 111(18):8510–8523, 1999.

[93] Wilfred F. van Gunsteren and Herman J. C. Berendsen. Computer Simulation of Molecular Dynamics: Methodology, Applications, and Perspectives in Chemistry. 1990.

[94] Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. Phys. Rev., 159:98–103, Jul 1967.

[95] M. Vijay and R. Mittal. Algorithm-based fault tolerance: a review. Microprocessors and Microsystems, 21(3):151–161, 1997. Fault Tolerant Computing.

[96] Jadran Vrabec, Gaurav Kumar Kedia, Guido Fuchs, and Hans Hasse. Comprehensive study of the vapour–liquid coexistence of the truncated and shifted Lennard–Jones fluid including planar and spherical interface properties. Molecular Physics, 104(9):1509–1527, 2006.

[97] Chip Walter. Kryder's law. Scientific American, 293(2):32–33, 2005.

[98] Rui Wang, Erlin Yao, Mingyu Chen, Guangming Tan, Pavan Balaji, and Darius Buntinas. Building algorithmically nonstop fault tolerant MPI programs. In 2011 18th International Conference on High Performance Computing, pages 1–9. IEEE, 2011.

[99] Eric W. Weisstein. Sphere point picking. From MathWorld—A Wolfram Web Resource. Accessed: 2020-09-15.

[100] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms. Computer Physics Communications, 182(3):611–615, 2011.

[101] Ulrich Welling and Guido Germano. Efficiency of linked cell algorithms . Computer Physics Communications, 182(3):611–615, 2011.

[102] John W. Young. A First Order Approximation to the Optimum Checkpoint Interval. Commun. ACM, 17(9):530–531, September 1974.

[103] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, and E Weinan. Deep Potential Molecular Dynamics: a scalable model with the accuracy of quantum mechanics. Phys. Rev. Lett., 120:143001, December 2018.