Exascale-Ready Molecular Dynamics Simulations with Efficient Algorithms for Extreme Core Counts

Von der Fakultät für Mathematik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des
akademischen Grades eines Doktors der Naturwissenschaften
genehmigte Dissertation

vorgelegt von

Nitin Malapally, M.Sc.

aus Bangalore, India

Berichter: Prof. Dr. Paolo Carloni
Prof. Dr. Maria Fyta

Tag der mündlichen Prüfung: 12.01.2026

I hereby declare that I have created this work completely on my own and used no other sources or tools than the ones listed, and that I have marked any citations accordingly.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 05.*2025*
N. Malapally

Contents

List of Figures

List of Abbreviations

MD	Molecular Dynamics
HPC	High-Performance Computing
PBC	Periodic Boundary Conditions
CPU	Central Processing Unit
GPU	Graphics Processing Unit
3D DFT	3D Discrete Fourier Transform
DFT	Density Functional Theory
QM	(adj.) quantum mechanical
SIMD	Single Instruction (stream) Multiple Data (stream)
SMT	Simultaneous MultiThreading
B	Byte
GB	Gigabyte
TB	Terabyte
FLOP	Floating-Point Operation
UMA	Uniform Memory Access
NUMA	Non-Uniform Memory Access
PE	Processing Element
MPI	Message Passing Interface
API	Application Programming Interface
cMD	Classical MD
QM/MM	quantum mechanical/molecular mechanical (method)

PEF	Potential Energy Function
PME	Particle-Mesh Ewald
aiMD	*ab initio* Molecular Dynamics
TISE	Time-Independent Schrödinger (wave) Equation
LD	Langevin Dynamics
BD	Brownian Dynamics
CV	Collective Variable
MoP	Metadynamics of Paths
FDM	Finite Difference Method
DD	Domain Decomposition
PP	Particle-Particle (domain decomposition)
COMM	Centre-Of-Mass Motion
GAI	Global Atomic Index
3D FFT	3D Fast Fourier Transform

Abstract

Biomolecular simulations, realized by molecular dynamics (MD) and enhanced-sampling approaches, are very powerful tools for studying the structural dynamics, kinetics, and energetics of biological systems. In combination with high-performance computing (HPC), increasingly larger systems and longer timescales can be simulated. However, the sequential nature of MD's time evolution imposes a hard parallel limit, resulting in reduced scalability and hence under-utilization of HPC systems. As a result, standard MD does not reach the typical timescale (millisecond and beyond) required to study many biological processes. Enhanced-sampling techniques, such as umbrella sampling, metadynamics, and replica-exchange MD, do simulate these long processes but often require various techniques to retrieve kinetic and thermodynamic properties and do not scale well. The arrival of exascale computers has made the need for highly scalable algorithms for MD simulations even more urgent. This doctoral thesis reports on my efforts to address these important issues via algorithmic optimization, design and development. The first was an attempt to speed up MD simulations by means of an alternative parallel 3D discrete Fourier transform (3D DFT) algorithm, which was implemented and benchmarked on the JUWELS Cluster, showing comparable scaling performance to the state-of-the-art. In the second, the software apparatus required for a massively parallel MD strategy was constructed within the highly popular GROMACS code and the PLUMED library. The implementation is capable of both multi-CPU and multi-GPU parallelism and was optimized and benchmarked on the JUWELS Booster. The results revealed its multi-modal scalability in that simulations using it can be both efficiently sped up and also greatly extended for a small increase in runtime. The implementation was shown to scale up to 94% of the JUWELS Booster (3,500 GPUs and 42,000 CPUs) with excellent parallel efficiency. Moreover, a plateauing of parallel efficiency was observed at 50% of the JUWELS Booster, which hints at its ability to scale to even higher node counts. This software has the potential to accelerate MD and thereby enable the study of more complex biological processes than was previously practicable.

Zusammenfassung

Biomolekulare Simulationen, die durch Molekulardynamik (MD) und "Enhanced-Sampling"-Methoden realisiert werden, sind sehr leistungsfähige Werkzeuge zur Untersuchung der strukturellen Dynamik, Kinetik und Energetik biologischer Systeme. In Kombination mit Hochleistungsrechnen (HPC) können immer größere Systeme und längere Zeitskalen simuliert werden. Allerdings setzt die sequentielle Natur der Zeitentwicklung der MD-Simulation eine harte Parallelitätsgrenze, was zu einer verringerten Skalierbarkeit und damit zu einer Unterauslastung von HPC-Systemen führt. Daher erreicht die Standard-MD die typischen Zeitskalen (Millisekunden und darüber hinaus) nicht, die für die Untersuchung vieler biologischer Prozesse erforderlich sind. Enhanced-Sampling-Techniken wie Umbrella Sampling, Metadynamik und Replica-Exchange-MD ermöglichen die Simulation dieser langen Prozesse, erfordern jedoch häufig verschiedene Methoden, um kinetische und thermodynamische Eigenschaften zu extrahieren, und skalieren zudem nicht gut. Mit dem Aufkommen von Exascale-Computern ist der Bedarf an hochskalierbaren Algorithmen für MD-Simulationen noch dringlicher geworden. Diese Dissertation berichtet über meine Bemühungen, diese wichtigen Herausforderungen durch Optimierung, Entwurf und Entwicklung von Algorithmen anzugehen. Die erste Arbeit war ein Versuch, MD-Simulationen durch einen alternativen parallelen Algorithmus zur 3D diskreten Fourier-Transformation (3D DFT) zu beschleunigen, der auf dem JUWELS-Cluster implementiert und getestet wurde und dabei eine mit dem aktuellen Stand der Technik vergleichbare Skalierbarkeitsleistung zeigte. In der zweiten Arbeit wurde die für eine massiv parallele MD-Strategie erforderliche Softwarevorrichtung innerhalb des weit verbreiteten GROMACS-Codes und der PLUMED-Bibliothek erstellt. Die Implementierung unterstützt sowohl Multi-CPU- als auch Multi-GPU-Parallelismus und wurde auf dem JUWELS Booster optimiert und getestet. Die Ergebnisse zeigten eine multimodale Skalierbarkeit, da Simulationen damit sowohl effizient beschleunigt als auch bei geringer Laufzeiterhöhung deutlich verlängert werden können. Es zeigte sich, dass die Implementierung mit exzellenter parallelen Effizienz auf bis zu 94% des JUWELS Boosters (3.500 GPUs und 42.000 CPUs) skalierbar ist. Zudem wurde ein Plateau der parallelen Effizienz bei 50% des JUWELS Boosters beobachtet, was auf die Fähigkeit der Implementierung hindeutet, auf noch höhere Node-Anzahlen skalieren zu können. Diese Software hat das Potenzial, MD-Simulationen zu beschleunigen und damit die Untersuchung komplexerer biologischer Prozesse zu ermöglichen, als bisher praktikabel war.

Acknowledgements

I begin by expressing my heartfelt gratitude to my supervisor Prof. Paolo Carloni for inspiring me to see the largeness of things and directing the course of my doctoral project. It is also not easy to fully describe the crucial help provided by my project manager Davide Mandelli, who was always available, eager to teach concepts of MD and support me in my research in every way, including proofreading and editing of texts. It should go without saying that without our successful co-working, we would not have the results that have been presented here. I also deeply thank my ex-project manager Viacheslav Bolnykh, who continued to discuss my work with me long after he left academia, thereby providing valuable insights and orthogonal viewpoints.

Naturally, my doctoral project would not have endured if not for the continuous support of my wife Annette Malapally, even during periods when she herself was shouldering a heavy workload. Especially in the final three months (she claims they were five), as I was gradually forced to reduce my duties towards my family to a minimum to complete the writing of this thesis. On a similar note, I thank my parents Kalpana C.S. and Ramesh Malapally, and my parents-in-law Manuela Göller and Reinhold Göller for helping out with the supervision of our daughter.

I would like to now thank many people who provided me with help at multiple points:

1. Mark Abraham, who helped me find my bearings in the GROMACS source code and answered many of my questions,
2. Georg Hager, who taught me (during my master's program and, later, doctoral project) how to do microbenchmarking properly,
3. Rolf Rabenseifner, who acquainted me with advanced MPI techniques and gave me an important hint to analysing the performance of the overlapping of computation and communication,
4. Ilya Zukhov, Jan-Oliver Mirus and the JSC team, who were always available to help researchers solve their supercomputer-related prob-

lems,

5. Peter Cordes, Jérôme Richard and Viktor Eijkhout on stackoverflow.com, for helping me understand many complex concepts,
6. Marta Devodier, for identifying problems in my pMD implementation, performing many additional test simulations and helping me analyze simulation data, and
7. Emiliano Ippoliti, for his readiness to answer questions about anything.

I also thank you, dear reader, for devoting your time and attention to reading this thesis.

Conventions

Throughout this thesis we use the following conventions.

1. If a technical term is supported by a definition in the text, it is marked by ✶. The reader can expect to find the definition in that page or an adjacent page, and the definition is set off in a coloured box. For example:

"The total force* acting on each atom is computed..."

DEFINITION : FORCE

A quantifiable theoretical construct in physics which is the cause of any change in the state of motion of material objects. Forces are responsible for starting, increasing/decreasing the rate of and stopping motion.

2. A term is *italicized* if it
   (a) comes with a definition (in a definition box as explained by the previous point),
   (b) is weakly defined i.e., it is a recurring concept that does not have a definition box, or,
   (c) serves to emphasize or highlight a particular sentence.
3. If a term is to be abbreviated for convenience, the abbreviation is set in parentheses immediately after it. If the term is a standard abbreviation or a name, the first letter of each word will be set in capital. If an abbreviation has not been expanded at all, please refer to the "List of Abbreviations".
4. Footnotes will be provided to give extra information, mention details, provide intuition and author's comments without interrupting the flow of the main text.

5. Figures will be made as self-sufficient as possible.
6. Source code and names of implementation details e.g., of variables, functions, classes etc., are written in typewriter-style text e.g., `myClass`. Similarly, code listings are written in C++ in the following format:

```
void exampleFunction()
{
    std::printf("Hello World!");
}
```

Listing 1: Sample Listing

The whole thesis is written in English.

Chapter 1

Introduction

As a computer experiment[1], the *Molecular Dynamics* (MD) simulation has become invaluable for its *predictive* power —its ability to accurately and consistently predict physical quantities that can be verified by future, real-world experiments [1, 2]. Since recent years, it has increasingly found application in the study of complex biological processes [3–6]. In this context, MD can not only predict the thermodynamics, but also, in contrast to other simulation techniques such as traditional *Monte Carlo* methods [1], the *kinetics* of the process under study. This property is crucial because biochemical events must often occur at the right time in complex cascades called signaling pathways. Therefore, a correct description of biological events requires the elucidation of their timescales. The residence time of ligands in their target receptors is hence a very important kinetic property in computer-aided drug design (CADD).

At least in theory, like any computer experiment, MD is *scalable*; its size parameters (atom count and simulated time) can be increased indefinitely, any number of simulations can be conducted simultaneously using multiple computers, and, most importantly, each simulation can be accelerated dramatically by harnessing the power of *High-Performance Computing* (HPC). However, the *time evolution* of the MD simulation is inherently sequential; the computational work associated with separate points in time cannot be done in parallel[2]. Moreover, MD relies heavily on statistical mechanics, one of the core tenets of which makes the accuracy of simulation contingent on its sampling an enormous number of distinct "snapshots" of the system. The combination of these two caveats results in the *time-scale problem*. It is difficult to overstate the severity of this requirement —to use MD as an effective tool, it is necessary to run very long simulations, which themselves

[1] If real-world experiments disagree with simulation, the model of simulation must be adapted. Then, using a sufficiently validated model, simulation can predict or confirm real-world phenomena.

[2] This would require the prediction of a future state in a causality-driven process.

take long to complete.

Added to this, the computational work of the MD simulation increases non-linearly with the atom count[3] and realistic descriptions of biological systems involve large atom counts [7]. Fortunately, the use of HPC systems has helped overcome this obstacle greatly; with the use of GPUs (and other specialized auxiliary processors), simulating protein systems in the microsecond time-scale is no longer a challenge and even the millisecond time-scale[4] has been attained [7, 8]. All the same, without methodological assistance, the *sampling efficiency* of conventional MD is typically low. Studying a protein's conformational[5] changes requires observing the (rare) event *multiple* times, which is made possible only by running extremely long simulations or multiple (very long) simulations. Further, events such as the unbinding of a drug-like molecule from its target, which is interesting from the perspective of residence time (which e.g., influences the therapeutic efficacy of the drug), could take several minutes, which puts studying them outside the reach of conventional MD altogether [7, 9].

As a result, it is not surprising that there is a very strong motivation to improve the performance of MD. On the one hand, we have creative techniques called *enhanced-sampling* methods that aim at extending the reach of MD by increasing its sampling efficiency. These methods attempt to sidestep the time-scale problem (to some extent) by cleverly and artificially increasing the probability of sampling the physical event of interest. However, when using such methods, retrieving the kinetics is generally not a straightforward task [10–12]. On the other hand, recent developments in HPC have brought *exascale* performance within our reach; bigger and increasingly more powerful computing systems are being (and have been) built to achieve an incredible billion billion operations per second. However, sheer hardware is not enough; we need scalable algorithms and well-implemented software to properly leverage the machinery and deliver even a fraction of the expected performance. Therefore, there is a growing awareness that a good approach to tackling the challenges at hand must attack both methodologically ("MD front") as well as algorithmically ("HPC front").

My doctoral project was conceptualized and realized within the interface between these two fronts, as a collaborative effort between the Institute of Neuroscience and Medicine (INM-9) and the Jülich Supercomputing Center (JSC) at the Forschungszentrum Jülich (FZJ), as part of the Supercomputing

[3] In the naïve implementation, simulating N mutually interacting atoms requires roughly N^2 computations per time step.

[4] Since the timestep is commonly specified in femtosecond, a millisecond is a *trillion* (10^{12}) time steps.

[5] A protein's conformation is a dynamic variation of its atomic structure realized by internal and environmental interactions that confers a biologically functional property to the protein.

and Modeling for the Human Brain (SMHB) project. Its objective was to improve the scalability of existing algorithms and implement new, highly scalable algorithms that enable efficient hardware resource utilization. In it, I have pursued two separate endeavours to attain my project goals:

1. Investigating the performance improvement of the *3D Discrete Fourier Transform* (3D DFT), which is a mathematical operation used in several instances in the context of MD —e.g., to compute long-range non-bonded interactions via Ewald-type methods and in plain wave basis set implementations of *Density Functional Theory* (DFT)-based *quantum mechanical* (QM) MD simulations. There is evidence to suggest that it becomes the performance bottleneck at high core counts in highly optimized MD codes [8].

2. Designing, implementing, testing and optimizing the software components of a massively parallel MD strategy that combines path-based MD with the enhanced-sampling technique of metadynamics, within the GROMACS code suite and the PLUMED library. The resulting algorithms have been benchmarked using up to 94% of the JUWELS Booster (3500 GPUs and 42,000 CPU cores) at the JSC, revealing an unprecedented scaling performance with $\geq$70% weak-scaling parallel efficiency.

The work of the second endeavour represents a way to overcome the time-scale problem, since it effectively converts it into a *size-scale problem* that closely resembles a type of perfectly scalable application. Moreover, it allows us to increase the sampling efficiency via metadynamics, with the valuable advantage that it is possible to retrieve the kinetics of the simulated process. Through several benchmarks and performance analyses, I have attempted to demonstrate the outstanding performance of my implementation and the dual-mode versatility of how it can be deployed (i.e., using OpenMP and MPI in the multicore/multinode setting and CUDA and MPI in the multi-GPU/multinode setting). Further, our correctness tests have established its basic feasibility and identified its strengths and limitations. Its practical feasibility in tackling a real-world research problem must now be ascertained by applying it to study a suitable problem.

1.1 Thesis Contents

Chapter 2 titled ”Introduction to Parallel Computing” provides the reader with reference material for the HPC topics that are relevant to the contents of this thesis. Similarly, chapter 3 titled ”Introduction to Simulations of Molecular Dynamics” introduces the basics of MD theory and highlights the exact contexts from within which the works of chapters 4 and 5 originate.

Chapter 4 titled ”Metadynamics of Paths” details the theory, algorithmic and implementation details, and the correctness and performance testing and results of the second research endeavour.

Chapter 5 titled ”Scalability of 3D DFT by Block-Tensor-Matrix Multiplication on the JUWELS Cluster” details the theory, algorithmic and implementation details, and the performance testing and results of the first research endeavour.

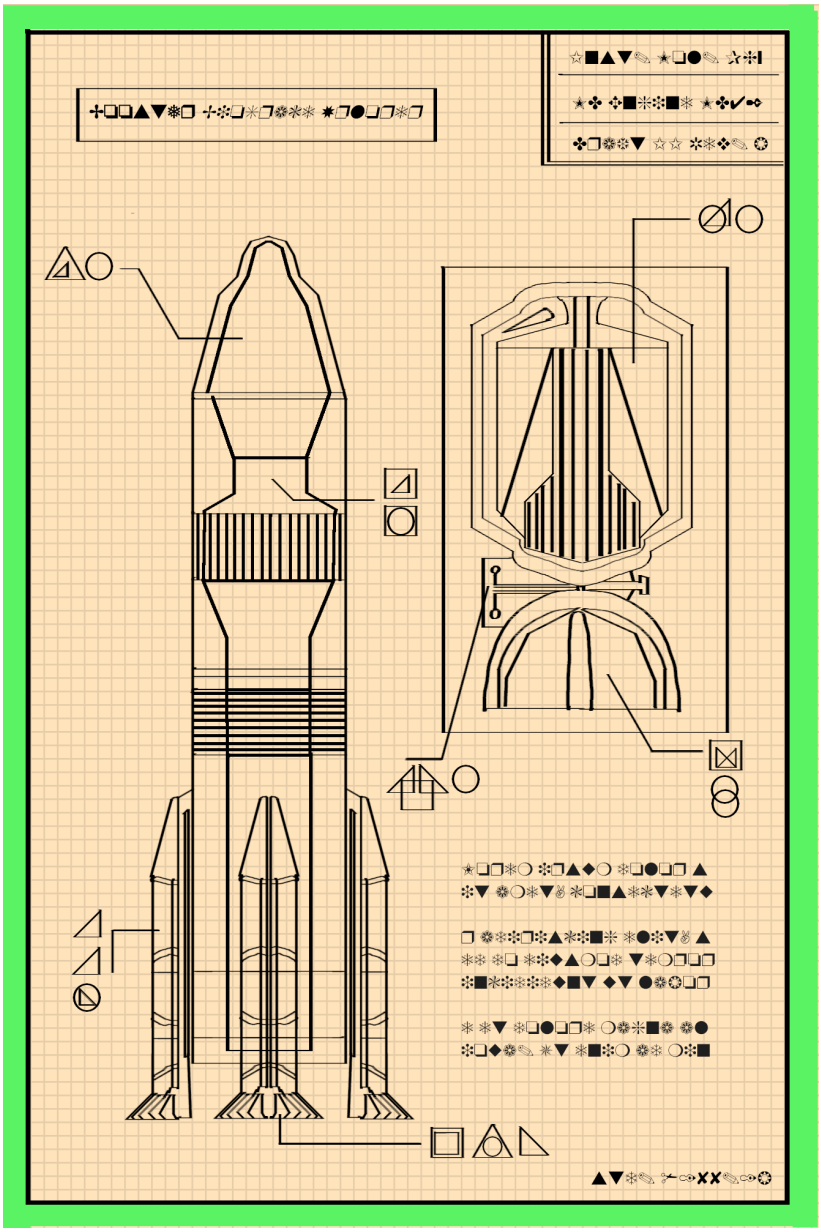
1.2 List of Publications

In accordance with §5(3) of the doctoral regulations extracts of this thesis have been published or submitted for publication in the following papers:

(A.) Nitin Malapally, Viacheslav Bolnykh, Estela Suarez, Paolo Carloni, Thomas Lippert and Davide Mandelli. Scalability of 3D DFT by Block-Tensor-Matrix Multiplication on the JUWELS Cluster. *Journal of Parallel and Distributed Computing*, 193:104945, 2024.

Publication (A.) corresponds to chapter 5, ”Scalability of 3D DFT by Block-Tensor-Matrix Multiplication on the JUWELS Cluster”. In accordance with §5(3) of the doctoral regulations I state my contribution to this publication:

(A.) In collaboration with Dr. Viacheslav Bolnykh, I conceptualized and designed the algorithm reported by the publication. I implemented the algorithm in the form of an open-source C++ library called S3DFT and validated it. In consultation with Dr. Viacheslav Bolnykh and Dr. Davide Mandelli, I conducted the primary investigation. I wrote the original draft and, along with my co-authors, participated in its review and editing.



Even extremely fast vehicles take forever to explore the confines of space

Chapter 2

Introduction to Parallel Computing

This monograph serves as the report of a doctoral project that was conceptualized to address the requirement of developing, implementing, testing and optimizing advanced algorithms that show good hardware resource utilization on multi-level-parallel clusters. Therefore, to make the contents of chapters 4 and 5 more accessible to the reader, in this chapter, we will present the basics of parallel computing. Further, we will discuss a powerful technique of performance analysis that was used extensively in our work.

This chapter is laid out as follows: the first section introduces modern approaches to computing, the second discusses multicore computers, the third illustrates the topic of parallelism from a theoretical viewpoint and the fourth presents a tool developed and used within the scope of this project that enables a unique approach to performance analysis.

2.1 Concepts of Modern Computing

In 1965, Gordon Moore, co-founder of Intel Corp., made an observation that the number of transistors on integrated circuits doubles every year. Ten years later, he revised this observation to a doubling approximately every two years. Remarkably, since then, this *de facto* law has held true and the transistor count on chips has grown exponentially. Even though increasing the transistor count introduces more complexity, processor designers have maintained pace by means of resourceful techniques collectively known as *Instruction-Level Parallelism* (ILP) and strategies such as caching and using instruction sets that enable more optimization [13]. These innovations have put computing performance more or less in direct relation to the transistor

count; not only has the transistor count grown exponentially till the early 2000s, but the computing performance has matched it [14].

In recent years, this trend has started to waver and processor designers are finding it harder to push the performance limits of processors. One reason is that there are many instances in which the instruction stream cannot make proper use of all available transistors, resulting in hardware under-utilization. Moreover, increased clock speed has proven to be an inevitable requirement to gaining performance, which unfortunately also results in increased heat generation and hence loss of efficiency [14]. Among other approaches to solving the problem of improving computing performance despite the above difficulties, the solution of multicore processing has emerged to be most promising, resulting in a paradigm shift that emphasizes multi-level parallelism [13, 14].

Parallelism begins at the level of the simplest sequential processor that is ascending in terms of transistor count-based complexity. How can such a processor ensure maximum hardware utilization to ultimately maximize its performance? To answer this question, we begin by presenting important concepts of modern computer systems that constitute the basics of HPC.

2.1.1 Von Neumann Architecture

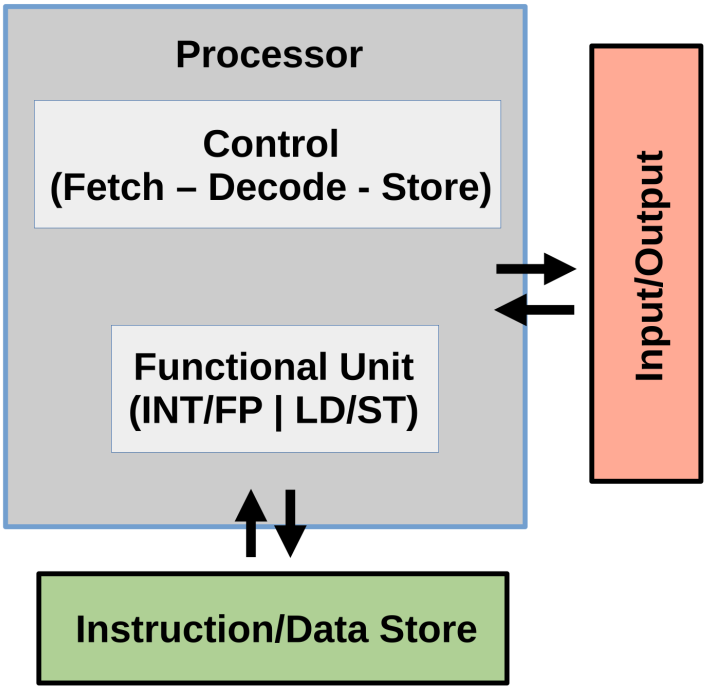


Figure 2.1: A figure illustrating the basic components of the von Neumann machine.

The term *von Neumann architecture* refers to a type of computer comprising a processor that executes programs, an undivided memory in which both programs and data are stored, and input/output modules. This concept, although historical, nevertheless offers a view on computing from a very high level of abstraction, and can thus be used to understand the vast majority of computer systems [15]. In fact, most modern computer architectures have evolved from it, distinguishing themselves from the theoretical von Neumann machine primarily by their technique of caching instructions and data separately, close to the processor.

The computational work of a program is done by *functional units* that specialize in integer and floating-point operations. The rest of the processor, which is the majority of it, consists of components that are responsible for the administrative logic that ultimately ensures that the functional units are uninterruptedly fed with instructions and operand data [13]. The cycle of operation can be summarized so [15]:

1. the next instruction is fetched from the store,
2. the instruction is decoded i.e., it is mapped to its corresponding functional units and its operands are identified and located,
3. the operand data is fetched from the store,
4. the operation is executed, and,
5. if required, the results are written back to the store.

In modern architectures like the x86 architecture, program code consisting of complex instructions is decomposed on the fly into much simpler micro-operations[1] that execute quickly, requiring only a few clock cycles to complete [13]. The advantage of this strategy is that techniques of ILP can then be applied to these micro-operations to optimize work on a very low level, resulting in better hardware utilization than if complex instructions were to be directly executed.

2.1.2 Instruction Pipelining

Since a single instruction may require multiple cycles[2] to be executed [15], a fully sequential *Single Instruction Stream, Single Data Stream* (SISD, as per Flynn's taxonomy [16]) machine will have an instruction throughput* less than one instruction per cycle.

[1] Intel calls them "μOps" and AMD calls them "rOps".

[2] For brevity, henceforth "clock" will be omitted when referring to clock cycles.

DEFINITION : INSTRUCTION THROUGHPUT

The number of instructions executed per cycle averaged over a period of time.

However, since instructions coming from an instruction stream may be unrelated to one another and could be processed by different functional units, they can be executed parallelly [14]. This ILP technique is referred to as instruction pipelining. Consider a simplified example in which execution involves the following steps:

1. fetch instruction,
2. decode instruction,
3. load operand 1,
4. load operand 2,
5. perform integer/floating-point operation 1 on operands 1 and 2,
6. perform integer/floating-point operation 2 on operands 1 and 2, and,
7. store the result.

In this example, assuming (for simplicity) that each step requires one cycle to complete, results are available every 7 cycles, as shown in figure 2.2.

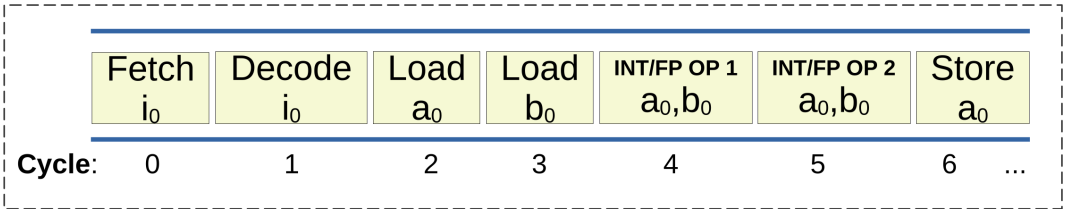


Figure 2.2: A figure showing the execution of a simplified example task involving seven steps, each requiring one cycle to be completed, performed sequentially. It can be seen that one instruction is executed every 7 cycles.

On the other hand, since instructions operating on different data are not dependent on each other, if the steps are executed parallelly in a pipeline, starting from cycle 5, results are available every cycle, as shown in figure 2.3.

In this way, using instruction pipelining can result in an ideal throughput of one instruction per cycle when the pipeline is full. However, pipelines are prone to "hazards", which can lead to *stalling* of the pipeline [17]; data and control hazards can cause the processor to delay the execution of subsequent instructions to resolve the hazard, resulting in pipeline "bubbles" that prevent the attainment of ideal throughput.

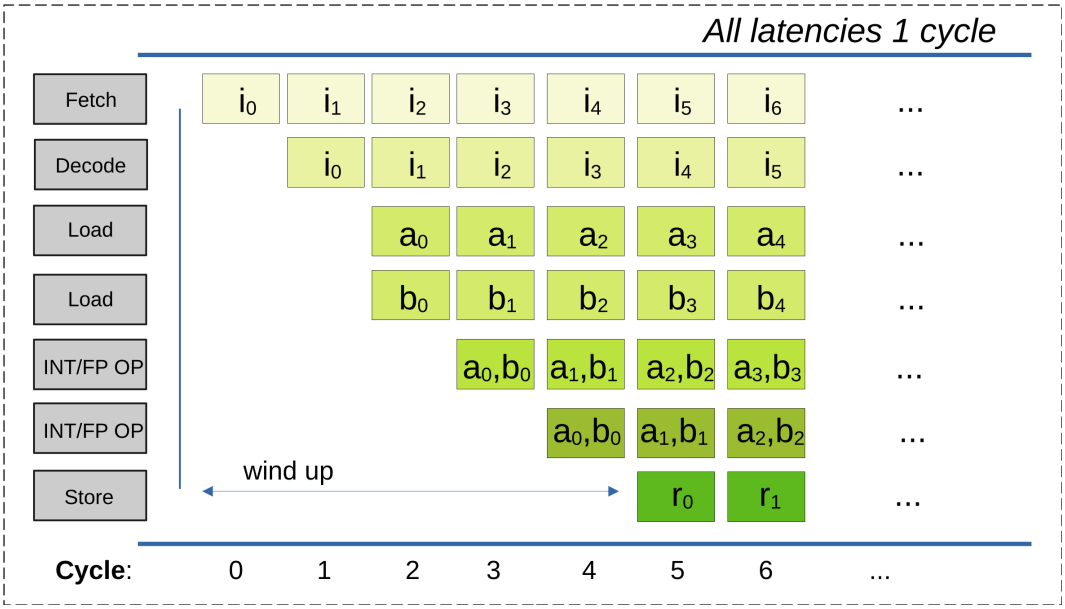


Figure 2.3: A figure showing the execution of a simplified example task involving seven steps, each requiring one cycle to be completed, performed using instruction pipelining. It can be seen that starting with cycle 5, one instruction is executed every cycle. Note that the figure depicts a replicated load unit, which is an example of superscalarity (see subsection 2.1.3).

2.1.3 Superscalarity

Superscalarity* is an important feature of modern CPUs that is used to boost instruction throughput beyond one instruction per cycle.

DEFINITION : SUPERSCALARITY

A microarchitectural technique that enables the issuing of multiple, independent instructions from a single instruction stream every cycle by utilizing replicated functional units.

Modern processors are usually 2-6 way superscalar [17], which means that (in combination with instruction pipelining) they can ideally execute that many instructions every cycle. To achieve this type of ILP, a superscalar processor incorporates replicated functional units, such as two or more floating-point units, arithmetic logic units and load/store units. Also, it makes use of an instruction dispatcher that reads instructions from a single instruction stream, dynamically resolves data dependencies and issues instructions to the appropriate execution unit. To fully exploit superscalarity, processors

must utilize out-of-order execution[3], optimized assembly code and instruction pipelining [13].

2.1.4 Vector Extensions

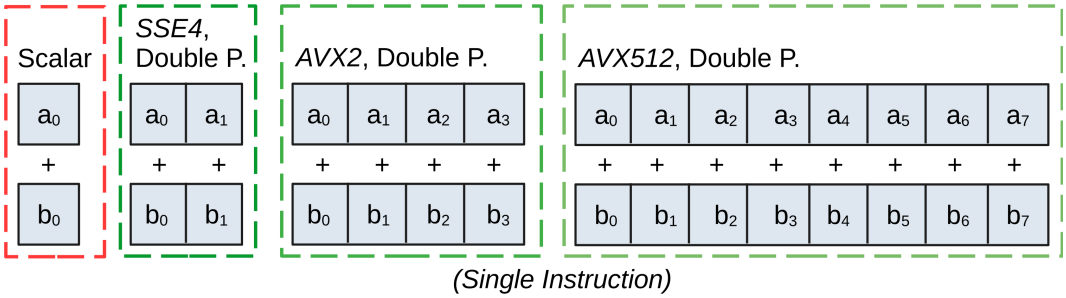


Figure 2.4: A figure contrasting scalar addition against Intel's SSE4, AVX2 and AVX512 vector additions. The scalar instruction operates on a single datum, whereas the vector instructions operate on different data in parallel.

Instructions that operate on single-datum operands are called scalar instructions. Prior to execution, their operands are fetched from memory and stored in units of temporary storage known as *registers* that are located very close to the required functional units. During execution, the functional units read the operands from the registers, perform the necessary operation and store the result in another register. After execution, the result may be written back to memory [13].

In contrast to scalar instructions, modern processors feature *vector* functional units capable of executing vector instructions, which operate on multiple operand data in parallel [14]. This approach is referred to as *Single Instruction stream, Multiple Data stream* (SIMD) as per Flynn's taxonomy [16]. The execution of vector instructions is made possible by "wide" registers that can typically store 128-512 bits of operand or result data. Before execution, multiple operand data i.e., 16-64 8-bit, 4-16 32-bit or 2-8 64-bit (integer, single- or double-precision floating-point) values are fetched and "packed" into such registers. Following this, a single instruction is executed that operates on the packed data simultaneously (see figure 2.4 for an illustration). Finally, the results may be written back to memory or retained for further processing.

[3] An ILP technique by which instructions from a single instruction stream may be executed in a different order than they appear in the program code, based on the availability of operand data and functional units.

DEFINITION : VECTORIZATION

The process of optimizing a numerical computation to exploit SIMD parallelism, by ensuring the generation of vector instructions either by the compiler (referred to as *auto*-vectorization) via compiler flags and pragma directives, or by manually writing vector instructions via inline assembly or compiler intrinsics.

DEFINITION : DATA LOCALITY

Temporal data locality refers to the frequency of access of given data within a short period of time. Spatial data locality refers to the proximity of given data to some other related data as determined by the data structure's organization in memory and the algorithmic data access pattern.

The use of vector instructions can boost peak performance considerably, especially when the memory system is fast enough to keep the vector units busy [13]. Therefore, *vectorization** becomes invaluable when *data locality** is given, which is often the case in programs that perform intensive numerical work in nested loops. Thankfully, most modern compilers are very good at auto-vectorizing most loops. If a compiler is unable to auto-vectorize (which could happen if it is unable to establish the absence of data dependencies), manual vectorization must be performed, which requires specialized, in-depth knowledge towards efficiently manipulating data using vector instructions. It is important to note that the degree to which vectorization can be efficiently performed is ultimately dependent on both the hardware and the nature of the numerical computation.

2.1.5 Multithreading

For modern processors, instruction pipelining (see subsection 2.1.2) is crucial for efficiency and high performance, which is the reason why most modern processors are "superpipelined" i.e., deeply pipelined [14, 17]. However, a variety of factors such as poorly generated assembly code, insufficient loop size and failed branch prediction[4] can cause bubbles in pipelines [13]. At the same time, the *nature* of the instruction stream from which the instructions are drawn into the pipeline may not allow further parallelism.

[4] An ILP technique by which processors attempt to anticipate the outcome of conditional branching prior to their resolution using either simple, static assumptions or complex, dynamically gathered statistical information, to prevent pipeline stalling.

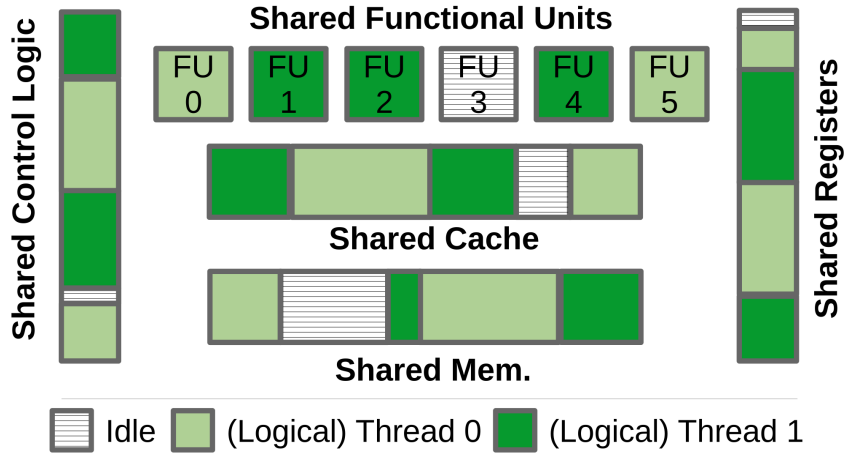


Figure 2.5: A figure illustrating the concept of SMT at a high level of abstraction using a simplified example with two threads. Although the hardware components of a processor (and its associated memory) are not replicated, multiple logical instances of a core are given access to them to improve the hardware utilization.

To overcome the efficiency problem caused by pipeline bubbles, a different mode of parallelism referred to as *Thread-Level Parallelism* (TLP) can be employed, in which the *logical*[5] architectural state of a single *core*⋆ is replicated [13]. This allows the processor to execute multiple instruction streams *concurrently*⋆ , behaving as though it contains more processors within it than it actually might. It is crucial to note that each of these instances of execution, referred to as a *thread*⋆ , is a purely logical entity with the processor core being its hardware counterpart.

Definition : Core

A subset of the components of a processor including control logic, registers, functional units and cache memory that is capable of fetching, decoding, and executing instructions on its own.

Definition : Thread

A light-weight, logical unit of execution that maintains its own execution context viz., general/special-purpose registers, program counter (also called instruction pointer) and stack.

[5] "logical" is a commonly used term in computer science taken to refer to entities on a higher level of abstraction than hardware. For example, variables are logical instances whose hardware counterparts are registers.

DEFINITION : CONCURRENCY

Concurrency is the ability of a computer to do multiple units of work in an overlapping period of time. Since this concept is not bound by the requirement of simultaneous execution, it distinguishes itself from parallelism in that the work may be performed intermittently by sharing hardware resources.

Multithreading refers to a set of techniques that implement TLP for the concurrent execution of instructions belonging to distinct execution contexts. To increase the hardware utilization efficiency and ultimately the instruction throughput, multithreading often requires multiple threads to share the hardware resources of a single core. This can be achieved by switching the context rapidly (fine-grained multithreading) or whenever a pipeline stalls (coarse-grained multithreading)[14]. A more advanced technique that has asserted itself is *Simultaneous MultiThreading* (SMT), in which fine-grained multithreading is combined with superscalarity and dynamic scheduling [14] to allow concurrent execution of threads, as illustrated by the simplified example in figure 2.5. However, it must be noted that highly optimized code may not benefit from SMT, especially if threads require the *same* hardware resources.

Ultimately, the common strategy of these techniques is to ”pop” pipeline bubbles and keep as many transistors simultaneously busy as possible. Of course, TLP can also be applied to multicore processors such that a single thread is run on each core. In this case, the objective becomes changed; the emphasis is no longer on improving hardware utilization efficiency but rather to enable truly parallel (as opposed to ”merely” concurrent) processing, typically leading to much greater speedups, provided the program’s algorithm exposes sufficient parallelism.

Threads serve as a construct to divide the work of a program for concurrent execution. However, unlike in ILP, the concurrency of multithreading is typically not automatically safeguarded by hardware or software, making it the responsibility of the programmer to ensure that the conditions for correct execution are met. Therefore, when using multithreading, the programmer must ensure the absence of data dependencies and synchronize the activity of the threads whenever necessary.

2.1.6 Memory System

Processor idling is inevitable if the processor cannot be constantly fed with data [17]. This makes the memory system a critical limiter of runtime performance [13].

The *data transfer rate** from the memory to the processor is very slow compared to the *computation rate** of the processor. For example, let us consider a simple array addition given by `c[i] = a[i] + b[i]`. To perform each loop iteration, one datum each of `a`, `b` and `c` has to be fetched from memory. When the data is available, one floating-point computation viz., one addition, can be performed. Finally, the result datum `c[i]` has to be written back to memory. Assuming that each datum is sized 4 B, this means that for every GFLOP/s of performance, we would need 16 GB/s of bandwidth. Now, even desktop processors of recent years are capable of peak performances in the range of hundreds to thousands of GFLOP/s[6]. Even for a processor with a relatively humble peak performance at 100 GFLOP/s, we would still need the extremely high memory bandwidth of 1.6 TB/s, whereas the peak memory bandwidth of today's consumer-grade computers is usually $\sim$ 50 GB/s and of server-grade computers $\sim$ 200 GB/s.

Definition : Data Transfer Rate (Bandwidth)

The amount of data transferred by a data path in unit time, typically measured in GB/s (10^9 B/s) or GiB/s (1024^3 B/s).

Definition : Computation Rate (Performance)

The number of floating-point operations (FLOPs) that a processor can effectively perform in unit time, typically measured in GFLOP/s (10^9 FLOP/s). The peak performance can be estimated as [21]:

$$P_{peak} = \text{Instructions Per Cycle} * \text{Max. Clock Frequency} \tag{2.1}$$

This relative slowness of the memory system with respect to the processor is not only historical but also relevant for the future because processors are projected to get faster while memory bandwidth is expected to fail to keep pace [13]. Using the insight that typical data access patterns offer data locality (see definition provided in subsection 2.1.4), memory systems mitigate the performance effect of slow memory access by utilizing fast (albeit small), temporary data storage located on the processor chip referred to as cache memory.

Modern processors commonly have up to three levels of cache memory with increasing access speeds and decreasing sizes, with the last level cache being 4-6 times as fast as the main memory [13]. The level one (L1) cache is usually

[6]For example, Intel's mid-2010s' Haswell processors (Intel Celeron® [18] to Core i7™ [19]) combining AVX2 and *fused multiply-add* (FMA) units on two ports [20], offer between $\sim$140 GFLOP/s and $\sim$760 GFLOP/s.

split into two, L1I for storing instructions and L1D for storing data. If data is unavailable in L1, the processor looks for it in the L2 and L3 caches, one after the other. Finally, if the data could not be found in the caches, it must be fetched from main memory.

To ensure that data is readily available in the caches, data is fetched in the form of contiguous *cache lines* of fixed size[7]. Moreover, a technique called prefetching is used to supply the caches with data prior to their actual requirement by observing and anticipating the current access pattern. Prefetching could be done by hardware, which limits its use to a limited set of access patterns implemented by the hardware prefetcher, or by software, in which case the programmer or compiler must use the required instructions at the right time in code to initiate proper prefetching [17].

2.2 Multicore Computers

2.2.1 Multicore Processors

As the name suggests, a multicore processor incorporates more than one core on a single chip. Usually, each core has private L1 and L2 caches, and it shares the L3 cache with the other cores. Since copies of a cache line can exist in the private caches of multiple cores, consistency between cached copies and the corresponding data in the main memory must be assured. This is most often done by hardware in compliance with a cache-coherence protocol [13].

Since all cores share the same logical view of the data in memory i.e., all cores take a given memory address to point to the same physical location in memory, we refer to such a system as a *shared-memory* computer. However, although all cores[8] of a multicore processor access a unified logical memory, the processor may be based on *Uniform Memory Access*⋆ (UMA) or *Non-Uniform Memory Access*⋆ (NUMA) architectures. See figures 2.6 and 2.7 for simplified examples illustrating the concepts of UMA and NUMA.

DEFINITION : UNIFORM MEMORY ACCESS

A shared-memory architecture in which there exists a single main memory in relation to which all processors have the same latency and bandwidth.

[7] The most common cache-line size is 64 B i.e., long enough to include 16 single-precision or 8 double-precision floating-point values in a single transfer.

[8] Here, ”cores” can also be taken to mean ”processors” because even multisocket setups i.e., with multiple (possibly multicore) processors, can be operated as shared-memory computers by connecting the processors via so-called *interconnects*.

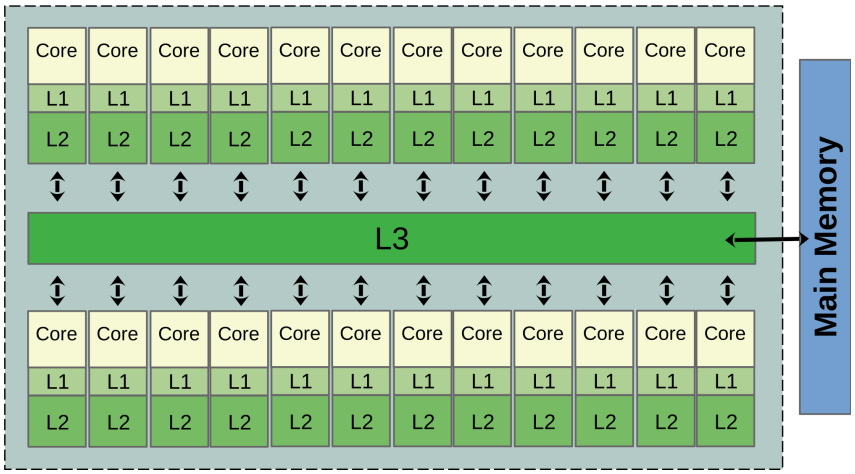


Figure 2.6: A figure showing a simplified layout of the Intel Xeon® Platinum 8168 processor based on UMA architecture.

DEFINITION : NON-UNIFORM MEMORY ACCESS

A shared-memory architecture in which the main memory is divided amongst the system's processor cores such that subsets of cores have their own dedicated memory units, resulting in varying latencies and bandwidths depending on which core accesses which memory. Memory access involving low latency and high bandwidth is referred to as *local access*, whereas memory access involving high latency and low bandwidth is referred to as *non-local access*. The NUMA architecture is a strategy to combat the exacerbated memory bottleneck caused by cores competing against each other for memory access.

Shared-memory computing references a unified address space, which greatly simplifies writing parallel programs in this paradigm. The most commonly used implementation of shared-memory parallelism is multithreading (see subsection 2.1.5). However, as we've highlighted previously, the onus of writing correct multithreaded programs is on the programmer, requiring them to be aware of common pitfalls such as race conditions* and false sharing* and how to avoid them e.g., using atomic sections when writing to shared data and using padding in the data structure, respectively, among numerous other strategies. Moreover, programmers developing applications that may run on NUMA-based systems must avoid writing code that makes non-local accesses as much as possible, since non-local accesses require using interconnects, which makes them distinctly slower than local accesses.

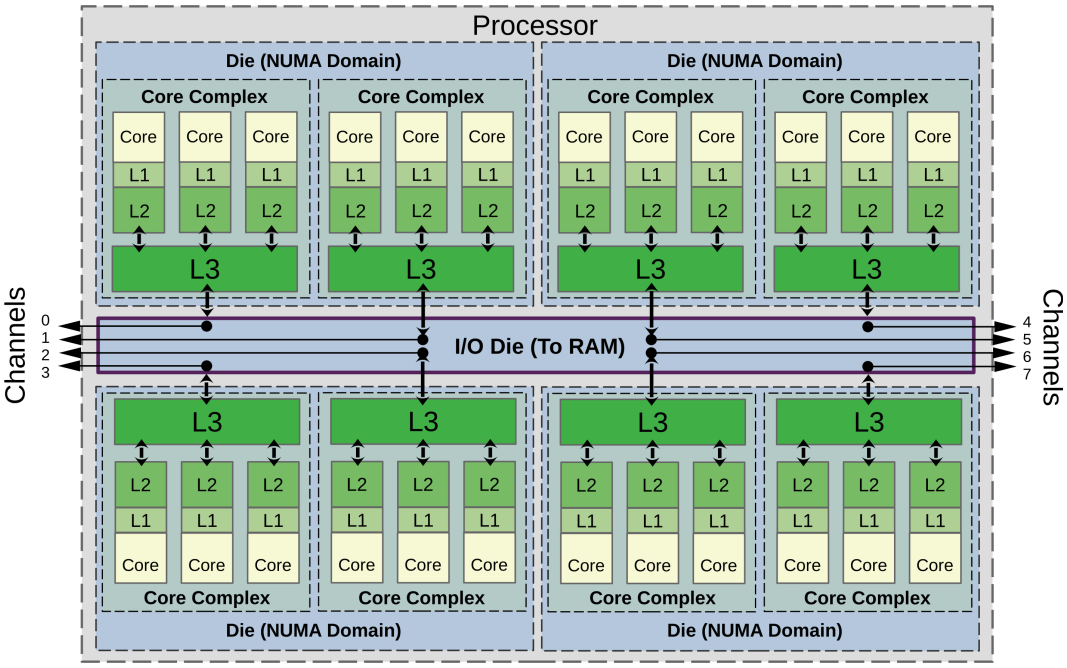


Figure 2.7: A figure showing a simplified layout of the AMD EPYC 7402 processor with NUMA architecture. Here, the NPS4 setting is portrayed, by which the processor has 4 NUMA domains, each with 2 memory channels.

Definition : Race Condition

Erroneous behaviour of a parallel program in which concurrent data accesses occur with at least one write access, resulting in an inconsistent state. Race conditions can occur when the programmer does not taken measures to guarantee either the correct order of accesses or exclusive write accesses in a context in which concurrent execution is possible.

Definition : False Sharing

Undesirable behaviour of a shared-memory parallel program in which multiple threads repeatedly modify (possibly different) data from the same cache line causing that cache line to be invalidated, evicted and reloaded to cache frequently (as imposed by the cache-coherence protocol) resulting thereby in degraded performance.

2.2.2 Graphics Processing Units (GPUs)

In scientific computing, the use of GPUs is widespread. Due to the speedups they offer over CPUs and their low power consumption, their popularity is increasing [22]. In fact, nine of the top 10 most powerful supercomputers owe their peak performance figures to their use of GPUs. When the application is well-suited for processing by GPUs and the program is well-optimized, a single modern GPU can reach 14 TFLOP/s [23], making it 4-6 times as powerful as their modern, high-performing CPU counterparts.

On a very abstract level, the key difference between CPUs and GPUs is that the former is task-parallel whereas the latter is data-parallel [24]. In other words, GPUs perform especially well when the same operations are to be done on different data simultaneously [22]. As the name suggests, GPUs were originally conceptualized to perform graphics-based processing e.g., texture mapping, pixel shading, vertex calculations [25] etc., and as such, made good use of SIMD parallelism. With the release of NVIDIA's Tesla architecture (see figure 2.8 for a simplified layout) in 2006, the hitherto-employed fixed-purpose pipelines were replaced by universal *Streaming Multiprocessors* (SM) containing general-purpose scalar functional units referred to as SM cores[9]. Moreover, the new architecture combined SIMD parallelism with fine-grained SMT into a new mode of parallelism NVIDIA refers to as Single Instruction Multiple Threads (SIMT). In SIMT, the SM cores host a large number of threads simultaneously (SMT aspect), with each thread executing the same instruction on different data (SIMD aspect), in groups called warps [26]. By switching contexts when warps stall (possibly due to their waiting for data from memory) and letting other warps continue execution, GPUs attempt to increase hardware resource utilization and hide memory latency.

A potential bottleneck when using GPUs in combination with CPUs is the low bandwidth of the PCI Express connection between them [8]. This makes the use of GPUs feasible either when the latency of data transfer is small in relation to the potential speedup, when data transfers can be overlapped with computations or when the data transfers are infrequent. Added to this, the programmer must be aware that the problem data size must be sufficiently large to fully exploit a GPU.

2.2.3 Supercomputers

Large-scale scientific projects such as climate simulations, simulations of the human brain and artificial intelligence-based research require large-scale computational machines commonly referred to as supercomputers [27–29].

[9] Since these "cores" do not independently decode instructions, they are fundamentally different from the CPU core as defined in subsection 2.1.5.

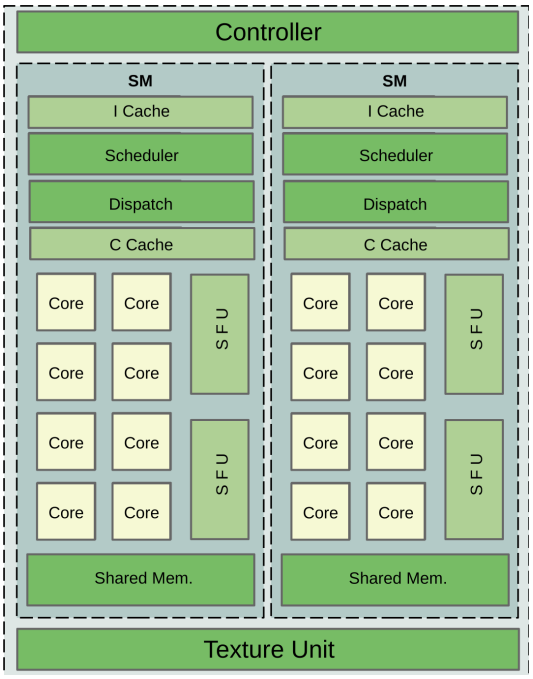


Figure 2.8: A figure showing a simplified layout of a Texture/Processor Cluster in NVIDIA's Tesla GPU architecture. A Tesla-based GPU contains an array of multiple such independent units. Here, "Core" denotes SM cores, "I Cache" denotes the instruction cache, "C Cache" a read-only cache and "SFU" represents a special-function unit. This image was created with the help of descriptions provided by ref. [25] and [26].

A supercomputer or HPC cluster is a computing system that aims to achieve its peak performance by leveraging multi-level parallelism across thousands of interconnected *processing elements*⋆ (PEs). It is the result of extrapolating on-chip parallelism into an increasingly multi-chip context. Therefore, moving from a single computer to a cluster, the most important performance metric becomes *scalability*⋆ .

DEFINITION : PROCESSING ELEMENT

An abstraction referring to any device capable of performing computations such as CPUs, GPUs, Many Integrated Core (MIC) chips, Accelerated Processing Units (APUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs) etc.

DEFINITION : SCALABILITY

The capacity of the system to efficiently add hardware resources towards either solving a computational problem faster or solving larger problems in the same approximate time. Here, "system" can refer to either the code or the computer. Scalability is typically expressed in terms of a performance metric and the amount of hardware resources used to achieve it.

The peak performance of supercomputers has increased from the terascale (1,000 GFLOP/s) in the early 2000's to the petascale (1 million GFLOP/s) in the early 2010's and entered the exascale (1 billion GFLOP/s) in the early 2020's [30]. Today, the most powerful supercomputer is El Capitan at the Lawrence Livermore National Laboratory, USA, with ∼11 million cores, recorded to have reached 1.7 EFLOP/s [31]. On a lower scale, another example is the JUWELS supercomputer at the Jülich Supercomputing Centre (JSC) [32], Germany. Its Cluster module has 114,480 cores, was ranked the 44th most powerful supercomputer by the end of 2020 [33] and is currently ranked 180. Its Booster module has 449,280 cores, was ranked 7 by the end of 2020 and is currently ranked 33 at 44.12 PFLOP/s [34]. These examples should serve to give the reader a glimpse of the level of scalable performance that one can expect in recent times.

Compute Node

The prefix "*super-*" hints at the supercomputer's composition of subordinate computers. Indeed, supercomputers are assemblages of building blocks called *compute nodes* that are connected to each other by means of a network. The compute node brings together multiple PEs and connects them to the network via a network interface card. These are typically one or more multicore CPUs and one or more accelerators such as GPUs. For example, the compute node of the Cluster module of the JUWELS supercomputer has two 24-core Intel Xeon® Platinum 8168 chips in NUMA configuration (see figure 2.6 for a simplified layout of a single chip). The compute node of the Booster has two 24-core AMD EPYC™Rome 7402 chips (see figure 2.7 for a simplified layout of a single chip) along with four NVIDIA Ampere A100 GPUs.

Distributed-Memory Parallelism

The physical memory of a supercomputer is available *separately* in the compute nodes, each of which is connected to the others via the network. However, as seen in the case of NUMA architectures, a physical separation in memory must not necessarily imply a separation in its logical rep-

resentation. Nevertheless, although shared-memory models for this setup are available [14], large-scale supercomputers almost exclusively employ a distributed-memory model [13]. Unlike in shared-memory computing (see subsection 2.2.1), in distributed-memory computing, the logical representation of the total memory is distributed [15]; a memory address to one PE does not point to a meaningful physical location in another PE's memory. Hence, distributed-memory computing is characterized by *message passing* between PEs to transfer data as per requirement. Historically, the programming model put forth by the *Message-Passing Interface* (MPI) standard has asserted itself in scientific computing communities ubiquitously, thereby becoming the most commonly used programming approach to distributed-memory computing.

In MPI's programming model, each logical unit of execution (analogous to the thread of shared-memory computing) is the *process*, which is an instance of a program that is granted its own memory space, execution environment and access to system resources by the operating system. Each MPI process, referred to as an MPI *task*, is capable of communicating with other MPI tasks within a given MPI *communicator*, a construct that helps isolate message-passing events to mutually isolated contextual spaces. In this way, MPI tasks can belong to multiple communicators, in each of which they are given a unique MPI *rank*. The model offers flexibility and defines a feature-rich and broad *Application Programming Interface* (API) to help programmers implement a variety of message-passing techniques. Although it is intended for distributed-memory computing, it is not uncommon to see MPI-parallel programs in the shared-memory context as well.

Network

The description of the connections between compute nodes, possibly through intermediate devices such as switches, encompassing all aspects of inter-node communication is referred to as the network's *topology*. The choice of the network topology significantly influences the system's scalability and performance, and it is motivated by the requirements of the supercomputer viz., high bandwidth, fault tolerance, cost-effective scalability etc [35].

A network can connect compute nodes either directly (such as in mesh networks) or indirectly through switches. Switched networks are widely employed [14]. In switched networks such as fat-tree and dragonfly networks, the compute nodes are divided into groups. Due to their grouping, the time of communication is not the same for all communicating pairs [13]. A metric to measure the communication latency between two communicating elements in a switched network is the "hop" count viz., the number of switches a message has to cross on its path between them [14]. As a result, with the

hop count, the communication time increases and the communication rate decreases.

A programmer developing software for HPC applications can benefit from acquainting themselves with the behaviour of the network of their target supercomputer. This can be done by running microbenchmark programs implementing well-known communication patterns at different node-count scales to develop an intuition for the effective communication rate at those scales. Further, knowledge about the network topology can be used to guide them to look for and use options provided by the workload/cluster management system to realize specific test- and use-cases, such as allocation of contiguous nodes, setting of a maximum switch count limit etc.

2.3 Theory of Parallel Performance

2.3.1 Strong Scaling

Although the objective of parallelism is to speed up work, while *parallelizing**, we focus rather on engaging multiple workers to work at the same rate instead of accelerating the rate of work of any worker.

DEFINITION : PARALLELIZATION

The process of converting a procedure executable by one worker into a *new* procedure that attains the same objectives faster by allowing execution by multiple workers (if the original procedure does not already allow it).

Let us assume that one worker requires unit time to perform the work of a particular procedure. In an ideal world, assigning n identical workers to the same procedure would complete the work in $\frac{1}{n}$ time. Then, we could keep assigning workers until the work would take no time to complete. However, in reality, we can observe that (at least) three factors prevent us from realizing perfect parallelization:

- there might be some parts of the procedure that can only be performed by any one worker (serial component),
- the workers might need to communicate with one another (during which time they may not be able to work) and
- they might need to *synchronize**.

DEFINITION : SYNCHRONIZATION

The requirement in a parallelized procedure that one or more workers must coordinate their activity with other workers i.e., wait for them to finish certain tasks, wait for their turn to access shared resources etc., so that the procedure can be conducted in its correct sequence of operations and maintain internal consistency to attain its objectives. As a result of the unavoidable waiting, synchronization is associated with an overhead that reduces parallelism.

DEFINITION : SPEEDUP

The factor by which a parallel procedure is faster than its serial analogue (i.e., the parallel procedure using one worker) for a problem of fixed size. This is given by $S = \frac{P_N}{P_1}$ where P_N is the performance of the parallel procedure using N workers and P_1 the performance of its serial analogue. Using the performance metric $P = \frac{1}{T}$ where T is the runtime, the speedup is given by

$$S = \frac{T_1}{T_N}.$$

We begin by analysing the *speedup*⋆ of parallelization by only considering the effect of the serial component of the work. If s is the fraction of the time corresponding to the serial component and p the fraction corresponding to that component of the work that can be perfectly parallelized, the speedup is given by

$$S = \frac{s+p}{s+\frac{p}{N}}$$

Now, since we've assumed that the work is only composed of a serial and a parallel component, we have $s + p = 1$, which implies

$$S = \frac{1}{s+\frac{1-s}{N}}. \tag{2.2}$$

In the above equation, it is clear that if $N \gg 1$, $S \approx \frac{1}{s}$. This is Amdahl's law [13–15], also referred to as the *law of diminishing returns*; as the number of workers increases, the speedup continuously decreases to converge to a constant value determined solely by the serial component. In other words, once the *parallel limit* has been reached, engaging additional workers becomes practically useless. Beyond the limit, performance can only be further improved by speeding up the serial component of the work.

If the parallel procedure involves communication, the combination of the multilateral nature of communication and the uncontrolled parallelism of the procedure imposes a synchronization requirement; when it is necessary for a group of workers to communicate, all workers in that group must wait for those workers that are not yet ready to participate in the communication. Consequently, there is a reduction in parallelism and a corresponding reduction in speedup. Moreover, the act of communicating itself is time consuming, which further increases the time of the parallel procedure. Left and right subfigures in figure 2.9 demonstrate Amdahl's law for fictitious parallel procedures without and with these additional overheads, respectively.

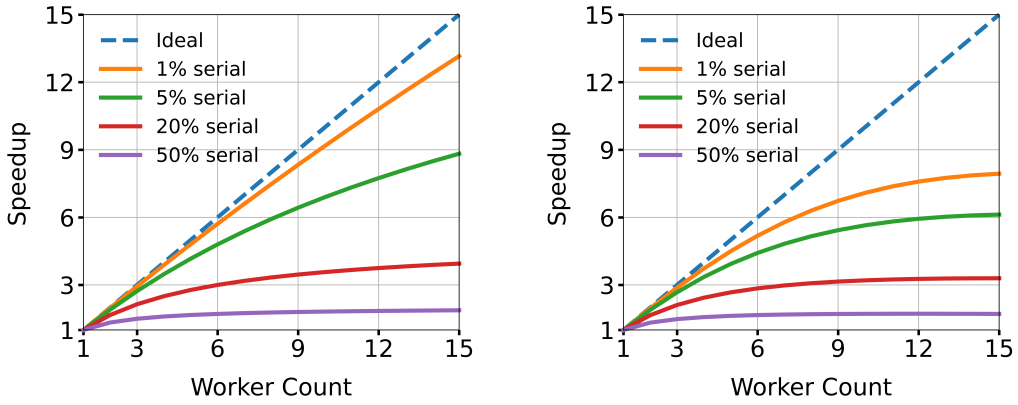


Figure 2.9: Figures illustrating the strong-scaling performance of fictitious parallel procedures with varying serial components. In the left subfigure, it is assumed that no overheads apply. In the right subfigure, the time fraction of the combined communication and synchronization overhead is modeled by a function that linearly interpolates between 0% at $N = 1$ to 5% at $N = 15$.

Definition : Strong Scaling

A mode of scaling in which additional workers are used to speed up a fixed quantity of work.

Amdahl's law explains the *strong-scaling** behaviour of parallel procedures. However, rather than as a realistic model of parallel execution, it should be understood as a source of insight yielding the concept of the parallel limit. Therefore, the serial/parallel time fraction is rarely computed during performance analyses. Instead, the *efficiency* of strong scaling (sometimes, simply "parallel efficiency") is used to quantify scaling performance. It is

calculated as

$$E(N) = \frac{S}{N}. \tag{2.3}$$

Rigorously justifying the identification of a procedure's *practical* parallel limit is not a trivial task because useful speedups can still be had at low parallel efficiencies. Therefore, it boils down to a decision made on a cost-vs-benefit basis. Some conventions simply consider the limit to be reached at 50% parallel efficiency. This gives us the (rather arbitrary) categorization of acceptable scaling when $50\% \leq E < 75\%$ and excellent scaling when $E \geq 75\%$.

2.3.2 Weak Scaling

The scaling performance of a parallel procedure must be considered based ultimately on its use case. In the previous subsection, we saw that strong-scaling analysis is conducted when the goal of parallelization is to perform work as fast as possible. However, the scope of parallelization is not exclusively limited to increasing the speed of work; in many cases, it is to be able to do more work in the same time. This use case is referred to as *weak scaling** .

DEFINITION : WEAK SCALING

A mode of scaling in which additional workers are used to complete more work in the same time.

In the case of weak scaling, we start with one worker and some quantity of work. As we start scaling up, the quantity of work is not maintained constant as in the case of strong scaling, it is instead also scaled up —ideally such that the quantity of work per worker remains constant. If this requirement is met, each worker will take the same time to complete its task. Therefore, the time will neither drop nor increase at higher worker counts. The time plot of a perfectly weak-scaling application (showing time in the y-axis and worker count in the x-axis) features a straight line at a particular time.

In reality, however, as in the case of strong scaling, communication and synchronization overheads are unavoidable, resulting commonly in a weak-scaling plot with a time curve that gradually increases as it progresses along the x-axis. Once again, the weak-scaling performance can be quantified by means of efficiency:

$$E(N) = \frac{T_1}{T_N}, \tag{2.4}$$

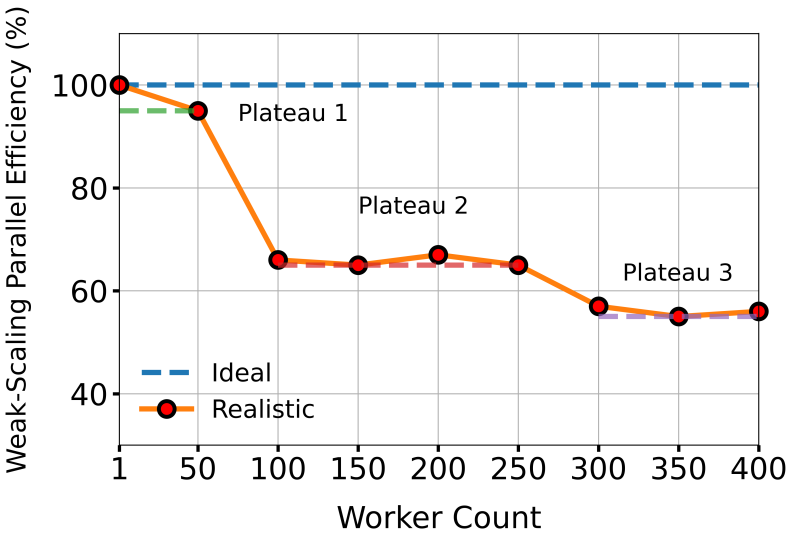


Figure 2.10: A figure comparing the weak-scaling parallel efficiency of the ideal case to that of a realistic case.

where T_1 is the time of the single-worker case and T_N the time of the N-worker case. See figure 2.10 for a comparison between the ideal weak-scaling efficiency plot and a realistic one.

2.4 Semi-Empirical Roofline Analysis

Until now, we have used this chapter to introduce the reader to the basics of parallel computing. Now, we will discuss a performance analysis model and a practical technique aimed at simplifying its application. Additionally, we will present a library that we've implemented to help us conduct perform optimization: by analysing performance, deducing the next direction of optimization and implementing the optimization in a cycle until satisfactory results are obtained. The reader is advised to look at appendix subsection A.1 for a tutorial to using this library.

2.4.1 Introduction to the Roofline Model

The roofline model provides essential insights for analyzing the performance of software components by transforming the naïve question, "How well does my code perform?" into one that orientates us towards a theoretical footing i.e., "How well *can* my code perform?". It directs us to look at the nature of the code as a deciding factor of attainable performance, connecting it with machine capacity to establish a theoretical performance upper bound —the roofline. Based on this, the basic *direction* of necessary optimization (if optimization is at all possible) can be discovered.

The performance of a particular code depends both on the properties of the code and on the properties of the target machine. These properties are captured by the concepts of *code balance* [13] and *machine balance* [36], respectively. From the viewpoint of measuring the performance of the code, both concepts represent *a priori* information.

The code balance answers the question, "How much data must be moved along a data path to perform a single floating-point operation?",

$$B_c = \frac{\text{Data traffic (B)}}{\text{Work (FLOP)}}. \tag{2.5}$$

The machine balance answers the question, "In the time it takes for one floating-point operation, how much data can *this* computer transfer?",

$$B_m = \frac{\text{Peak Bandwidth (GB/s)}}{\text{Peak performance (GFLOP/s)}} = \frac{b_{\text{max}}}{P_{\text{max}}}. \tag{2.6}$$

Importantly, these concepts highlight the possibility of data movement and computational work being *separate*, potential performance limiters. Now, we can estimate the peak performance by factoring in both the code's demands and the capabilities of the machine [13]

$$P_{\text{roof}} = \min\left(\text{P}_{\text{max}}, \text{P}_{\text{max}}\frac{B_m}{B_c}\right) = \min\left(\text{P}_{\text{max}}, \frac{b_{\text{max}}}{B_c}\right). \tag{2.7}$$

The code balance can be estimated solely by inspecting the code, whereas it is left to the analyst to ascertain P_{max} and b_{max} by referring to technical datasheets published by hardware vendors. A theoretical estimate for P_{max} can be calculated using equation (2.1) (from "Computation Rate (Performance)" p. 16) if the instruction with the highest throughput and the number of its available independent pipelines are known. Using utilities that benchmark the memory bandwidth (which are sometimes provided by vendors e.g., Intel MLC), b_{max} can be obtained. Applying different machine balances corresponding to different data paths of the machine, the various roofs are often reported on a 2D chart with *arithmetic intensity* (given by $I = \frac{1}{B_c}$) along the X-axis and the performance along the Y-axis as shown in figure 2.11.

If the bandwidth actually attained by the code during its execution is known, it is possible to estimate the corresponding performance as follows

$$P_{\text{attained}} = \frac{b_{\text{attained}}}{B_c}. \tag{2.8}$$

Alternatively, it can be measured by instrumenting the code with performance counters and running it in the target setting. Once the performances

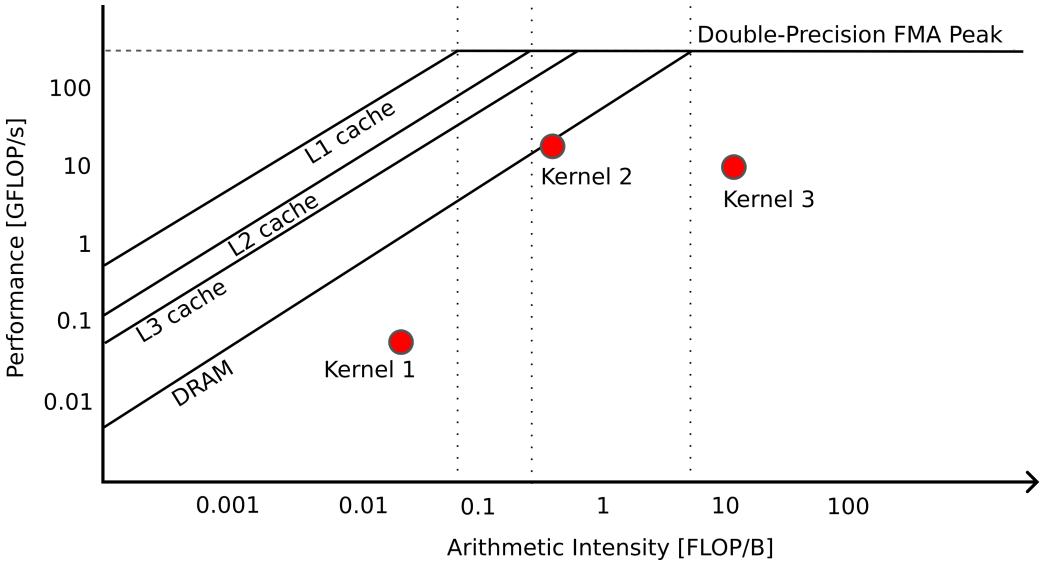


Figure 2.11: Example roofline chart illustrating the rooflines for the data paths of the memory and the caches. The red circles depict the target functions (i.e., functions under analysis).

are known, the target functions can be plotted in the roofline chart as depicted by the red circles in the example in figure 2.11. In the example, the analyst can conclude that kernels 1 and 3 are performing sub-optimally, whereas kernel 2 has already attained its peak performance and does not require further optimization in code. Moreover, it can be seen that kernels 1 and 2 are *memory bound** whereas kernel 3 is *core bound** .

Definition : Memory-Bound Code

Code whose performance can only be improved by improving the memory bandwidth utilization. On a roofline chart, it is located under the slanted roof.

Definition : Core-Bound Code

Code whose performance can only be improved by improving the processor utilization. On a roofline chart, it is located under the horizontal roof.

Knowing the roof and knowing whether the code is memory- or code-bound is crucial for the analyst to decide the next steps of optimization. Without this knowledge, they might

1. stop optimizing before the code is optimally performing,
2. continue attempting optimization when the code is already optimally performing, or,
3. attempt to improve memory/core utilization when the code is core/memory-bound.

Therefore, the roofline model acts as a "compass" of optimization for the analyst navigating the (2D) "sea of performance".

2.4.2 A Semi-Empirical Approach

While it is possible to measure b_{attained} from equation (2.8) using performance counters, it can be replaced by a value that is calculated using the runtime and the data traffic requirement of the code as

$$b_{\text{effective}} = \frac{\text{Data traffic required (GB)}}{\text{Time (s)}}. \tag{2.9}$$

In this way, we have deduced an *effective bandwidth*[10] which can be used to estimate a corresponding *effective performance* in the same fashion as in equation (2.7), as

$$P_{\text{effective}} = \frac{b_{\text{effective}}}{B_c} = \frac{\text{Work required (GFLOP)}}{\text{Time (s)}}. \tag{2.10}$$

To help the reader gain a solid understanding of how data traffic and computational work requirements (and hence, code balance) can be estimated using *a priori* information contained in the code, we have dedicated appendix subsection A.1 to providing real-world examples in which the semi-empirical roofline analysis is conducted.

We consider the above to be a semi-empirical approach because it does not involve actually measuring either the data traffic or the computational work, relying instead only on a code balance model and a reliable measurement of the runtime of the code. In a fully empirical approach, performance must be measured, which requires using performance counters via tools that are hardware- and operating system-dependent. Instrumenting the code, configuring the tool and analyzing the measured information[11] is often not straightforward. On top of that, hardware performance counters can sometimes be unreliable. The semi-empirical approach represents an alternative to them, especially one that is portable, relatively easy to apply, easy to interpret and reliable.

[10] "effective" is used to indicate the fact that the bandwidth is not actually measured.

[11] For example, measurements relating to cache events are notoriously difficult to analyze.

2.4.3 TiXL: Timed eXperiments in Loops

At the beginning of this doctoral project, inspired by the possibility of a semi-empirical approach to performance analysis, we conceptualized a simple library that could perform *microbenchmarking*⋆ by means of an intuitive and easy-to-use API. The result of this inspiration is the open-source C++ library, TiXL [37]. Throughout the course of this doctoral project and for all the work of which the results have been reported in this thesis, TiXL was used to conduct semi-empirical roofline analysis. Later in the project, a profiler module was also developed and integrated[12].

DEFINITION : MICROBENCHMARK

A microbenchmark program (or simply, microbenchmark) is designed to measure the performance of a single software component in isolation. A high-quality microbenchmark reliably measures simple and well-defined quantities such as the runtime. It can be used to eliminate the influences of peripheral software components that unavoidably interact with the component of interest during system-level tests.

In practice, writing a valid microbenchmark can be tricky and error prone, with many factors influencing the accuracy of measurement. TiXL takes care of some of these matters internally.

- **Dynamic frequency scaling**: Modern microprocessors regulate their clock frequencies dynamically to conserve energy and/or prevent overheating. However, microbenchmarking assumes a constant and high clock frequency, which cannot be immediately guaranteed. As a countermeasure, TiXL calls the target function in a loop to properly "warm up" the system, to simulate the conditions under which the target function will be intensively executed.
- **Choice of clock**: TiXL uses wallclock time to measure the runtime of the target function. Wallclock time is a better choice for microbenchmarking than CPU clock time because it measures contributions from input/output, context switches etc. On the one hand, measuring wallclock time will undoubtedly result in the introduction of noise in the measurements. On the other hand, since those effects are also present in the target setting, it can be argued that they should be reported.
- **Experimental robustness**: The runtime environment is unpredictable due to complex background activity resulting in measurement noise. This leads to outliers in the microbenchmarking measurements. To

[12] A tool used to identify *hot spots* i.e., components of code that account for the major fractions of the runtime.

combat measuring *atypical* noise, TiXL performs many experiments and computes statistical details (such as minimum, maximum and average time, standard deviation etc.) using the samples. Using this information, the experiment count can be adjusted until the standard deviation is acceptably low[13].

- **Treatment of argument data before each experiment**: If the argument data of the target function is located in the caches when the timed experiment is conducted, it is impossible to measure certain quantities e.g., the memory bandwidth, which requires the data to be available *only* in the main memory. To ensure this, each timed experiment should allocate data afresh and take special measures to evict the data from the caches. TiXL enables this through its API design.

The main interface of TiXL's microbenchmarking module (which makes *cold microbenchmarking*⋆ its goal[14]) is an abstract class that requires the user to implement three functions:

```
class experiment_functor
{
public:
    virtual void init() = 0;
    virtual void perform_experiment() = 0;
    virtual void finish() = 0;
};
```

The first should allocate argument data and apply pre-experiment treatment on it, the second should call the target function and the third should deallocate argument data and reset its state.

DEFINITION : COLD MICROBENCHMARKING

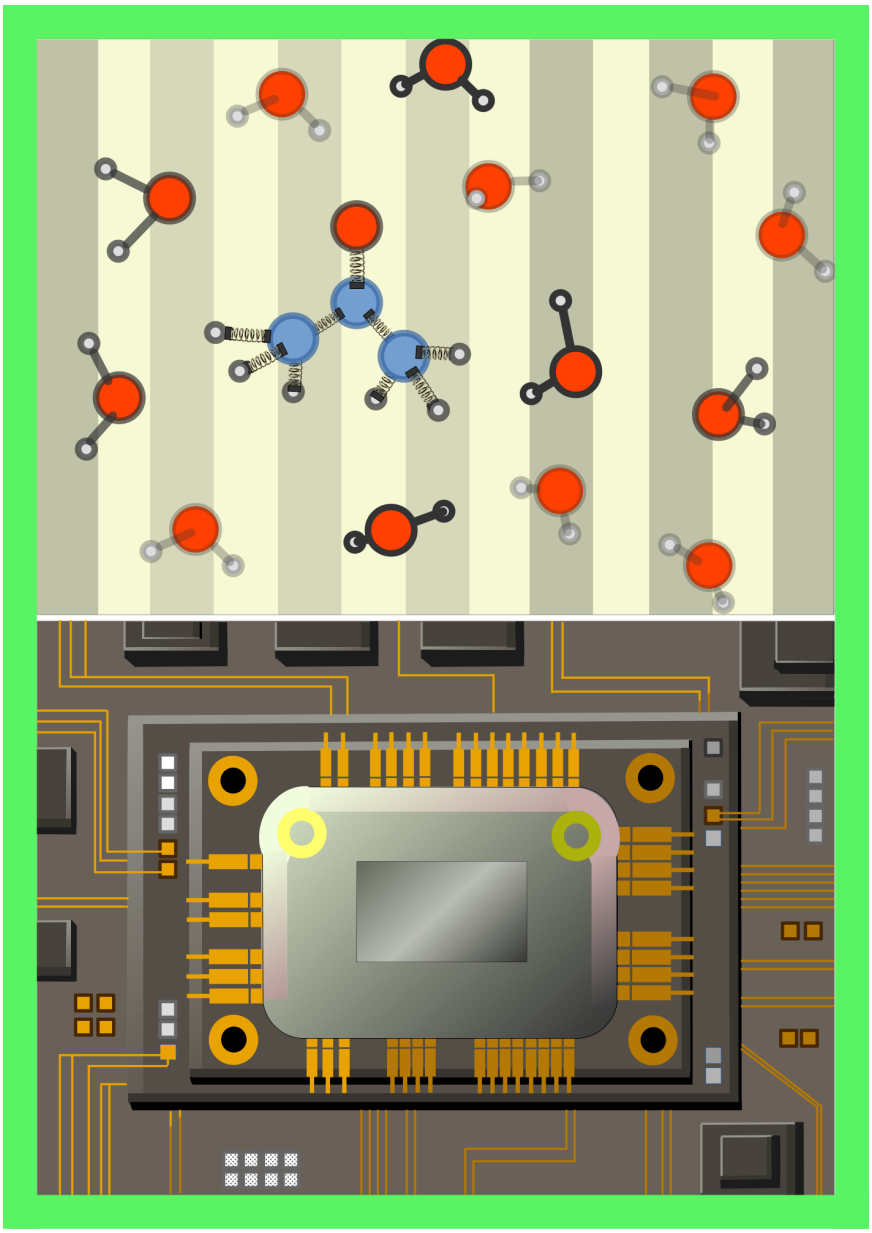
A type of microbenchmarking in which measures are taken to ensure the presence of operand data *only* in the main memory at the start of each experiment.

Having implemented this class, the user can call the function `perform_experiments` (for MPI support, `mpi_perform_experiments`) to perform the microbenchmarking. Once the experiments have been conducted, the function `write_measurements`

[13] A low standard deviation may not be enough to ensure high-quality benchmarking. We recommend writing the measurements to a file and plotting them to visually inspect them.

[14] Perfect cold microbenchmarking is very hard, since data notoriously clings to caches. One way is by using the compiler intrinsic `_mm_clflush`, which causes the compiler to emit the CLFLUSH instruction to evict cache lines to memory.

can be used to write the measured times to an output file for plotting. Further, the function `compute_statistics` (for MPI support, `mpi_compute_statistics`) can be used to compute the statistics of the measured times and the function `output_results` can be used to print the statistics to console output.



Naturally arranged atoms arranging atoms to find out how atoms move.

Chapter 3

Introduction to Simulations of Molecular Dynamics

As MD is at the overarching objective of this doctoral project, it is first necessary to discuss basic theoretical concepts of MD to provide the context of the original work reported in chapters 4 and 5. Hence, we will devote this chapter to that purpose and pinpoint the exact contexts (in subsection 3.2.2 and section 3.4) of the work reported in the above-mentioned chapters.

This chapter is organized as follows. We begin with the basic theory of *classical MD* (cMD), followed by a discussion of the modeling of potential energy. Next, we present core ideas of QM-based MD and the *quantum-mechanical/molecular-mechanical* (QM/MM) hybrid strategy and introduce foundational concepts of statistical mechanics. Finally, we conclude by discussing the motivation of enhanced-sampling techniques.

3.1 Basics of Classical Molecular Dynamics

3.1.1 Roots in Classical Physics

The concept of the atomistic cMD[1] simulation is based on the assumption that Newtonian dynamics can be applied to study the motion of atomic nuclei [38]. By iteratively solving time-discretized Newton's equations of motions, a cMD simulation generates a *trajectory* as a sequence of atomic *configurations** in time.

[1] Henceforth, any occurrences of cMD/MD are to be taken to mean *atomistic* cMD/MD.

DEFINITION : CONFIGURATION

A specific arrangement of atoms belonging to a system represented by a position vector $\mathbf{R} \in \mathbb{R}^{3N}$ where N is the number of atoms. Alternatively, $\mathbf{R} = \begin{pmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \\ \dots \\ \mathbf{r}_Q \end{pmatrix}$, where each vector $\mathbf{r}_i \in \mathbb{R}^3$ represents the position coordinates of one atom.

We begin by establishing the roots of cMD in physics and presenting how cMD simulations can be realized using computers. In a system of N atoms with position coordinates $\mathbf{R} \in \mathbb{R}^{3N}$ and atomic masses m_i for $1 \leq i \leq N$, we have the total energy

$$E_{\text{tot}} = \frac{1}{2}\dot{\mathbf{R}}^\intercal M \dot{\mathbf{R}} + U(\mathbf{R}), \tag{3.1}$$

where the first term captures the kinetic energy and the second term, the potential energy. The block diagonal mass tensor $M \in \mathbb{R}^{3N \times 3N}$ is given by

$$M = \begin{bmatrix} \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_1 & 0 \\ 0 & 0 & m_1 \end{bmatrix} & 0 & \dots & 0 \\ 0 & \begin{bmatrix} m_2 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_2 \end{bmatrix} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \begin{bmatrix} m_N & 0 & 0 \\ 0 & m_N & 0 \\ 0 & 0 & m_N \end{bmatrix} \end{bmatrix}.$$

Now, if this system is governed by energy-conserving dynamics, we have $\frac{dE_{\text{tot}}}{dt} = 0$, therefore,

$$\frac{dE_{\text{tot}}}{dt} = \frac{1}{2}\ddot{\mathbf{R}}^\intercal M \dot{\mathbf{R}} + \frac{1}{2}\dot{\mathbf{R}}^\intercal M \ddot{\mathbf{R}} + \nabla U(\mathbf{R})^\intercal \dot{\mathbf{R}} = 0,$$

where $\dot{\mathbf{R}} = \frac{d\mathbf{R}}{dt}$. Since M is a symmetric square matrix, this results in

$$\begin{aligned} \ddot{\mathbf{R}}^\intercal M \dot{\mathbf{R}} + \nabla U(\mathbf{R})^\intercal \dot{\mathbf{R}} &= 0, \\ M\ddot{\mathbf{R}} + \nabla U(\mathbf{R}) &= 0 \\ M\ddot{\mathbf{R}} &= -\nabla U(\mathbf{R}) \\ \implies \mathbf{F} &= -\nabla U(\mathbf{R}), \end{aligned} \tag{3.2}$$

which yields the equation for a *conservative* force. In the context of MD, equation (3.1) has been traditionally formulated as the *Hamiltonian* given by

$$\mathcal{H}(\mathbf{R}, \mathbf{P}) = \frac{1}{2}\mathbf{P}^\intercal M^{-1} \mathbf{P} + U(\mathbf{R}), \tag{3.3}$$

where $\mathbf{P} \in \mathbb{R}^{3N}, \mathbf{P} = M\dot{\mathbf{R}}$ is the momentum. Hamilton's equations of motion are equivalent to Newton's equations and are given as

$$\dot{\mathbf{R}} = \frac{\partial \mathcal{H}}{\partial \mathbf{P}}, \quad \dot{\mathbf{P}} = -\frac{\partial \mathcal{H}}{\partial \mathbf{R}}. \tag{3.4}$$

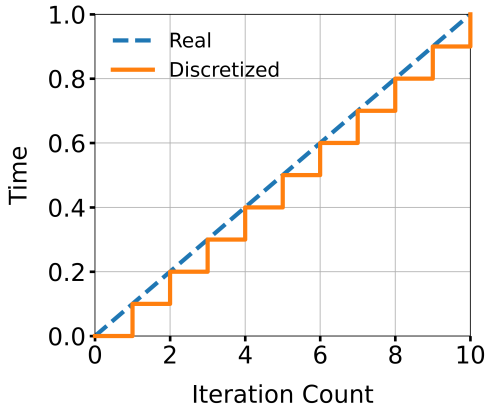


Figure 3.1: Figure illustrating the concept of time discretization in a computer simulation. Time seems to progress in the form of "steps" in a staircase, thus inspiring the term timestep.

Equation (3.2) expresses time-independent (i.e., static) quantities. When we consider the dynamic problem of simulating the *motion* of atoms, it is necessary to replace $\mathbf{R}$ with $\mathbf{R}(t)$, which gives us

$$\mathbf{F}(\mathbf{R}; t) = -\nabla U(\mathbf{R}; t). \tag{3.5}$$

DEFINITION : TIMESTEP

The unit of time used to approximate the smallest possible duration of an MD simulation.

An MD simulation iteratively solves equation (3.5) by discretizing time (see figure 3.1 for an illustration) using *timesteps** , which must be chosen by the user to be small enough such that even the fastest modes of the system conform with equation (3.5) [8]. This procedure is referred to as the *time evolution* of the system. An integration scheme is required, which uses the force vectors to compute the new position vector for the subsequent iteration. These schemes are typically derived from the Taylor series

$$\mathbf{R}(t) = \sum_{n=0}^{\infty} \frac{\mathbf{R}^{(n)}(t-1)}{n!} \Delta t,$$

where $\mathbf{R}^{(n)} = \frac{d^n\mathbf{R}}{dt^n}$ and Δt is the timestep. Since infinitely differentiable functions cannot be represented by computers, we settle for k-term approximations

$$\mathbf{R}(t) = \sum_{n=0}^{k} \frac{\mathbf{R}^{(n)}(t-1)}{n!}\Delta t + \mathcal{O}(\Delta t^{k+1}),$$

which introduce an error represented by the last term. Two widely used integration schemes based on approximate Taylor expansions are the leapfrog [39] and the velocity-verlet schemes [40, 41]. Although quite similar, the former computes half-step velocities and full-step positions

$$\mathbf{V}(t+\frac{1}{2}\Delta t) = \mathbf{V}(t-\frac{1}{2}\Delta t) + \Delta t M^{-1}\mathbf{F}(t)$$
$$\mathbf{R}(t+\Delta t) = \mathbf{R}(t) + \Delta t \mathbf{V}(t+\frac{1}{2}\Delta t)$$

whereas the latter computes full-step positions and velocities

$$\mathbf{R}(t+\Delta t) = \mathbf{R}(t) + \Delta t\mathbf{V}(t) + \frac{1}{2}\Delta t^2 M^{-1}\mathbf{F}(t)$$
$$\mathbf{V}(t+\Delta t) = \mathbf{V}(t) + \frac{1}{2}\Delta t M^{-1}\left[\mathbf{F}(t) + \mathbf{F}(t+\Delta t)\right].$$

The velocity-verlet integration scheme is typically more accurate when a temperature/pressure coupling is used [39].

3.1.2 Potential Energy Function

A *potential energy function** (PEF) is computed as the sum of contributions of multiple modes

$$E = \phi_1 + \phi_2 + \phi_3 + ...,$$

where ϕ_1 is the sum of potential energies that are *not* due to atomic interactions (e.g., external potentials) and ϕ_n the sum of potential energies due to n-atom interactions [42].

DEFINITION : POTENTIAL ENERGY FUNCTION

A mathematical model that calculates the potential energy of a system based on the atomic spatial arrangement.

The terms of the PEF fall into the categories of *bonded* and *non-bonded* interactions [43, 44]. The former is primarily used to model covalent bonds and is further classified as follows:

1. bonds (two-atom),

2. angles (three-atom) and
3. proper/improper dihedral angles (four-atom),

whereas the latter is used to model long-range interactions such as

1. Van der Waal's forces and
2. electrostatic forces.

A good PEF not only faithfully captures the behaviour of a system's potential energy, it is also computationally efficient as it is intended ultimately to extend the timescale of MD simulations [45]. However, the PEF is only the first prerequisite for conducting realistic cMD simulations. It provides the foundational model, which is open ended in that it leaves many parameters undetermined. Therefore, the second prerequisite is the parameter set, which refers to the values of all the parameters required to implement the model. The parameterization is commonly very elaborate [46], fitted using empirical data from physics and chemistry, and QM calculations [43, 46].

DEFINITION : FORCE FIELD

A combination of a PEF and a parameter set.

The *force field** is a central concept to cMD. Some commonly used force fields in biophysics are [47]:

1. AMBER —proteins, RNA and DNA,
2. CHARMM —proteins, nucleic acids and lipids,
3. GROMOS —proteins, nucleotides, and sugars, and,
4. OPLS —condensed phase simulations and biomolecules.

Additionally, since recently, many machine learning-based force fields such as GEMS [48], CHARMM-NN [49] and BIGDML [50] have gained popularity for providing accuracy comparable to QM-based MD.

A given force field may treat an atom very specifically to its context or chemical environment, e.g., oxygen in a water molecule may be parameterized differently from oxygen in a carbonyl group [51]. Making ad hoc changes to a force field's parameters or choosing the wrong force field can result in drastic, unpredictable changes, leading to incorrect simulations [52, 53][2]. Consequently, a well-informed (at best, tested for the specific use case)

[2]When this happens, the scientist becomes acutely aware of the fine line between conducting valid cMD simulations and playing computer games.

choice of force fields is a crucial pre-step to rigorous scientific study via simulation.

3.1.3 Bonded Interactions

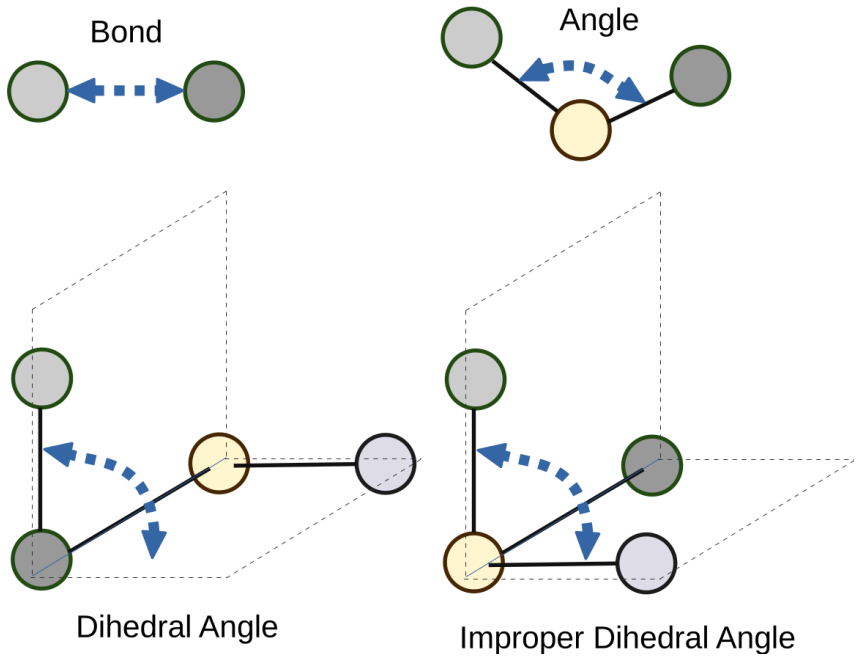


Figure 3.2: Figure showing commonly used bonded force fields —bonds, angles and dihedral angles.

It is generally sufficient to model bonded interactions using simple harmonic PEFs [44]. Two-atom interactions are modeled as linear springs,

$$U_{\text{bond}}(\mathbf{r}_\alpha, \mathbf{r}_\beta) = \frac{1}{2} k_{\alpha\beta}(||\mathbf{r}_\beta - \mathbf{r}_\alpha|| - r_e)^2,$$

where parameters $k_{\alpha\beta}$ and r_e are the stiffness coefficient and equilibrium length, respectively. Three-atom interactions are modeled as angular springs as

$$U_{\text{angle}}(\theta) = \frac{1}{2} k_\theta(\theta - \theta_e)^2,$$

where parameters k_θ and θ_e are the stiffness coefficient and equilibrium angle, respectively. Four-body interactions are modeled as (proper) dihedral angles and improper dihedral angles [8]. The former is modeled as a multi-term periodic expansion,

$$U_{\text{dihedral}}(\omega) = \frac{1}{2} \sum_m k_m(1 + cos(m\omega - \omega_m)),$$

where parameters k_m and ω_m are the stiffness coefficients and equilibrium angles, respectively. Improper dihedral angles are modeled as angular springs:

$$U_{\text{dihedral}}(\xi) = \frac{1}{2} k_\xi (\xi - \xi_e)^2.$$

These models are illustrated by figure 3.2.

The above are the most commonly used models. Apart from them, there are a variety of other models, such as the anharmonic Morse model [54], non-linear cubic models [55, 56] and the harmonic Urey-Bradley model [57].

3.1.4 Non-Bonded Interactions

Van der Waal's interactions are most commonly modeled using the *Lennard-Jones* (LJ) potential, the Buckingham potential and the Exponential-6 potential [58, 59]. The first is the most commonly used and is given by

$$U_{LJ} = \sum_{i=0}^{N} \sum_{j>i}^{N} 4\varepsilon \left[(\frac{\sigma}{||\mathbf{R}_i - \mathbf{R}_j||})^{12} - (\frac{\sigma}{||\mathbf{R}_i - \mathbf{R}_j||})^{6} \right] \tag{3.6}$$

where σ and ε are parameters. Electrostatic interactions are computed as per Coulomb's law as

$$U_C = \sum_{i=0}^{N} \sum_{j>i}^{N} \frac{q_i q_j}{4\pi\varepsilon ||\mathbf{R}_i - \mathbf{R}_j||}, \tag{3.7}$$

where parameters q_i and q_j are charges, and ε is the permittivity.

Computing the above interactions requires $\mathcal{O}(N^2)$ computations and it represents the most computationally intensive work in an MD simulation. To mitigate the computational costs, spherical/cubic truncation methods are widely employed, in which interactions between pairs separated by any distance greater than a cut-off value are ignored [60–62]. However, it has been observed that e.g., when simulating polar fluids, cut-off techniques result in strongly changed orientational structures [60], dielectric properties and translational and rotational motions [63]. Avoiding these effects seems to be only possible by providing a full treatment. Making matters worse, to reduce boundary effects, PBCs are also applied to mimic a very large system, which makes computing the interactions between all pairs even more expensive [64].

As a result, techniques that are derivatives or variants of the Ewald summation method are extensively used, which make use of the Fourier transform to not only ensure the convergence of the calculation but to also speed it up [61, 65]. The general approach is to decompose the calculation into part-calculations: one in the real space and the other in the Fourier (or reciprocal) space, each of which converges quickly.

3.1.5 Ewald-Type Methods

We will now provide a brief overview of the basic technique employed by variants of the Ewald summation using the specific example of the Coulomb interaction[3]. Consider a three dimensional lattice of points $L = \{\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, ..., \mathbf{r}_N\}$, with corresponding point charges $q_1, q_2, q_3, ..., q_N$. The electrostatic potential field at coordinates $\mathbf{r}$ with respect to a single charge is given by

$$\phi_i(\mathbf{r}) = \frac{1}{4\pi\varepsilon} \frac{q_i}{||\mathbf{r} - \mathbf{r}_i||}, \tag{3.8}$$

where ε is the permittivity constant. The potential energy contribution of a charge q_α at $\mathbf{r}_\alpha$ in the electric field of another charge q_β at $\mathbf{r}_\beta$ is,

$$U(\mathbf{r}_\alpha, \mathbf{r}_\beta) = q_\alpha \phi_\beta(\mathbf{r}_\alpha) = \frac{1}{4\pi\varepsilon} \frac{q_\alpha q_\beta}{||\mathbf{r}_\alpha - \mathbf{r}_\beta||},$$

using which, we get the potential energy of the lattice by summing up all possible interactions as

$$E = \frac{1}{4\pi\varepsilon} \sum_{i=1}^{N} \sum_{j>i}^{N} \frac{q_i q_j}{||\mathbf{r}_j - \mathbf{r}_i||}. \tag{3.9}$$

More often than not, MD simulations make use of PBC as shown in figure 3.3. Hence, we will consider an infinite lattice composed of cells, each of which is a repetition of L. This is a set:

$$L_\infty = \{L \cup L^{(1)} \cup L^{(2)} \cup L^{(3)} \cup ...\},$$
$$L_\infty = \{\mathbf{r}_\lambda \in \mathbb{R}^3 \mid \mathbf{r}_\lambda = \mathbf{r} + R_\lambda \mathbf{p}_\lambda \; \forall \, \mathbf{r} \in L, \; \forall \, \lambda \in \mathbb{Z}\},$$

where $R_\lambda \in \mathbb{R}^{3\times3}$ represents a rotation matrix and $\mathbf{p}_\lambda \in \mathbb{Z}^3$ represents the repetition vector, both of which are uniquely associated with the λ-th cell. For convenience, we define a function $\mathbf{e}_c : (\mathbb{Z}^+, \mathbb{Z}^+) \to L_\infty$ that allows us to enumerate the coordinates within each cell such that

$$\mathbf{e}_c(i, j) = \mathbf{r}_j \in L^{(i)}.$$

Using this, we can now adapt equation 3.9 for the infinite lattice so:

$$U_\infty = \frac{1}{4\pi\varepsilon} \frac{1}{2} \sum_{n_1=0}^{\infty} \sum_{n_2=0}^{\infty} \sum_{i=1}^{N} \sum_{j=1}^{N}{}' \frac{q_i q_j}{||\mathbf{e}_c(n_1, j) - \mathbf{e}_c(n_2, i)||},$$

where $'$ is used to denote the exclusion of the self-interaction term for which $i = j$ when $n_1 = n_2$. The calculation of this infinite sum converges not only

[3]The same technique can be used for the Van der Waal's interaction as well.

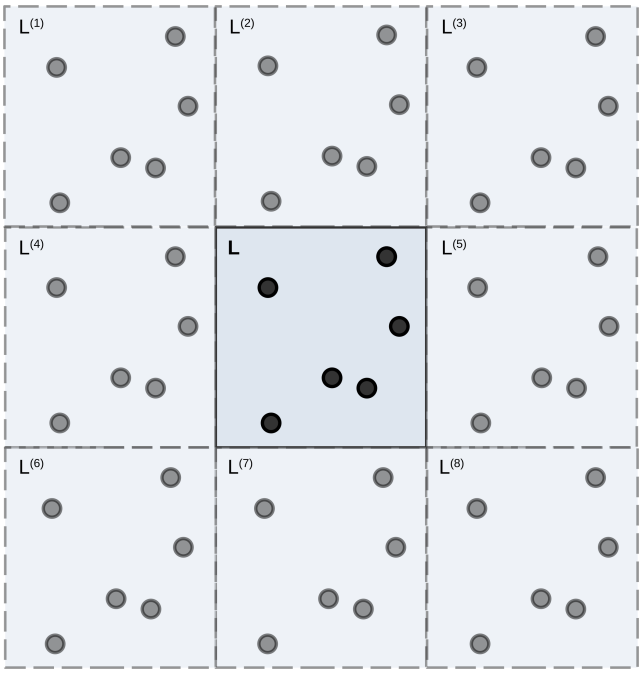


Figure 3.3: Figure illustrating the use of 2D PBCs, using which the 3D case can be easily imagined. The whole lattice L is repeated infinitely in all possible directions.

slowly but also conditionally, depending on the order of the summation [61, 66].

The above equations actually describe the case of point charge distribution, for which the charge distribution function may be defined as follows,

$$\rho_i(\mathbf{r}) = q_i \delta(\mathbf{r} - \mathbf{r}_i), \tag{3.10}$$

where δ is the Dirac delta distribution (also called the unit impulse). The charge distribution function can be split into short- and long-range terms

$$\begin{aligned}
\rho_i(\mathbf{r}) &= \rho_i^{sr}(\mathbf{r}) + \rho_i^{lr}(\mathbf{r}) \\
\rho_i^{lr}(\mathbf{r}) &= q_i G_\sigma(\mathbf{r} - \mathbf{r}_i) \\
\rho_i^{sr}(\mathbf{r}) &= q_i \delta(\mathbf{r} - \mathbf{r}_i) - q_i G_\sigma(\mathbf{r} - \mathbf{r}_i)
\end{aligned}$$

where G_σ is the Gaussian distribution function, given as

$$\begin{aligned}
G_\sigma(\mathbf{r}) &= \frac{1}{(2\pi\sigma^2)^{\frac{3}{2}}} \exp\left(-\frac{||\mathbf{r}||^2}{2\sigma^2}\right) \\
\lim_{\sigma \to 0} G_\sigma(\mathbf{r}) &= \delta(\mathbf{r}).
\end{aligned}$$

Similar to the charge distribution, using the solution of Poisson's equation

$$\nabla^2 \phi_i(\mathbf{r}) = -\frac{\rho_i(\mathbf{r})}{\varepsilon},$$

we can derive the split electric potential fields,

$$\phi_i(\mathbf{r}) = \phi_i^{sr}(\mathbf{r}) + \phi_i^{lr}(\mathbf{r}). \tag{3.11}$$

The above splitting results in a splitting of the energy terms as well, resulting in

$$E = E^{sr} + E^{lr}.$$

Now, the characteristic idea is to exploit the fact that the short- and long-range terms converge rapidly in the real and reciprocal spaces, respectively. In the classical Ewald method, this strategy yields the following equation:

$$E = E^{sr} + E^{lr} - E_{\text{self}},$$

where the short-range sum is

$$E^{sr} = \frac{1}{4\pi\varepsilon}\frac{1}{2}\sum_{n_1=0}^{\infty}\sum_{n_2=0}^{\infty}\sum_{i=1}^{N}\sum_{j=1}^{N}{}' \frac{q_i q_j}{||\mathbf{e}_c(n_1,j) - \mathbf{e}_c(n_2,i)||}\text{erfc}\left(\frac{||\mathbf{e}_c(n_1,j) - \mathbf{e}_c(n_2,i)||}{\sqrt{2}\sigma}\right),$$

where 'erfc' denotes the complementary error function. The long-range sum is,

$$E^{lr} = \frac{1}{2V\varepsilon}\sum_{\mathbf{k}\neq 0}\frac{e^{-\sigma^2||\mathbf{k}||^2/2}}{||\mathbf{k}||^2}||S(\mathbf{k})^2||,$$

in which the summation is over the reciprocal lattice, V denotes the volume of the periodic cell and the so-called structure factor is given by

$$S(\mathbf{k}) = \sum_{i=1}^{N} q_i e^{i\mathbf{k}\mathbf{r}_i}.$$

Finally, the self-interaction term is given by,

$$E^{\text{self}} = \frac{1}{4\pi\varepsilon}\frac{1}{\sqrt{2\pi}\sigma}\sum_{i=1}^{N} q_i^2.$$

It can be observed that E^{sr} is truncated by the 'erfc'-factor and E^{lr} by $e^{-\sigma^2||\mathbf{k}||^2/2}$ [67]. Well-optimized implementations of the classical Ewald method can achieve a complexity of $O(N^{\frac{3}{2}})$ [64].

The most commonly used methods are the *Particle-Mesh Ewald* method (PME), the Smooth Particle-Mesh Ewald method (SPME) and the Particle-Particle-Particle-Mesh method (P3M). These techniques mainly differ in how the particle properties are evaluated on the grid and then transferred back to the particles. Most implementations make use of the FFT to compute the Fourier transform, which allows them to further reduce the complexity to $O(Nlog_2N)$ [64].

3.2 *ab initio* Molecular Dynamics

Combining concepts of cMD and QM electronic structure calculations has resulted in a family of techniques know as *ab initio* Molecular Dynamics (aiMD) [68]. In contrast to cMD, instead of depending on the kind of extensive empirical modeling that force fields represent (as presented in section 3.1.2), aiMD relies only on QM first principles [4] to compute forces on the fly. This enables accurate simulations of bond breaking or formation, electronic polarization effects and quantum effects. Moreover, aiMD simulations can show new phenomena happening as necessary, which were not anticipated prior to simulation [68].

3.2.1 Born-Oppenheimer Approximation

DEFINITION : WAVE FUNCTION

A complex-valued function that encodes the state of an isolated quantum particle or system at any given time.

It is well-known that the determinism of classical mechanics gives way to the probabilistic descriptions of quantum mechanics. This is captured by the notion of the complex-valued *wave function** [69]. According to the Born interpretation, the wave function has the property that the modulus of its square is the probability density function of finding the particle at a particular location at that time [70] i.e., in one dimension,

$$\rho(x) = \int_a^b dx|\Psi(x,t)|^2 \quad \text{for } x \in [a,b],$$
$$1 = \int_{-\infty}^{\infty} dx|\Psi(x,t)|^2.$$

The non-relativistic[5] Hamiltonian considering the electronic and nuclear degrees of freedom $\{\mathbf{r}_i\}$ and $\{\mathbf{R}_I\}$, respectively, is an *operator*[6] acting on the wave function that is the solution of the time-dependent Schrödinger wave equation given by

$$i\hbar\frac{\partial}{\partial t}\Psi(\{\mathbf{r}_i\},\{\mathbf{R}_I\};t) = \hat{\mathcal{H}}\Psi(\{\mathbf{r}_i\},\{\mathbf{R}_I\};t) \tag{3.12}$$

[4] A first principle is a proposition that cannot be deduced from any other proposition.

[5] A non-relativistic concept assumes that all velocities are much lower than that of light and hence ignores effects such as time dilation or mass-energy equivalence.

[6] Generally speaking, the relationship between an operator and its operand function is analogous to that between a matrix and vector.

where $\hbar$ is the reduced Planck's constant [68]. The Hamiltonian can be written as,

$$\hat{\mathcal{H}} = \hat{T}_n + \hat{U}_{nn} + \hat{\mathcal{H}}_e \quad \text{with}$$
$$\hat{\mathcal{H}}_e = \hat{T}_e + \hat{U}_{ne} + \hat{U}_{ee},$$

where $\hat{T}$ represents the kinetic energy operator, $\hat{U}$ the potential energy operator and subscripts nn, ne and ee represent nucleus-nucleus, nucleus-electron and electron-electron interactions, respectively.

The difficulty of solving equation (3.12) can be lessened by applying the Born-Oppenheimer approximation. Since the mass of the nucleus is three orders of magnitude greater than that of an electron, it moves correspondingly much slower as well. This means that every change in the nuclear geometry will be followed by a rapid electronic structural adaption in response to it [69]. This lets us assume that during the *adiabatic* motion of the nuclei, there is no change in the electronic structure. Applying this approximation results in the factorization of the wave function into an electronic and a nuclear wave function as follows,

$$\Psi_{\text{total}}(\{\mathbf{r}_i\}, \{\mathbf{R}_I\}; t) = \psi_{\text{electron}}(\{\mathbf{r}_i\}, \{\mathbf{R}_I\}; t)\Phi_{\text{nucleus}}(\{\mathbf{R}_I\}; t).$$

Luckily, this approximation is applicable to a large number of problems. With it, we can solve the time-*independent* Schrödinger wave equation (TISE) as if the nuclei were stationary. This is given as

$$\hat{\mathcal{H}}_e\psi_k(\{\mathbf{r}_i\}; \{\mathbf{R}_I\}) = E_k(\{\mathbf{R}_I\})\psi_k(\{\mathbf{r}_i\}; \{\mathbf{R}_I\}), \tag{3.13}$$

where the electronic wave function ψ_k denotes the k-th quantum eigenstate and the corresponding eigenvalue E_k the energy associated with it.

At this point, it would be necessary to solve the TISE for the nuclear wave function using the obtained E_k. However, an additional approximation can be applied by neglecting potential nuclear quantum effects i.e., the nuclei can be treated classically. Hence, similarly to cMD (see subsection 3.1.1), the time evolution of the nuclei is performed by computing the forces on the nuclei as the negative gradient of the Born-Oppenheimer effective potential U_{BO} obtained by solving equation (3.13) for the k-th state:

$$M\ddot{\mathbf{R}} = -\nabla U_k^{BO}(\mathbf{R}) = -\nabla E_k(\mathbf{R}),$$

for $\mathbf{R} = (\mathbf{R}_I^\intercal)^\intercal$. Hence, the forces are ultimately a function of the nuclear coordinates.

3.2.2 Density Functional Theory

Density Functional Theory (DFT) is a powerful and versatile technique to solve the TISE (equation (3.13)) for many-particle systems. Instead of attempting to solve the challenging problem as it is (which is the approach of wave function-based methods), DFT reduces the complexity imposed by the exact treatment of the many-particle problem by avoiding the explicit calculation of the electron-electron interactions in the TISE. This is made possible by an approximation converting the many-particle problem *effectively* into a single-particle problem [71]. To achieve this, our focus must initially shift from the wave function to the *electron density* as the quantity central to solution [69].

Equation (3.13) for a many-particle system can be written as[78]:

$$\left[\underbrace{\sum_{j=1}^{N} -\frac{(\hbar\nabla_j)^2}{2m}}_{\text{I}} + \underbrace{\sum_{j=1}^{N} v_{\text{ext}}(\mathbf{r}_j)}_{\text{II}} + \underbrace{\sum_{\alpha=1}^{N}\sum_{\beta>\alpha}^{N} \frac{q^2}{||\mathbf{r}_\alpha - \mathbf{r}_\beta||}}_{\text{III}}\right] \Psi_k(\{\mathbf{r}_{1\to N}\}) = E_k \Psi_k(\{\mathbf{r}_{1\to N}\}), \tag{3.14}$$

where the first and third terms represent the kinetic operator and electrostatic repulsive interactions between the system's electrons, respectively, and are referred to as the universal part of the electronic Hamiltonian. The second term stands for interactions with an abstract, external (local) potential, which is defined with respect to the system's nuclei (i.e, similar to U_{ne} in our previous formulation of the electronic Hamiltonian) [69, 71]. DFT conceptually isolates this term, recognizing that it is the system-specific part of the electronic Hamiltonian.

The Hohenberg–Kohn (HK) theorems, which are instrumental in developing the DFT, state [72]:

1. (the first theorem can be expressed by equivalent statements)
   (a) the ground-state energy E_0 is *uniquely* determined by a *functional*[9] of the ground-state electron density n_0;
   (b) there is a one-to-one correspondence between the external potential, the non-degenerate ground state $|\Psi_0\rangle$ and the associated ground-state electron density;
   (c) there cannot be multiple external potentials for a given ground-state electron density;

[7] Strictly speaking, a complete description must include spin as well, which we neglect here for simplicity.

[8] In this subsection, N refers to the number of electrons and not the number of atoms.

[9] Intuitively, a functional is a function of a function i.e., it takes a function as an input and provides a scalar as an output. It corresponds to the linear form in vector spaces.

2. the required ground-state electron density minimizes the above-mentioned functional.

We will now attempt to illustrate that, using these theorems, we can convert the difficult task of solving equation (3.14) for a wave function of $\mathcal{O}(N)$ variables into a task of effectively solving the following equation for $\mathcal{O}(1)$ variables:

$$\left[-\frac{\hbar^2\nabla^2}{2m} + v_s(\mathbf{r})\right]\psi_k(\mathbf{r}) = \varepsilon_k\psi_k(\mathbf{r}). \tag{3.15}$$

Using Dirac's notation for a continuous basis, we can express the following example relation:

$$\langle a(\mathbf{r})|\hat{O}|b(\mathbf{r})\rangle = \int d^3\mathbf{r}' \int d^3\mathbf{r}\; a^*(\mathbf{r}')\;\hat{O}\;b(\mathbf{r}),$$

for arbitrary state functions $a(\mathbf{r})$ and $b(\mathbf{r})$, operator $\hat{O}$ (which takes on a matrix representation within the integral) and $a^*(\mathbf{r})$ denoting the complex conjugate of $a(\mathbf{r})$. In the same way, the electron density for the many-particle problem can be expanded from

$$n(\mathbf{r}) = \langle\Psi_k|\hat{n}|\Psi_k\rangle,$$

where Ψ_k is the k-th state wave function and $\hat{n}$ is the electron density operator, which gives the spatial distribution of electrons as

$$\hat{n}(\mathbf{r}) = \sum_{i=1}^{N}\delta(\mathbf{r}-\mathbf{r}_i).$$

Now, the functional conceptualized by the HK theorems can be defined as follows (for arbitrary quantum eigenstate and electron density):

$$\begin{aligned} E[n] &= \langle\Psi[n]|\hat{T}+\hat{U}+\hat{V}|\Psi[n]\rangle, \\ &= \langle\Psi[n]|\hat{T}+\hat{U}|\Psi[n]\rangle + \int d\mathbf{r}\, v_{\text{ext}}(\mathbf{r})n(\mathbf{r}) \end{aligned} \tag{3.16}$$

The universality of $\hat{T}$ and $\hat{U}$ lets us define the ground-state wave function Ψ_0 that minimizes the first term above and reproduces n_0. This double requirement sufficiently constrains Ψ_0 in terms of n_0 so that it is not necessary to specify $v_{\text{ext}}(\mathbf{r})$ explicitly [71].

The equivalence of the ground-state electron density and the ground-state wave function as suggested by the HK theorems theoretically makes the latter redundant [71]. Therefore, if we know $v_{\text{ext}}(\mathbf{r})$ and can adequately approximate $\hat{T}$ and $\hat{U}$, we could minimize

$$E_v[n] = T_{\text{approx}}[n] + U_{\text{approx}}[n] + V[n]$$

with respect to $n(\mathbf{r})$ to obtain the ground-state energy and electron density, thus avoiding having to solve the many-particle TISE altogether. However, in practice, not all observables can be computed so straightforwardly in this manner. The solution of this minimization problem is also a non-trivial numerical problem and does not represent the most efficient method.

One widely used strategy to indirectly solve the minimization problem is via the Kohn-Sham scheme, which facilitates the previously mentioned mapping of the interacting many-particle (primary) problem onto a hypothetical, non-interacting single-particle (auxiliary) problem as shown in equation (3.15). The fictitious system's Hamiltonian $\hat{H}_s = \hat{T} + \hat{V}_s$[10] yields the energy functional as

$$E_s[n] = \langle \psi[n] | \hat{T} + \hat{V}_s | \psi[n] \rangle = T_s[n] + \int d\mathbf{r} v_{\text{ext}}(\mathbf{r}) n(\mathbf{r}).$$

Keeping these quantities in mind, we can revisit the many-particle energy functional of equation (3.16) to decompose it:

$$\begin{aligned} T[n] &= T_s[n] + T_c[n] \\ &= -\frac{\hbar^2}{2m} \sum_{i=1}^{N} \int d^3\mathbf{r} \phi_i^*(\mathbf{r}) \nabla^2 \phi_i(\mathbf{r}) + T_c[n], \\ U[n] &= U_H[n] + U_{si}[n] \\ &= \frac{q^2}{2} \int d^3\mathbf{r}' \int d^3\mathbf{r} \frac{n(\mathbf{r}')n(\mathbf{r})}{||\mathbf{r}' - \mathbf{r}||} + U_{si}. \end{aligned}$$

Now, in the above decompositions, the kinetic energy term has been divided into a single-particle component and a "correlation" component, and the interaction energy has been approximated by the Hartree energy with a remainder term representing the self-interactions [72]. If we allow the latter components in each decomposition to wander into a separate term called the exchange-correlation energy functional $E_{xc}[n]$, we can formulate the many-particle energy functional so:

$$E[n] = T_s[n] + U_H[n] + V[n] + E_{xc}[n],$$

where the exchange-correlation functional represents a mathematical correction term. A large number of these functionals e.g., the Local Density Approximation (LDA) and the Generalized Gradient Approximation (GGA), have been developed to represent different classes of systems [8].

Through a derivation that is not presented here, $v_s(\mathbf{r})$ can be obtained, which realizes the link between the primary and the auxiliary problems when [71]

$$v_s(\mathbf{r}) = v_{\text{ext}}(\mathbf{r}) + v_H(\mathbf{r}) + v_{xc}(\mathbf{r}),$$

[10] The subscript 's' indicates quantities belonging to the single-particle problem.

where,

$$v_{\text{ext}}(\mathbf{r}) = \frac{\delta V}{\delta n(\mathbf{r})}; \quad v_H(\mathbf{r}) = \frac{\delta U_H}{\delta n(\mathbf{r})}; \quad v_{xc}(\mathbf{r}) = \frac{\delta E_{xc}}{\delta n(\mathbf{r})}.$$

Inserting it in equation (3.15), we get the Kohn-Sham equations. By solving these equations, the *Kohn-Sham orbitals* are obtained, which can be used to compute the ground-state density of the primary problem by

$$n_0(\mathbf{r}) = \sum_i |\psi_i(\mathbf{r})|^2. \tag{3.17}$$

The orbitals are typically expanded in a finite basis set known as the *plain wave basis set*, which is represented in the Fourier space for efficiency and simplicity. The Fourier transform is most commonly performed using the parallel FFT, which (although highly efficient) represents a performance bottleneck (both here and in the Ewald-type computations detailed in subsection 3.1.5) [8]. Hence, we attempted to improve on the performance of the parallel 3D FFT in the work reported by chapter 5.

RESEARCH FOCAL POINT

Investigate the possibility of improving the performance of the 3D DFT.

The Kohn-Sham equations can be solved in a self-consistent manner using the following numerical, iterative procedure:

1. start with an initial guess for the electron density $n^{(0)}(\mathbf{r})$ using simple models or previous calculations,
2. compute $v_s(\mathbf{r})$ using the current $n^{(\lambda)}(\mathbf{r})$,
3. solve the auxiliary problem (Kohn-Sham equations) to update the Kohn-Sham orbitals $\psi_i^{(\lambda)}(\mathbf{r})$,
4. update the electron density $n^{(\lambda+1)}(\mathbf{r})$ using equation (3.17),
5. check for convergence $||n^{(\lambda+1)}(\mathbf{r}) - n^{(\lambda)}(\mathbf{r})|| < \varepsilon_{\text{tol}}$
6. if converged, done; otherwise, go to step 2 and repeat.

The method typically converges in a few dozen iterations and it rarely fails to converge [71].

DFT offers accurate computational models even for crude approximations of the exchange-correlation energy. However, a strict limitation of DFT is that we cannot obtain the wave function using it. Another downside is its

requirement of having to choose from a variety of different functionals with slightly different performances for different, specific properties [69]. It finds widespread applications in diverse fields of physics, chemistry and even biophysics.

3.2.3 Multiscale MD —QM/MM

Multiscale methods bring together theoretical approaches of different timescales into a hybrid approach to extend the scope of study. This is especially useful when the simulated system consists of a large part that can be treated using a computationally less expensive method and a small part that needs a more accurate (and hence, expensive) treatment.

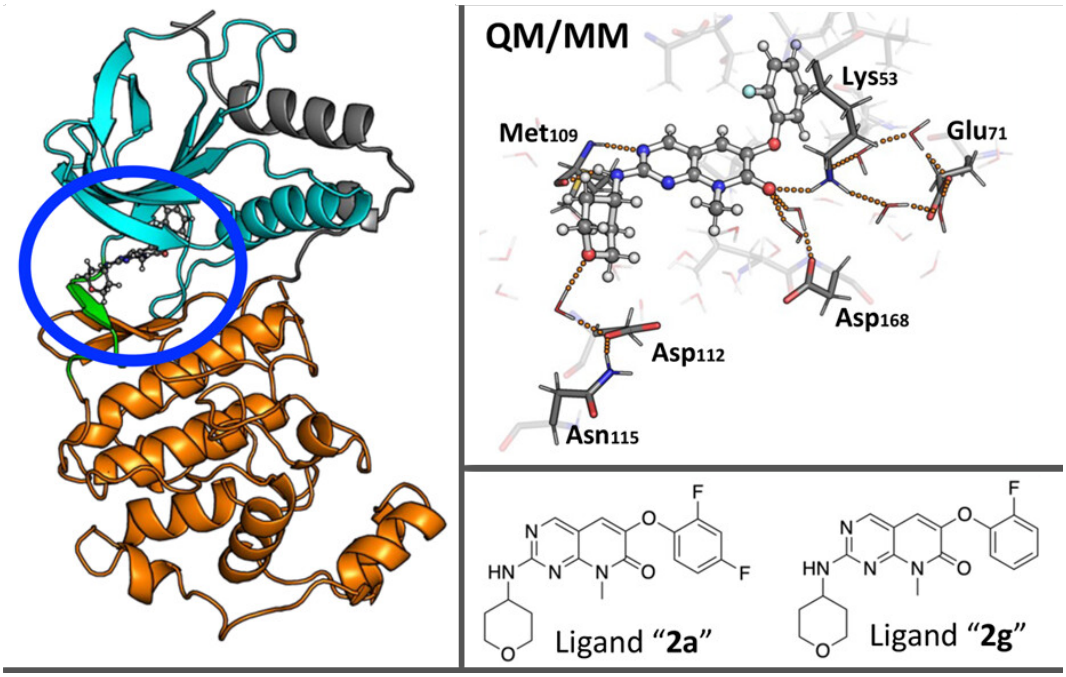


Figure 3.4: Figure illustrating the "representation of the p38α MAPK enzyme in complex with a ligand. The N-lobe (cyan) and the C-lobe (orange) of the enzyme are connected via a hinge region (green). The ligand binding pocket is located in between the lobes (blue ellipsis)". The image was originally published in the article [73], which is an Open Access publication licensed under CC-BY 4.0 (as per https://creativecommons.org/licenses/by/4.0/).

QM/MM methods recognize that it is often sufficient to treat only a small part of the simulated system using the QM approach (core region or QM region) while treating the rest of it using the approach of cMD (i.e., with force fields) [8, 74]. Implementations of QM/MM differ in how they handle boundary zone interactions (subtractive or additive energy terms) and

in how they couple QM and MM regions to model their mutual interactions (mechanical or electrostatic embedding) [8, 74]. QM/MM techniques are used in research areas such as metalloprotein structures, RNA-protein interactions and transition state analogues of enzymes [75].

Subtractive QM/MM is a relatively simple scheme in which the energy of the hybrid system is obtained as

$$E^{\text{total}} = E_q^{\text{QM}} + E_{q+c}^{\text{MM}} - E_c^{\text{MM}},$$

where superscripts 'QM' and 'MM' refer to the level of treatment provided and q and c refer to the QM and MM subsystems, respectively. Using this scheme, it is easy to setup more than two methods and their associated subsystems [76]. Another advantage is the lack of communication between the codes handling the subsystems. The disadvantages are that a special force field is required for the QM subsystem that is flexible enough to handle reactions and that the polarization of the electron density of the QM periphery by the MM environment cannot be described [77]. Example software for this scheme are ONIOM and COMQUM [8, 76, 77].

Additive QM/MM uses cross-terms of interaction between the QM and MM subsystems. The total energy is given by

$$E^{\text{total}} = E_q^{\text{QM}} + E_c^{\text{MM}} + E_{q+c}^{\text{QM/MM}}.$$

Here, only the interactions within the MM subsystem are given an MM treatment. In bonded interactions, if a chemical bond crosses the boundary between the subsystems, it is saturated using either a fictitious hydrogen atom or a specially built pseudopotential [8, 77]. Non-bonded interactions are mostly handled via mechanical and electrostatic embeddings[11].

When using the mechanical embedding, all interactions between the QM and MM subsystems are handled using force fields. For the electrostatic interactions, the QM region can be approximated as having a fixed set of charges, or the charges can be fitted periodically to reproduce the electrostatic potential using a large grid [78]. Parameters of the LJ potential are usually not changed during the simulation. This embedding has the disadvantage that the MM subsystem cannot polarize the QM subsystem [77].

In case of the electrostatic embedding, the computation of the electrostatic interactions between the subsystems is integrated into the computation of the QM wave function; interactions with the atomic charges of the MM subsystem appear within the external potential term of the TISE formulation. In this approach, the polarization of atoms in the QM subsystem by those

[11] The most realistic results can be obtained using the polarization embedding, for which polarizable force fields are typically necessary.

in the MM subsystem can be simulated. However, ensuring realistic simulation of the same may require adapting the parameterization of both force fields and the LJ potential of the MM subsystem. [77]. Electrostatic embedding is also compatible with fully periodic boundary conditions, which enables the systems to be smaller and improves energy conservation. [8].

3.3 Statistical Mechanics

3.3.1 Ensemble Averages

The output of MD simulations consists of a large number of *microstates*, each of which comprises the positions and velocities of every individual atom of the system. However, the motivation of the scientist performing MD simulations is to observe macroscopic quantities under the simulated conditions. The necessary bridge between the microscopic level of atomic motion and the macroscopic level of thermodynamics is provided by statistical mechanics [38].

One of the key insights that lets us build this bridge is the notion of degeneracy —that multiple, (in fact, a large number of) distinct microstates of the system can correspond to a given macroscopic quantity [1]. Therefore, to observe a macroscopic quantity, it is not necessary to provide a description of the microstate that is exact on the atomic level, rather it is sufficient to average across a vast number of plausible descriptions. This leads us to the concept of the *ensemble** .

DEFINITION : ENSEMBLE

A vast collection of hypothetical copies of a system subject to given macroscopic constraints (volume, particle count, energy etc.) that are consistent with a given set of macroscopic (i.e., thermodynamic) properties.

DEFINITION : PHASE SPACE

The space containing all possible vectors $\mathbf{x} \in \mathbb{R}^{6N}$, $\mathbf{x} = \begin{pmatrix} \mathbf{R} \\ \mathbf{P} \end{pmatrix}$, each capturing a microstate of a system.

Let us imagine a volume element $d\mathbf{x}$ moving through the *phase space** , sampling microstates corresponding to multiple trajectories. Then, at a given

time t, we can use the ensemble distribution function $f(\mathbf{x}, t)$[12] to ascertain the *relative likeliness*[13] of finding the system (under the constraints of the ensemble) at the phase space point x [38]. Now, a theoretical pillar of MD is provided by Liouville's theorem, which states that for systems propagating Hamilton's laws of motion (see equation (3.4), f is time independent i.e., $\frac{df}{dt} = 0$.

Liouville's theorem lets us link every macroscopic quantity A of a system to a microscopic phase space function $a(\mathbf{x})$ as follows:

$$A = \int d\mathbf{x} f(\mathbf{x}) a(\mathbf{x}).$$

The ensemble distribution function is given as

$$f(\mathbf{x}) = \frac{1}{\mathcal{Z}} \mathcal{F}(\mathcal{H}(\mathbf{x})),$$

where $\mathcal{Z}$ is called the partition function and normalizes $\mathcal{F}$,

$$\mathcal{Z} = \int d\mathbf{x} \mathcal{F}(\mathcal{H}(\mathbf{x}))$$

and $\mathcal{F}$ is a function that relates the form of the ensemble distribution function to the Hamiltonian. Considering that the link between A and $a(\mathbf{x})$ is realized by means of a normalized distribution and an integral, it becomes clear that A represents an average, which we refer to as the *ensemble average*.

The concept of the ensemble average reflects how statistical mechanics forwards one of the core requirements of statistics viz., adequate sampling; converged results[14] are obtainable only if the set of sampled microstates is large enough. Consequently, if we want to use a trajectory that is the output of an MD simulation to compute ensemble averages, it must be long enough to meet the requirement of adequate sampling. Moreover, the important assumption of *ergodicity*[15] needs to be made —that given enough time, a system will explore its phase space completely.

3.3.2 Microcanonical Ensemble

A system for which time evolution is performed based on Hamilton's equations of motion 3.4 conserves energy [38]

$$\mathcal{H}(\mathbf{x}, t) = E; \quad \frac{d\mathcal{H}(\mathbf{x})}{dt} = 0.$$

[12] This is a probability density function, i.e., $f(\mathbf{x}, t) \geq 0$ and $\int d\mathbf{x} f(\mathbf{x}, t) = 1$ over the whole phase space.

[13] Although microstates either correspond to ensemble members (or not), f provides a probabilistic (i.e., non-binary) view.

[14] The results of simulating a system are said be converged if the macroscopic quantities of interest become stationary i.e., do not fluctuate significantly over sufficiently long periods of simulated time.

[15] This assumption is formally referred to as the ergodic hypothesis.

Such a system belongs to a *microcanonical* or constant atom count (N), volume (V) and energy (E) ensemble, for which,

$$\mathcal{F}(\mathcal{H}(\mathbf{x})) = \mathcal{N}\delta(\mathcal{H}(\mathbf{x}) - E),$$
$$\mathcal{Z} = \int d\mathbf{x}\mathcal{N}\delta(\mathcal{H}(\mathbf{x}) - E)$$

and hence, the ensemble distribution function is

$$f(\mathbf{x}) = \frac{\mathcal{N}}{Z}\delta(\mathcal{H}(\mathbf{x}) - E),$$

where $\mathcal{N}$ is a normalizing constant. The Dirac delta distribution δ is useful in "sifting" values; it weights all $\lambda \neq 0$ with 0 and provides an infinite boost at $\lambda = 0$ such that $\delta(\lambda) \geq 0$ and $\int d\lambda\delta(\lambda) = 1$.

The microcanonical ensemble is useful when studying fundamental properties of systems for which the conservation of energy is a necessary requirement.

3.3.3 Canonical Ensemble

The microcanonical ensemble is naturally realized by ergodic Hamiltonian dynamics. However, it can only describe perfectly isolated systems, which are not easily found in reality. Observable systems typically maintain constant pressure rather than constant volume. Nevertheless, an ensemble that fixes the atom count (N), volume (V) and temperature (T) can still serve as a good approximation of real systems at large atom counts [38].

A system at a constant temperature is in interaction with surrounding systems and exchanges energy with them. Yet, simulating a system interacting with its surroundings might require the simulation of a number of surrounding systems[16]. Therefore, we simplify our model by considering the immersion of the system in a perfectly elastic thermal bath as shown in figure 3.5.

Now, our composite system satisfies the requirement of a microcanonical ensemble since it can be considered to be in complete isolation. Its partition function can be given by

$$\mathcal{Z} = \mathcal{N}\int d\mathbf{x}_2 d\mathbf{x}_1 \delta(\mathcal{H}_1(\mathbf{x}_1) + \mathcal{H}_2(\mathbf{x}_2) - E),$$

where $\mathcal{N}$ is a normalizing constant and quantities with indices 1 and 2 belong to the system and the thermal bath, respectively. Using the assumption

[16] Taken to the theoretical extreme, the perfectly accurate model must include the entire universe!

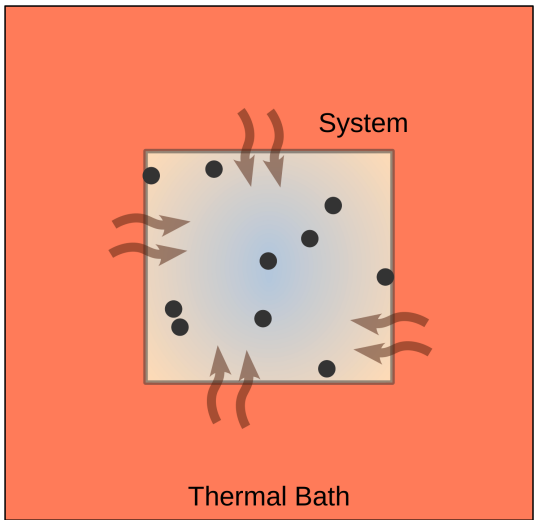


Figure 3.5: Figure illustrating the modeling of our system as immersed in a thermal bath.

that $\mathcal{H}_2(\mathbf{x}_2) \gg \mathcal{H}_1(\mathbf{x}_1)$, it can be shown that

$$\mathcal{Z}_1 = C_N \int d\mathbf{x}_1 e^{-\beta \mathcal{H}_1(\mathbf{x}_1)},$$

where C_N is a normalizing constant, $\beta = \frac{1}{k_b T}$ for Boltzmann constant k_b and temperature T. Correspondingly, the ensemble distribution function is

$$f(\mathbf{x}_1) = \frac{1}{\mathcal{Z}_1} e^{-\beta \mathcal{H}_1(\mathbf{x}_1)} \tag{3.18}$$

and it is referred to as the Boltzmann distribution.

When the thermal bath has a sufficiently high heat capacity and is at a fixed temperature, the energy of our system will fluctuate so as to maintain that temperature. This fluctuation ΔE can be shown to satisfy

$$\frac{\Delta E}{E} \sim \frac{1}{\sqrt{N}}$$

for total energy E, which is extensive in N. Therefore, at atom counts approaching infinity, i.e., $N \to \infty \implies \Delta E \to 0$, the canonical ensemble becomes a microcanonical ensemble [38].

3.3.4 Langevin Dynamics

Simulations of canonical ensembles do not actually use the above model, since it would result in the additional computational cost for the *degrees of freedom** of the thermal bath. In practice, there are more efficient ways

to model the required (virtual) thermal interactions. One such method is provided by *Langevin Dynamics* (LD).

DEFINITION : DEGREE OF FREEDOM

An independent parameter that defines a system's microstate and manifests energy —either by a distinct mode of motion (kinetic energy) or by virtue of location (potential energy). A system's degrees of freedom relate to the number of agents who can express them and its physical dimensionality. In cMD, a system has $6N$ degrees of freedom.

The generalized Langevin equation represents a phenomenological description of the dynamics of a system using both deterministic and random forces [38]:

$$M\ddot{\mathbf{Q}}(t) = -\nabla U(\mathbf{Q};t) - \int_0^t ds\dot{\mathbf{Q}}(s)K(t-s) + \mathbf{R}_n(t),$$

where $\mathbf{Q}$ are generalized position coordinates. The first term is the deterministic component, the second is a friction component (that depends on past states) and the final term is a random function. This reduces to the standard Langevin equation when the so-called memory kernel $K(t-s)$ is replaced by the Dirac delta distribution $2\zeta\delta(t-s)$ (which removes its dependence on past states) and the random function by a white noise function [79],

$$M\ddot{\mathbf{R}}(t) = -\nabla U(\mathbf{R};t) - \zeta\dot{\mathbf{R}}(t) + \sqrt{\frac{2\zeta}{\beta}}\mathbf{W}(t),$$

where ζ is the friction tensor[17] and $\mathbf{W}$ is a stationary Gaussian process satisfying $\langle\mathbf{W}(t)\rangle = 0$ and $\langle\mathbf{W}(t).\mathbf{W}(t')\rangle = \delta(t-t')$.

In case we are interested in simulating a highly viscous system in which friction forces dominate over inertial forces, we can derive the equation for overdamped Langevin or *Brownian Dynamics* (BD) by neglecting the inertial term:

$$0 = -\nabla U(\mathbf{R};t) - \zeta\dot{\mathbf{R}}(t) + \sqrt{\frac{2\zeta}{\beta}}\mathbf{W}(t),$$

$$\dot{\mathbf{R}}(t) = -D\beta\nabla U(\mathbf{R};t) + \sqrt{2D}\mathbf{W}(t),$$

where $D = \frac{1}{\zeta\beta}$ is referred to as the diffusion coefficient.

[17] $\zeta = \gamma M$, where γ is called the damping coefficient with units time^{-1}

Depending on the applicability of their approximations, LD/BD can be used as simple and efficient thermostatting methods in MD simulations to generate an approximation of the canonical ensemble [38].

3.3.5 Isothermal-Isobaric Ensemble

The constant atom count (N), pressure (P) and temperature (T) ensemble most closely reflects real-world experimental conditions. To keep the pressure constant, it must be permissible that the volume of the system changes. As we demonstrated the modeling of our thermal-bath coupling for the NVT system, so can we model a pressure coupling for an isobaric system, which involves a piston compressing or expanding the system in response to fluctuations of pressure so as to keep it constant.

In MD simulations, approximations of NPT ensembles can be generated using barostats. However, since the ensembles that are most relevant to our work are canonical ensembles, we will not discuss the details of NPT ensembles and barostats here.

3.4 Enhanced Sampling

As of today, state-of-the-art cMD software allow simulating microsecond timescales for systems containing many hundred thousand atoms [8]. Still, there is a strong motivation to be able to simulate longer timescales because phenomena such as drug-receptor binding, phase transitions, protein folding, slow diffusion in solids and chemical reactions occur e.g., on the millisecond timescale [12, 80]. This is due to the fact that such simulation systems present several metastable states that are separated by high free-energy barriers. In such cases, the transition between such metastable states becomes a "rare event" that requires very long simulations in order to be spotted [81]. So much so, that such simulations are often not even feasible using modern HPC infrastructure. Moreover, observing a single transition is often not enough; we need to study numerous transitions to obtain meaningful results. Out of this necessity, myriad advanced techniques have been invented, which are collectively referred to as *enhanced-sampling methods*.

Enhanced-sampling methods are varied and numerous. Some examples are umbrella sampling, replica exchange, parallel tempering, potential smoothing, temperature-accelerated MD, conformational flooding, adaptive biasing force, variationally enhanced sampling, hyperdynamics and metadynamics [82]. To guide the simulation, many methods make use of *collective variables** (CVs), whose fluctuations are critical for the rare event to take place [81]. In metadynamics, sampling is accelerated by adding a time-dependent external potential (i.e., the bias, which is a function of the CVs) to

the Hamiltonian. However, as a consequence, this addition of bias changes the natural dynamics of the system. On the one hand, we can rely on several re-weighting approaches that have been designed to retrieve the correct statistics of the unbiased Hamiltonian. On the other hand, retrieving kinetic information from biased enhanced-sampling trajectories is less straightforward; it can typically only be achieved by using an especially engineered bias [12]. Although more general methods have also been proposed to recover dynamic properties from biased trajectories, they are prone to numerical instabilities [83].

DEFINITION : COLLECTIVE VARIABLES

A set of low-dimensional degrees of freedom used to simplify the description of a transition process while (selectively) reflecting essential quantities, the variations of which are indicative of progress towards/regression from the target metastable state.

Based on theory involving the Onsager-Machlup action, there are other methods that focus on sampling the trajectory space (as opposed to the configuration space) to find reactive paths between the metastable states. One such path-based method is the Monte Carlo-based approach known as Transition Path Sampling (TPS), which connects two *a priori* known metastable states. Although highly successful and widely used, this method has the limitation that the initial and final states, as well as at least one reactive path between them, must be known prior to simulation. Moreover, if multiple reactive paths exist, an algorithmic tendency to get drawn into a nearby saddle, thereby "getting trapped" by a local reactive path, is not uncommon, which requires special, corrective measures [84].

RESEARCH FOCAL POINT

Extend the reach of enhanced-sampling techniques by designing and implementing a robust and highly scalable MD algorithm.

Combining CV-based enhanced-sampling with TPS, Mandelli *et al.* proposed a method called *Metadynamics of Paths* (MoP) that can sample multiple reactive paths in a single run without requiring the final state to be chosen. Using this method, in spite of applying bias, dynamic properties of the system can be retrieved using established, exact re-weighting techniques. Recently, the method has also been used to design optimal data-driven CVs for free energy calculations [85]. However, so far, MoP has only been applied to small "toy" systems using a proof-of-concept implementa-

tion in LAMMPS. To pave the way to its application to real-world biological problems, we have implemented the required software components in the GROMACS suite of codes [86] and the PLUMED library [87], the algorithmic design, implementation and performance testing of which are detailed in chapter 4.

I'm told that, to see rare things, one must climb mountains.

Chapter 4

Metadynamics of Paths

In section 3.4 of chapter 3, we mentioned an advanced scheme in which path-based MD can be combined with metadynamics to *effectively* extend the simulation timescale. As the name suggests, this scheme involves sampling entire trajectories instead of configurations as in cMD. In this chapter, we will discuss the theory behind MoP, detail the implementation of a MoP algorithm in a high-performing MD code and present the performance results of this implementation.

4.1 Theory

4.1.1 Path Molecular Dynamics for Stochastic Trajectories

We begin by looking into how a path-based MD technique can generate a set of trajectories instead of a set of configurations. Let us consider a molecular system with Q atoms interacting via a time-dependent potential $U(\mathbf{R};t)$ where $\mathbf{R} \in \mathbb{R}^{3Q}$ is the atomic coordinates vector. We also assume that this system is coupled to a thermal bath at temperature T. Further, BD is applicable, described by

$$\begin{aligned} M\gamma\dot{\mathbf{R}}(t) &= \mathbf{F}(\mathbf{R};t) + \boldsymbol{\xi}(t), \\ \mathbf{F}(\mathbf{R};t) &= -\nabla U(\mathbf{R};t) \end{aligned} \tag{4.1}$$

where $\mathbf{F}(\mathbf{R};t)$ is the time-dependent force associated with the potential, M is the diagonal mass tensor, γ is the damping coefficient and $\boldsymbol{\xi}(t)$ is a white noise function. Under this assumption, it can be shown that the probability of observing a specific time-discretized trajectory $\mathbf{R}^0 \to \mathbf{R}^1 \to ... \to \mathbf{R}^{N-1}$ starting in a metastable state $\mathbf{R}^0$ having Boltzmann probability $e^{-\beta U(\mathbf{R}_0)}$ and ending in an arbitrary state $\mathbf{R}^{N-1}$, respectively, after N time steps of size Δt,

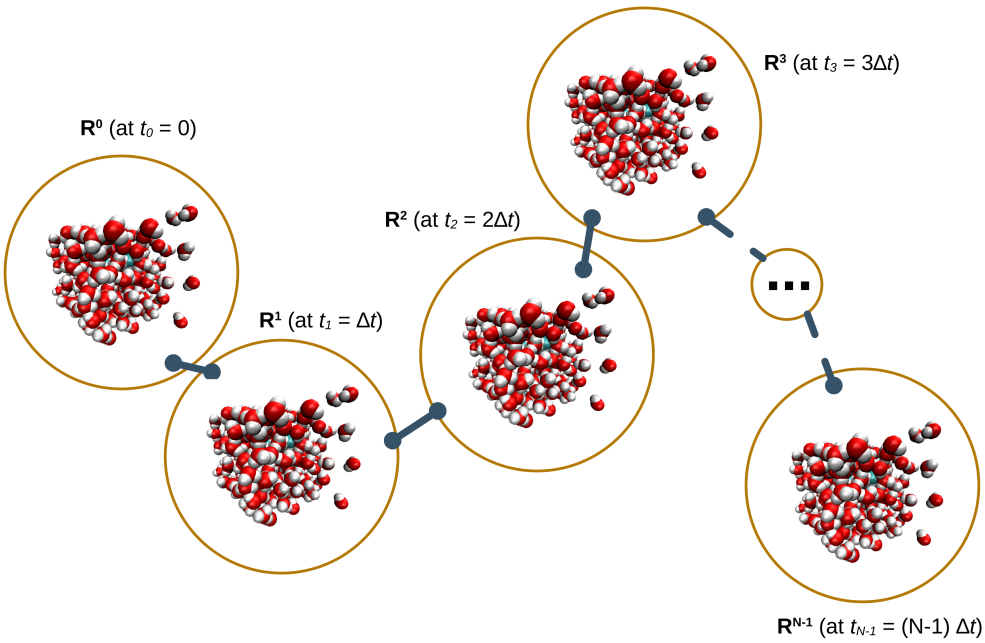


Figure 4.1: An illustration of pMD's fictitious polymer with N beads.

is

$$P(\{\mathbf{R}^0, \mathbf{R}^1, ..., \mathbf{R}^{N-1}\}) \propto \exp\left(-\beta V_{\text{poly}}(\{\mathbf{R}^0, \mathbf{R}^1, ..., \mathbf{R}^{N-1}\})\right), \tag{4.2}$$

which has the form of a Boltzmann distribution in the enlarged phase space spanned by all possible N-configuration trajectories of the system, with $\beta = \frac{1}{k_B T}$ and k_B the Boltzmann constant [80]. Here,

$$V_{\text{poly}} = U(\mathbf{R}^0) + \frac{1}{2} K \sum_{n=0}^{N-2} ||\mathbf{R}^{n+1} - \mathbf{R}^n - \mathbf{L}^n||^2 \tag{4.3}$$

defines the *polymer potential*. Using it, equation (4.2) can be interpreted as the potential of a harmonic *fictitious polymer* whose *beads* correspond to the set of configurations $\{\mathbf{R}^0, \mathbf{R}^1, ..., \mathbf{R}^{N-1}\}$ sequentially visited by the system along the time-discretized trajectory (see figure 4.1). Adjacent beads interact via "springs" of constant $K = \frac{\gamma}{2\Delta t} M$ and equilibrium length $\mathbf{L}^n = \frac{\Delta t}{\gamma} M^{-1} \mathbf{F}^n$, where $\mathbf{F}^n = -\nabla U(\mathbf{R}^n)$ is the time-independent force acting on the (arbitrary) n-th bead[1].

Now, equation (4.2) effectively converts the time-dependent problem of finding a trajectory $\mathbf{R}^0 \to \mathbf{R}^1 \to ... \to \mathbf{R}^{N-1}$ that is a solution of equation (4.1) into a time-independent problem of sampling a polymer configuration $\{\mathbf{R}^0, \mathbf{R}^2, ..., \mathbf{R}^{N-1}\}$ distributed according to equation (4.2). By run-

[1] Henceforth, any unexplained mention of n should be taken to mean an arbitrary bead index $0 \leq n < N$.

ning a finite-temperature MD simulation of the fictitious polymer, we effectively generate a complete trajectory of the system in *every* time step. This is done by evolving the dynamics of the polymer according to Newton's equations of motion

$$\mathbf{F}_{\text{poly}}(\mathbf{R}^n;\tau) = \mathcal{M}\ddot{\mathbf{R}}^n = -\nabla_n V_{\text{poly}}(\mathbf{R}^n;\tau) \tag{4.4}$$

in combination with a suitable thermostat. In the above equation, τ refers to the time associated with the time evolution of the fictitious polymer. It is important to note that $\mathcal{M} \in \mathbb{R}^{3Q}$ is a fictitious mass tensor, which can be considered a tuning parameter of the polymer simulation. Further, $\mathbf{F}_{\text{poly}}$ is called the *polymer force* and is given by three cases:

$$\mathbf{F}^0_{\text{poly}} = \mathbf{F}^0 + K\boldsymbol{\eta}^0 + \frac{1}{2}H_U(\mathbf{R}^0)\boldsymbol{\eta}^0, \tag{4.5}$$

$$\mathbf{F}^n_{\text{poly}} = K(\boldsymbol{\eta}^n - \boldsymbol{\eta}^{n-1}) + \frac{1}{2}H_U(\mathbf{R}^n)\boldsymbol{\eta}^n, \tag{4.6}$$

$$\mathbf{F}^{N-1}_{\text{poly}} = -K\boldsymbol{\eta}^{N-2}, \tag{4.7}$$

for the first bead, any middle bead (with $0 < n < N-1$) and the last bead, respectively. In equations (4.5) and (4.6),

$$\boldsymbol{\eta}^n = \mathbf{R}^{n+1} - \mathbf{R}^n - \mathbf{L}^n \tag{4.8}$$

can be referred to as the inter-bead *displacement vector* and H_U refers to the Hessian matrix of the potential U.

Henceforth, we will refer to the concept represented by equation (4.4) as *path MD* (pMD) and to any algorithm that details a structured approach towards performing MD simulations of the associated fictitious polymer as a pMD algorithm. Further, we will refer to MD in configuration space as *standard MD*, to distinguish it from pMD.

In equations 4.5 and 4.6, to avoid the requirement of an analytical formulation of the Hessian matrix for all possible PEFs, Mandelli *et al.* suggested an algorithm that employs the *Finite Difference Method* (FDM) to compute the derivative [80], making it applicable to generic PEFs and thereby automatically extending its scope of use. This is done in the same fashion as suggested by Putrino *et al.* [88] i.e., we have the central difference formula,

$$H_U(\mathbf{R}^n)\boldsymbol{\eta}^n = \frac{\mathbf{F}(\mathbf{R}^n + \varepsilon\boldsymbol{\eta}^n) - \mathbf{F}(\mathbf{R}^n - \varepsilon\boldsymbol{\eta}^n)}{2\varepsilon}, \tag{4.9}$$

where ε represents the discretization step.

4.1.2 Metadynamics of Paths

When studying a rare transition, we are fundamentally interested in finding trajectories that connect two metastable states. The ensemble of all these trajectories is called the *transition path ensemble*, which can be analyzed to obtain microscopic information about the molecular mechanism of the reaction, and to compute kinetic rates [80, 89]. Trajectories belonging to the transition path ensemble are characterized by vanishingly small Boltzmann weights of the corresponding polymer configuration (given by equation (4.2)). Therefore, sampling such polymer configurations during unbiased pMD simulations is a rare event. To overcome this problem, one can use any enhanced-sampling technique. Here, we focus on metadynamics [90–93], the basic concept of which will now be briefly described.

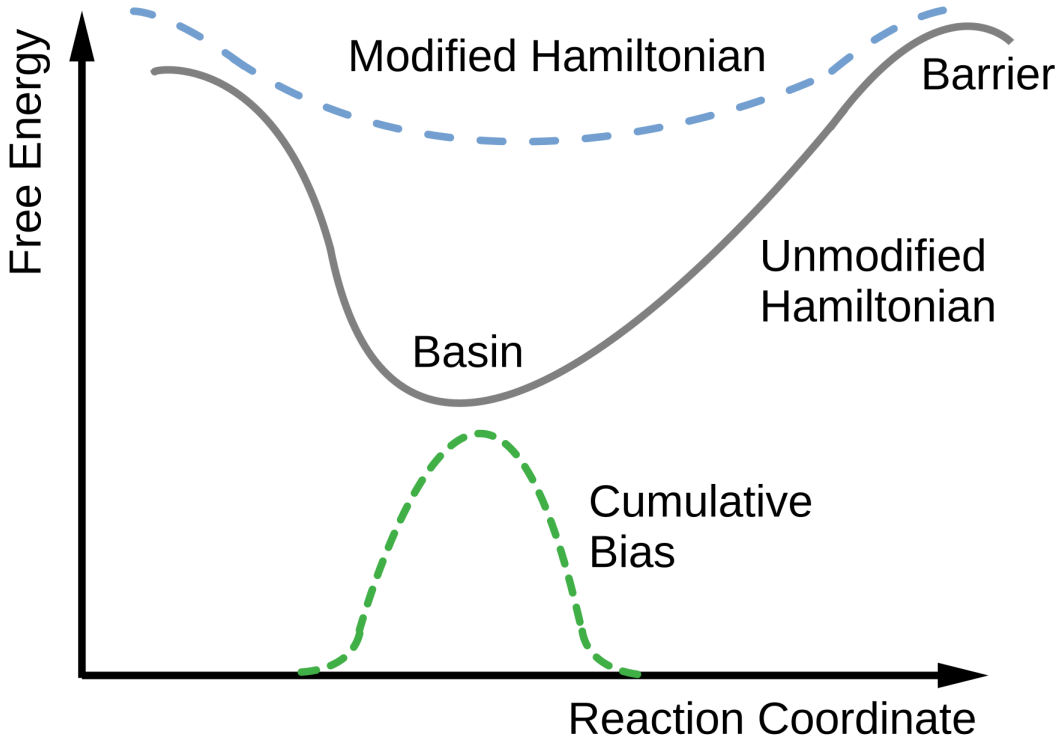


Figure 4.2: A figure demonstrating how modifying the Hamiltonian by adding bias can help accelerate phase space exploration by filling out a free-energy basin, thereby helping the system to escape it.

Let us consider two metastable states separated by a high free-energy barrier ($> k_BT$). If the system starts in either metastable state and we are interested in studying its transition to the other metastable state over the barrier, simply running an unbiased simulation could very likely result in the system oscillating around the local equilibrium state, thus getting stuck [81].

Applying the technique of metadynamics, an external (bias) potential is constructed dynamically during the simulation as a sum of Gaussian "hills", modifying the Hamiltonian as follows

$$H = T + V + \sum V_{\text{bias}}(s_1, s_2, ..., s_d; t),$$

where H is the Hamiltonian, T the kinetic energy, V the potential energy and V_{bias} the accumulated bias. This results in the filling up of the free-energy basin as shown in figure 4.2, which raises the system and helps it to transition to the target metastable state. In addition to this "lifting" effect, metadynamics can amplify CV fluctuations, which is crucial for making transitions more probable. Importantly, exact re-weighting techniques [94–100] can be used to recover the natural (unbiased) Boltzmann statistics to compute the correct ensemble averages excluding the effect of the added bias potential.

As presented in the above equation, the bias is constructed as a function of one or more carefully defined CVs $s_1, s_2, ..., s_d$, which themselves are (usually non-linear) functions of the atomic coordinates $s = s(\mathbf{R})$. Although the technique of metadynamics is popular for being somewhat insensitive to the choice of CVs, properly choosing CVs results in speedy convergence [81]. In particular, a necessary condition is that the CVs should be able to distinguish between all the relevant metastable states of interest. Ideally, they should also provide a description of the slow degrees of freedom underlying the studied physical process [81].

To apply metadynamics in the context of pMD, we need to define CVs capable of enhancing the sampling of polymer configurations representing trajectories belonging to the transition path ensemble. Previous research [80, 101] has shown that a good CV for this task is provided by a generalized *polymer end-to-end distance*, expressed as the difference between values of an order parameter $s(\mathbf{R})$ evaluated for the last and first beads as

$$S_{e2e} = s(\mathbf{R}^{N-1}) - s(\mathbf{R}^0). \tag{4.10}$$

It is important that $s(\mathbf{R})$ be able to distinguish between the starting state (A) and the final state (B) i.e., $s_A \neq s_B$. If this condition is met, trajectories in which the system visits only one of the two targeted metastable states (i.e. trajectories that are not reactive paths) will correspond to values of $S_{e2e} \approx 0$. On the other hand, trajectories that are reactive paths that sample transition events will correspond to values of $|S_{e2e}| = |s_B - s_A|$. By enhancing the fluctuations of S_{e2e}, the metadynamics bias increases the chances of sampling the reactive paths of interest.

4.2 Implementation

A proof-of-concept implementation of our pMD algorithm has been made in LAMMPS [102] and it has been used to test "toy" systems [80]. However, to apply MoP in the realm of complex biological systems, a new implementation prioritizing optimization and scalability was necessary. The new implementation would have to be capable of comprehensive, multi-level parallelism using *heterogeneous computing systems** and it would have to be efficient on every level of parallelism. Based on these requirements, we chose GROMACS as the MD code that would serve as the base for it.

DEFINITION : HETEROGENEOUS COMPUTING SYSTEM

A computing system that makes use of a combination of different types of PEs to boost performance while maintaining energy efficiency. The most commonly implemented combination is CPU-GPU, but CPU-MIC, APUs (CPU and GPU on a single chip) and CPU-FPGA combinations are also widespread.

This section is organized as follows. We begin by justifying our choice of MD code. Next, we discuss the parallelization scheme developed for our pMD implementation. We then explain the key details of the polymer force and energy computations and present error-correction strategies. Following this, we discuss algorithms used to improve the hardware resource utilization of our pMD implementation and highlight how GPU-based parallelism can be used with it. Finally, we show how metadynamics can be coupled with our pMD implementation.

4.2.1 Choice of MD Code: Why GROMACS?

GROMACS is a highly popular, open-source software package for MD simulations known for its performance and efficiency [103–105], which is the result of extensive algorithmic optimizations [106]. It uses an ambitious bottom-up approach to parallelism with the goal of leveraging it from the lowest to the highest level [104, 107]. This manifests itself in hand-crafted compute kernels that make use of explicit vector instructions, multithreading via OpenMP (or, alternatively, the thread-MPI library), NUMA-aware optimizations and, on the highest level, the MPI library for large-scale, process-based parallelism[2]. Beyond this, it is capable of dynamic run-time performance tuning with the help of its PME-tuning tool and dynamic load-balancing feature [106, 107].

[2]Vectorization increases hardware utilization within a core, multithreading within a node and multiprocess parallelism across nodes.

GROMACS is also well-known for its support for GPU-based computing via CUDA, OpenCL and SYCL [107]. Since GROMACS 2020, it allows the use of the "GPU-resident mode", in which all the key computational sections are run on the GPU, which eliminates the overhead of repetitive CPU-GPU data transfers [103]. Additionally, it recognizes GPU-aware MPI implementations, taking care to perform inter-GPU communication wherever necessary [107]. The above, key details have made GROMACS one of the most highly-performing MD codes available today [103–106].

GROMACS is also known for its versatility, offering a wide range of force fields, solvent models and free-energy algorithms [104]. It brings with it multiple integrators (e.g., for BD/LD), time-stepping algorithms and temperature and pressure couplings. Importantly, the community already provides a patch for GROMACS to perform metadynamics simulations via the PLUMED library [87, 104, 108], which can allow us to conveniently employ the technique of MoP.

For the above reasons of performance and versatility, as well as for its strong community support, we chose GROMACS (v2024) for the implementation of our pMD algorithm. The commit merging the multi-simulation feature was copied into a separate Gitlab repository (https://gitlab.com/anxiousprogrammer/gmx_pmd.git), which was dedicated to the implementation.

4.2.2 Parallelization Scheme

As explained in subsection 4.1.1, the core concept of pMD involves propagating the dynamics of a fictitious polymer composed of several beads. This, coupled with the fact that the concept exposes a new dimension of parallelism, lets intuition suggest that an approach to parallelism with one or more standalone parallel instances (viz., processes) doing the work of simulating a single bead will be both easy to implement and highly efficient.

Multi-Simulation

Since v2024, GROMACS provides the MPI-based *multi-simulation* feature, which allows the instantiation of several *replicas* of the simulated system[3], each of which can be simulated by independent MPI tasks. Although these simulations may run completely independently of one another, the feature is nonetheless outfitted with a dedicated MPI communicator that allows message passing between the replicas, which is required for techniques such as the replica exchange [39]. Since this existent feature suits our requirement

[3] In the context of this thesis, the word "system" can mean computing systems as well as simulated systems. Hence, we will take care to specify it.

very closely, it provides the basis for the implementation of our pMD algorithm in GROMACS.

Domain Decomposition

Internally, GROMACS also makes use of MPI within its system of *domain decomposition* (DD), which refers to the technique of splitting up the computational work of a simulation such that multiple PEs can share the work. In GROMACS, DD can be activated in two modes —for the computation of the short-ranged, non-bonded interactions referred to as *particle-particle* (PP) interactions and for the computation of the long-ranged, non-bonded interactions referred to as PME electrostatics. Since the implementation of our pMD algorithm is not affected by PME-based DD, we will not discuss it here.

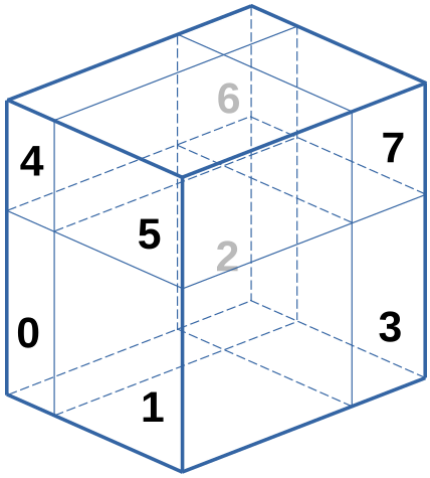


Figure 4.3: An illustration of GROMACS' spatial PP-based DD with $2 \times 2 \times 2$ cells.

For the PP interactions, GROMACS uses the *eighth shell* technique, which is a spatial DD; the simulation box is divided into multiple spatial domains such that each domain (along with the atoms geometrically located in it) can be handled by an MPI task running on a single PE. An example of this type of DD is presented in figure 4.3. Not only does each MPI task do the work of the force computation (excluding PME electrostatics, if separate PME tasks are active) but also of the key steps of the simulation such as the updating of coordinates and the parallel constraints algorithm, always operating on its *home atoms** only.

DEFINITION : HOME ATOMS

A subset of the system's atoms that is allocated to a particular domain by DD.

It has been documented that DD provides superior performance to OpenMP-based multithreading on multicore computers. To confirm this, we performed strong-scaling tests of GROMACS simulations by means of the example system lysozyme and benzene in explicit water (with $\sim$ 100k atoms) on the JUWELS Booster. The results proved not only that the performance improves when moving from OpenMP-based multithreading towards DD, but also that the simulation shows good strong-scaling performance via DD within a single node. Moreover, significant performance boosts can be attained applying DD beyond the single node, saturating at about 20 nodes (albeit at a low strong-scaling efficiency) as shown in figure 4.4. Therefore, to utilize hardware resources well, the conceptualized implementation of our pMD algorithm would have to make use of DD.

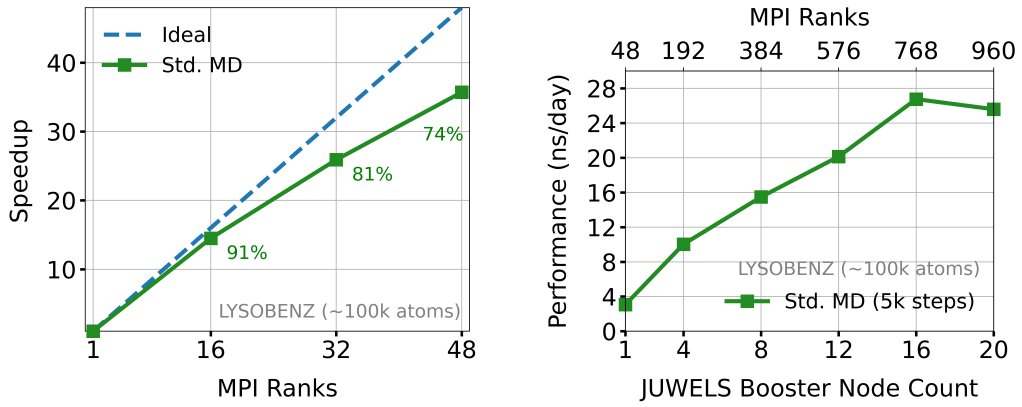


Figure 4.4: Subfigures show the strong-scaling performance of standard MD using DD on the intra-node level (left) and on the inter-node level (right) of the JUWELS Booster. The label 'LYSOBENZ' refers to the system lysozyme and benzene in explicit water (with $\sim$ $100k$ atoms). Here, only the short-range non-bonded interactions are computed using separate MPI tasks.

pMD's Double-Layered Scheme

The combination of the multi-simulation feature and DD results in a double-layered, MPI-based parallelization scheme in which each layer has its own MPI communicator. In this scheme, there is a unique MPI task for each bead

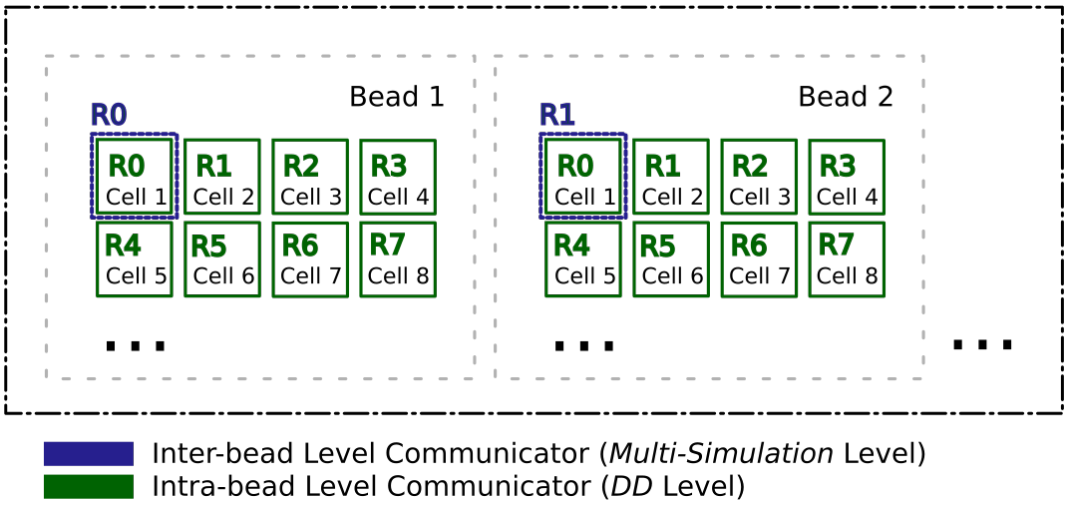


Figure 4.5: An illustration of the double-layered MPI-based parallelization scheme that serves as the base for our pMD implementation in GROMACS.

that has access to the multi-simulation communicator (in which it has a *main rank**) and a unique MPI task for each of the bead's domains that has access to the DD communicator (in which it has a *domain rank**). An MPI task corresponding to a main rank has access to both communicators and thus serves as the links between the two levels, as illustrated by figure 4.5. Using this scheme, it can be ensured that inter- and intra-bead message passing events take place within their own mutually insulated spaces.

Definition : Bead's Main Rank

The MPI rank that is responsible for a unified representation of its bead's information and for coordinating its domain ranks.

Definition : Bead's Domain Rank

The MPI rank that is employed primarily to do work within the context of the particular domain with which it is associated by DD.

Our pMD implementation has been conceived to seamlessly integrate this scheme so that we can make full and efficient use of the already existing structures in GROMACS. Given the multinode scaling performance enabled by DD and taking into account that the typical application uses 500-5,000 beads, the envisioned pMD implementation in GROMACS should theoretically be capable of exploiting *more parallelism* than can be provided by even very large clusters.

4.2.3 Polymer Force and Energy Computations

To be able to use as many features as possible, we planned to place the implementation of our pMD algorithm within the legacy code base of GROMACS. Here, the primary point of change can be found at that location in code where the internal function `do_force` is called to compute the forces of the simulated system for each time step. When running a pMD simulation, since we are essentially performing the time evolution of the fictitious polymer, this function call is replaced by a call to our new function `computePolymerForces` —which computes the polymer forces by including the spring-like inter-bead contributions described by equations (4.5), (4.6) and (4.7). The procedure for this new function —and hence our pMD algorithm itself —is depicted, detailing its computations and communications, by the flowchart shown in figure 4.6.

Communication

The new function presupposes inter-bead communication at various stages, since position and force vectors belonging to neighbouring beads are required by the numerical calculations at these stages. For example, the computation represented by equation (4.6) for a given bead requires the position vectors from its neighbours on the left and the right. The inter-bead communication pattern in all communication events is the well-known open ring messaging pattern illustrated by figure 4.7. There are three communication events in `computePolymerForces`: position vectors are sent to both left and right neighbours, and force vectors are sent to the right neighbour.

Our implementation ensures that the communication is performed asynchronously using non-blocking MPI functions, taking care to ensure that calls to `MPI_Irecv` are made prior to calling `MPI_Isend`. Moreover, to enable parallelism of communication and computation, calls to the most computationally intensive functions have been placed in between calls to `MPI_Irecv/MPI_Isend` and the calls to their corresponding waiting functions. The time window for MPI communications is extended as much as possible by placing the calls to `MPI_Irecv/MPI_Isend` as early as possible and those to their waiting functions as late as possible in code.

Computation

Within `computePolymerForces`, the vast majority of the computational effort is concentrated in three separate calls to the function `do_force`. The first call computes the non-polymer forces of the simulated system and the energy and virial tensor as and when necessary. The second and third calls are required by the FDM computation from equation (4.9) to numerically

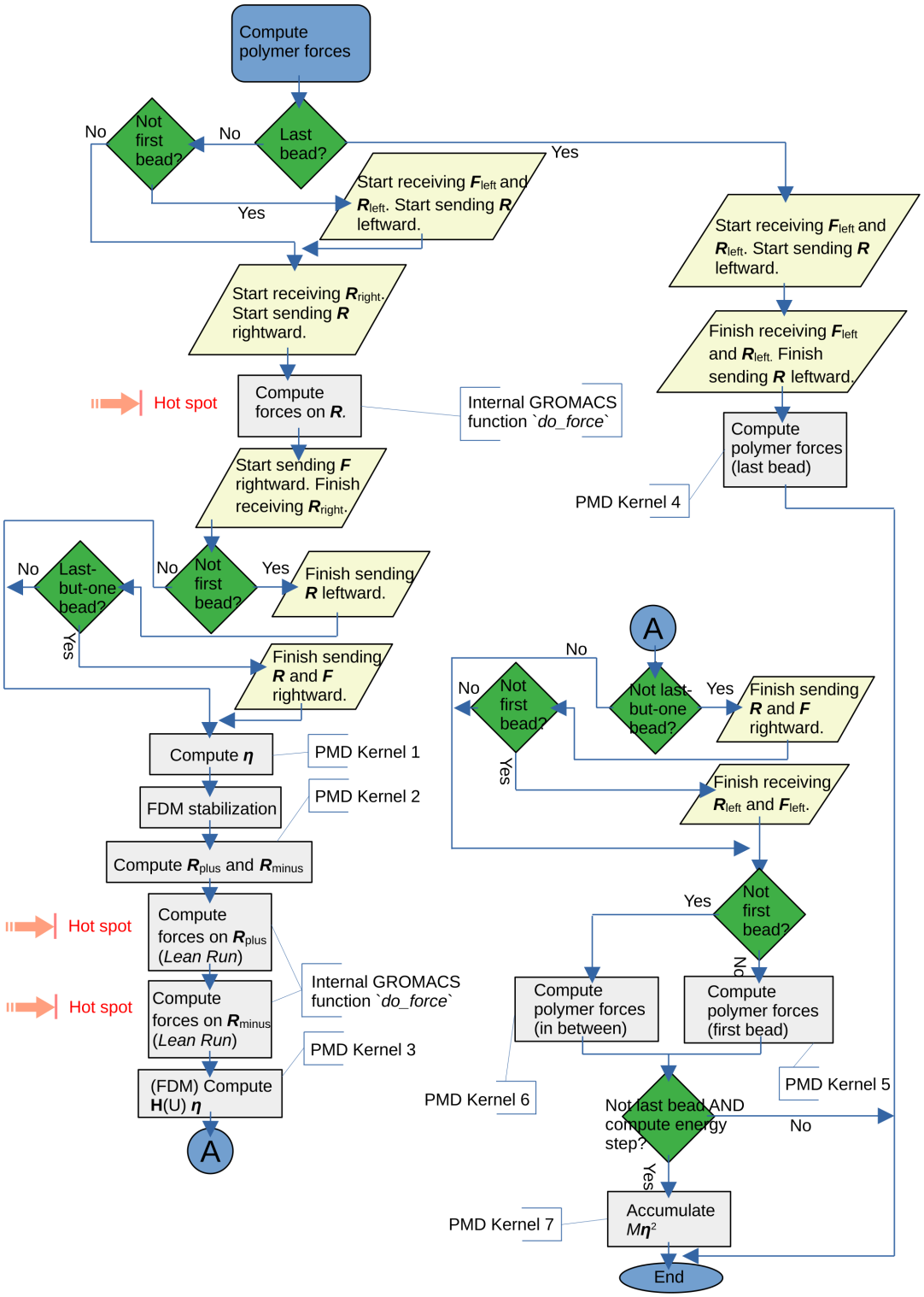


Figure 4.6: Flowchart for `computePolymerForces` highlighting the details of communication and computation. The second and third calls to `do_force` are referred to as "lean runs" because they do not perform energy and virial computations.

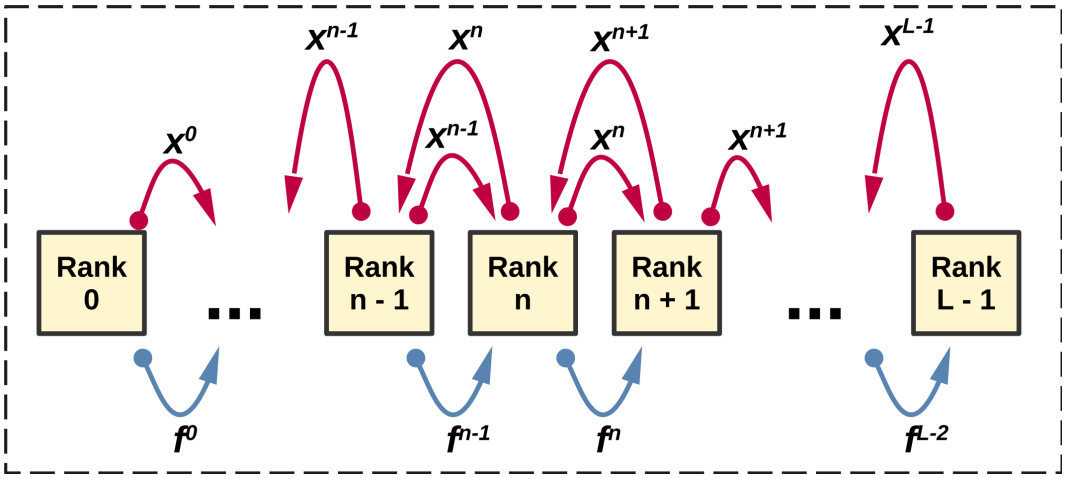


Figure 4.7: An illustration of the ring messaging pattern that is required by the pMD algorithm. Position vectors are sent to both left and right neighbours, whereas force vectors are only sent to the right neighbour.

compute the product $H_U^n \boldsymbol{\eta}^n$. For the latter, since energy and virial computations are not required, we disabled them by turning off the corresponding bits in the force flags argument for efficiency. The remaining work is handled by new pMD kernels (numbered 1-7 in the flowchart):

1. `computeEtas`:
   computation of the displacement vectors, observing the minimum-image convention[4] (MIC),
   $$\boldsymbol{\eta}^n = \mathbf{R}^{n+1} - \mathbf{R}^n - \frac{1}{2} K^{-1} \mathbf{F}^n$$

2. `plusMinus`:
   coordinate perturbation (pre-step to FDM computation)
   $$\mathbf{R}^{n(\pm)} = \mathbf{R}^n \pm \varepsilon \boldsymbol{\eta}^n$$

3. `computeHessianUTimesEtas`:
   computation of the finite difference expression
   $$H_U^n \boldsymbol{\eta}^n = \frac{\mathbf{F}(\mathbf{R}^{n(+)}) - \mathbf{F}(\mathbf{R}^{n(-)})}{2\varepsilon},$$

4. `computePolymerForcesFirstBead`:
   computation of the polymer forces in the first bead (MIC applies here as well)
   $$\mathbf{F}^0_{\text{poly}} = \frac{1}{2}\mathbf{F}^0 + K(\mathbf{R}^1 - \mathbf{R}^0) + \frac{1}{2} H_U^n \boldsymbol{\eta}^n,$$

[4]The minimum image convention ensures that each atom interacts only with that image of every other atom in the periodic system that is closest to it, thereby preventing double interactions.

5. `computePolymerForcesLastBead`:
   computation of the polymer forces in the last bead (MIC applies here as well)
   $$\mathbf{F}_{\text{poly}}^{N-1} = \frac{1}{2}\mathbf{F}^{N-2} - K(\mathbf{R}^{N-1} - \mathbf{R}^{N-2}),$$

6. `computePolymerForcesInBetween`:
   computation of the polymer forces in any other bead (MIC applies here as well)
   $$\mathbf{F}_{\text{poly}}^{n} = \frac{1}{2}(\mathbf{F}^{n-1} - \mathbf{F}^{n}) - K(2\mathbf{R}^{n} - \mathbf{R}^{n+1} - \mathbf{R}^{n-1}) + \frac{1}{2}H_U^n\boldsymbol{\eta}^n$$
   for $0 < n < N - 1$, and

7. `accumulateEtaSqTimesMass`:
   computation of the inter-bead spring energy
   $$Q^n = \frac{1}{2}K||\boldsymbol{\eta}^n||^2.$$

These core functions have been parallelized using OpenMP-based vectorization and multithreading. The computation of the numerical derivative via FDM, carried out by the functions `plusMinus` and `computeHessianUTimesEtas`, has been adapted for stability, as presented in subsection 4.2.4. Support for using GROMACS' main parallelization strategies has been provided, including the possibility to use DD and GPU offloading, as explained in subsections 4.2.6 and 4.2.9.

In GROMACS, the computation of the energies is periodic, as per the frequency provided by the user in the input *.mdp* file. During a pMD simulation, in an energy computation step, the polymer potential energy is computed according to equation (4.3) using the function `accumulateEtaSqTimesMass`. If DD is active, the value returned by this function only includes the contributions of the reference domain's[5] home atoms. Similarly, if the value is computed at the bead level, it only includes the contributions of the reference bead. Therefore, these values have to be summed across domains (if DD is active) and across beads to compute their corresponding values on the polymer level. Moreover, the kinetic energy is computed on the bead level internally elsewhere in the code base. It too must be summed across beads to compute the polymer kinetic energy. These summations are implemented by means of `MPI_Reduce` such that only the main rank of the first bead has these polymer-level values. The polymer temperature is computed by averaging the temperatures of all beads.

[5]Here, "reference domain" is used in place of "*that* domain" or "current domain"; it is to be taken to mean an arbitrary domain that is the point of orientation for a description or example.

After the polymer forces and energies have been computed, the simulation can proceed as in a standard MD simulation: external forces, e.g., the metadynamics force can be added, constraints can be enforced, the coordinates of atoms can be updated etc.

4.2.4 FDM Stabilization

Initial testing of our pMD implementation showed considerable total energy drift in constant-energy simulations. By comparing the output of the FDM computation with that of an analytical Hessian function for a simplified system, we attributed the high total energy drift to the FDM-based error.

In its naïve implementation, for any given bead (except the last bead), the kernel `computeHessianUTimesEtas` computes

$$H_U^n \boldsymbol{\eta}^n = \frac{\mathbf{F}\left(\mathbf{R}^n + \varepsilon \boldsymbol{\eta}^n\right) - \mathbf{F}\left(\mathbf{R}^n - \varepsilon \boldsymbol{\eta}^n\right)}{2\varepsilon} \text{ for } \varepsilon \in (0, 1)\,. \tag{4.11}$$

Here, ε is the discretization step used to approximate an infinitely disappearing value. Hence, to reduce the truncation error ($\propto \varepsilon^2$), its value must be as low as can still be properly represented, which for the above central difference scheme is $\epsilon_m^{1/3}$, where ϵ_m represents the machine precision[6] [109].

To begin with, it must be noted that in equation (4.11), we are not approximating H_U^n, rather we are approximating the *linear transformation* of the displacement vector $\boldsymbol{\eta}^n$ by it i.e., the row vectors of H_U^n are projected onto $\boldsymbol{\eta}^n$. Here, if $\boldsymbol{\eta}^n$ is a large value, then vectors $\mathbf{R}^n + \varepsilon\boldsymbol{\eta}^n$ and $\mathbf{R}^n - \varepsilon\boldsymbol{\eta}^n$ become separated by a large distance. Similarly, if $\boldsymbol{\eta}^n$ is a very small value, the vectors come too close to each other. Both cases introduce error in the FDM calculation. To counter this, we adapted the calculation using a stabilization technique that is very similar to the one provided by Ref. [110]. We define the unit vector,

$$\hat{\boldsymbol{\eta}}^n = N\boldsymbol{\eta}^n, \quad \text{where} \quad N = \frac{1}{||\boldsymbol{\eta}^n||_\infty}. \tag{4.12}$$

Moreover, analogously to equation (4.11), we have,

$$H_U^n \hat{\boldsymbol{\eta}}^n = \frac{\mathbf{F}\left(\mathbf{R}^n + \varepsilon \hat{\boldsymbol{\eta}}^n\right) - \mathbf{F}\left(\mathbf{R}^n - \varepsilon \hat{\boldsymbol{\eta}}^n\right)}{2\varepsilon}. \tag{4.13}$$

Now, substituting (4.12) in (4.13), we get,

$$H_U^n \boldsymbol{\eta}^n = \frac{\mathbf{F}\left(\mathbf{R}^n + \varepsilon N \boldsymbol{\eta}^n\right) - \mathbf{F}\left(\mathbf{R}^n - \varepsilon N \boldsymbol{\eta}^n\right)}{2\varepsilon N}, \tag{4.14}$$

[6] The machine precision of a computer is the smallest positive floating-point number that, when added to 1, results in a value distinguishable from 1.

which reduces the error of (4.11) because $||N\boldsymbol{\eta}^n||_\infty = 1$ is self-balancing. Equation (4.14) is therefore a more robust way to compute the required vector than equation (4.11).

In our implementation, the kernel `computeEtas` determines and returns $||\boldsymbol{\eta}^n||_\infty$. When DD is active, the returned value represents the L^∞ norm of only the *part* of the global vector that is associated with the reference domain. To ensure that *all* domains have the value of $||\boldsymbol{\eta}^n||_\infty$ computed on the global vector, `MPI_Allreduce` is used, which (we speculated at the outset) should reduce parallelism by imposing an implicit barrier on the DD communicator in *each* time step. Therefore, to retain performance while simultaneously improving the stability, we decided to give the user the possibility to apply the stabilization periodically. The period of applying stabilization is decided by a frequency provided by the user in the input *.mdp* file. In the time steps in which the stabilization is not performed, the $||\boldsymbol{\eta}^n||_\infty$ value from the previous stabilization step is used as an approximation, under the assumption that $||\boldsymbol{\eta}^n||_\infty$ changes continuously and slowly over time.

4.2.5 Centre of Mass Motion Removal

In an MD simulation, the numerical integration introduces a slow error in the centre-of-mass velocity, which correspondingly leads to a slowly growing error in the kinetic energy. In case of very long simulations, if this accumulation is not checked, it can result in incorrect or unrealistic results [111]. This effect has also been attributed to velocity rescaling in finite-temperature simulations [86, 112]. To counter this effect, MD codes commonly utilize a strategy called *centre-of-mass motion* (COMM) removal in which the centre-of-mass velocity is subtracted from each atomic velocity.

The GROMACS-native COMM removal cannot be employed straightforwardly in pMD simulations because it can only be applied to each simulation (i.e., bead) individually, which leads to inconsistencies in the inter-bead force computations. Therefore, it was necessary to develop a COMM removal strategy that considers the fictitious polymer as a whole. Our strategy removes COMM contributions in both polymer forces as well as velocities.

Polymer Forces Removal

Before the temperature coupling is applied, the polymer forces comprise of purely conservative terms. Hence, by Newton's third law, the total force on the polymer must be equal to zero, and any residual, non-zero value can be attributed to floating-point error. Now, this requirement can be expressed

as,

$$\text{if} \quad \mathbf{F}^n_{\text{poly}} = \begin{pmatrix} \mathbf{f}^n_0 \\ \mathbf{f}^n_1 \\ \dots \\ \mathbf{f}^n_{Q-1} \end{pmatrix}, \; \mathbf{f}_{\text{COM}} = \sum_{n=0}^{N-1} \sum_{k=0}^{Q-1} \mathbf{f}^n_k \stackrel{!}{=} \mathbf{0}.$$

To prevent the accumulation and propagation of this error, it can be removed periodically, with a frequency that is provided by the user in the input *.mdp* file. The procedure for removal is given as follows (actions are performed by all domain ranks):

1. compute $\mathbf{f}^n_{\text{local}} = \sum_{k=0}^{Q^*-1} \mathbf{f}^n_k$ locally (where Q^* is the number of atoms present locally),
2. if DD is active, all-reduce $\mathbf{f}^n_{\text{bead}} = \sum_{\text{domains}} \mathbf{f}^n_{\text{local}}$ across domains by summation,
3. all-reduce $\mathbf{f}_{\text{COM}} = \sum_{n=0}^{N-1} \mathbf{f}^n_{\text{bead}}$ across beads by summation,
4. compute the average atomic force $\mathbf{f}_{\text{avg}} = \frac{1}{QN} \mathbf{f}_{\text{COM}}$, and,
5. set $\mathbf{f}^n_k \leftarrow \mathbf{f}^n_k - \mathbf{f}_{\text{avg}} \quad \forall\, 0 \leq k < Q^*$.

By subtracting the average atomic force on the whole polymer from each atomic force, we thus ensure that this residual, non-zero value is removed.

Velocity Removal

The centre-of-mass velocity is obtained as the time-derivative of the centre of mass of the fictitious polymer. If there is no COMM, it must be equal to zero, i.e.,

$$\dot{\mathbf{r}}_{\text{COM}} = \frac{\sum_{n=0}^{N-1} \sum_{k=0}^{Q-1} m_k \dot{\mathbf{r}}^n_k}{N \sum_{k=0}^{Q-1} m_k} \stackrel{!}{=} \mathbf{0},$$

where entries m_k represent the main diagonal entries of $\mathcal{M}$ viz., the fictitious atomic masses.

Since the denominator of the above equation does not change during the simulation, it can be computed at the beginning. During the simulation, COMM removal can be periodically enforced by the following procedure (actions are performed by all domain ranks):

1. compute $\mathbf{p}^n_{\text{local}} = \sum_{k=0}^{Q^*-1} m_k \dot{\mathbf{r}}^n_k$ locally,
2. if DD is active, all-reduce $\mathbf{p}^n_{\text{bead}} = \sum_{\text{domains}} \mathbf{p}^n_{\text{local}}$ across domains by summation,
3. all-reduce $\mathbf{p}_{\text{COM}} = \sum_{n=0}^{N-1} \mathbf{p}^n_{\text{bead}}$ across beads by summation,

4. compute the centre-of-mass velocity $\dot{\mathbf{r}}_{\text{COM}} = \frac{1}{N \sum_{k=0}^{Q-1} m_k} \mathbf{p}_{\text{COM}}$, and,
5. set $\dot{\mathbf{r}}_k^n \leftarrow \dot{\mathbf{r}}_k^n - \dot{\mathbf{r}}_{\text{COM}} \quad \forall\, 0 \leq k < Q^*$.

We implemented both removals as separate functions. The force removal is performed immediately after computing the polymer forces. The velocity removal is performed at that location in code where the COMM removal of standard MD is performed.

4.2.6 pMD and DD

DEFINITION : TWIN

The replica of an atom or domain of the reference bead in another bead.

Out-of-the-box DD support is not possible for the pMD algorithm because of their incompatibilities in regard to the inter-bead communication. In a pMD simulation, each atom in the reference bead has a corresponding *twin** atom in every other bead. Recall from subsection 4.2.3 that during each inter-bead communication event, every bead must acquire a vector array[7] of its atoms' twins from a neighbouring bead. With DD active, the exact procedure for this communication becomes ambiguous. The first incompatibility arises from the fact that the communication can no longer be performed by a *single* MPI task —simply because no single MPI task possesses the global vector array. Now, there are two possible solutions for this problem. In the first solution, the main rank of the reference bead gathers the global vector array from its domain ranks and sends it to the main rank of the neighbouring bead, which then distributes the vectors correctly to its domain ranks. In the second solution, each domain rank of the reference bead communicates directly and independently with those domain ranks in the neighbouring bead which —prior to the communication event —have been identified as its communicating partners.

However, after accepting either solution, the second incompatibility becomes apparent: in a pMD simulation, each atom's twin is slightly spatially displaced with respect to it[8]. Hence, it cannot be ruled out that, at the time of communication, an atom's twin could have been *re-partitioned** to a different domain in the neighbouring bead as shown in figure 4.8. In the first solution, this complicates the problem of distributing the vectors to the correct domain ranks. In the second solution, it complicates the problem of

[7] Here, a vector $\mathbf{V} \in \mathbb{R}^{3Q}$ is being re-interpreted as an array of 3D vectors i.e., $V = \{\mathbf{v}_i | \mathbf{v}_i \in \mathbb{R}^3 \,\forall\, i \in [0, Q)\}$.
[8] Each atom is one time step behind/ahead of its right/left twin on the level of the BD given by equation (4.1).

identifying each domain rank's communicating partners.

DEFINITION : RE-PARTITIONING

GROMACS periodically re-assesses the allocation of atoms to domains based on the geometric locations of the (constantly moving) atoms. The frequency of re-partitioning is either provided as input in the *.mdp* file or internally estimated.

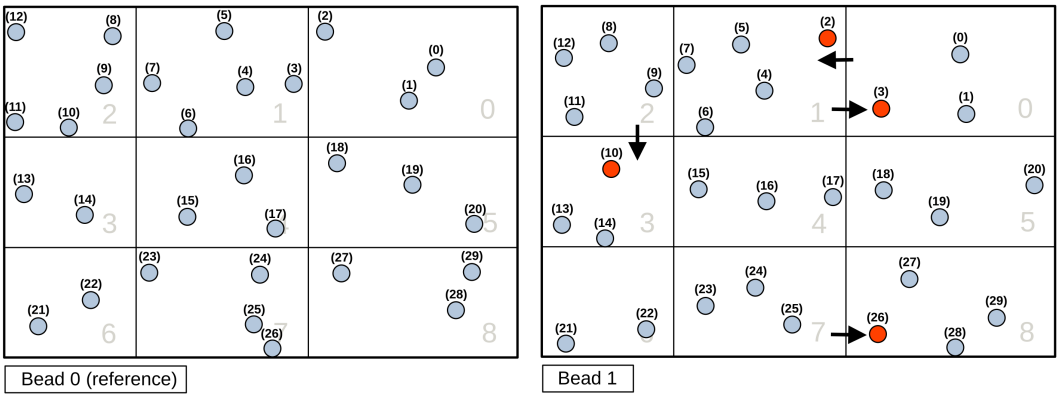


Figure 4.8: Different re-partitioning of atoms in beads 0 and 1 results in a problem for communication in the pMD algorithm. Here, atoms with global atomic indices 2, 3, 10 and 26 are no longer in the same domains in bead 1 as in bead 0.

To solve these problems, we developed algorithms based on the above-mentioned first and second solutions, which we refer to as the *centralized* and *decentralized twin-finding* algorithms, respectively.

4.2.7 Centralized Twin-Finding Algorithm

DEFINITION : GLOBAL ATOMIC INDEX

A unique identifier for every atom in a GROMACS simulation. In a pMD simulation, GAIs are conserved across beads.

Of the two algorithms, the centralized twin-finding algorithm follows the simplest approach. It consists of a twin-finding phase in which the inter-bead communication pattern is predetermined and a communication phase in which the communication actually takes place.

Twin-Finding Phase

The twin-finding phase of this algorithm takes place immediately upon re-partitioning. In this phase, the reference bead's main rank begins by gathering the *global atomic indices*⋆ (GAIs) of all atoms from its domain ranks in a list, and saves the domain GAI counts and GAI offsets so that the order of the sub-lists with respect to the domain indices is preserved. Having participated in the gather operation, the non-main ranks have no further work to do and thus return from the function.

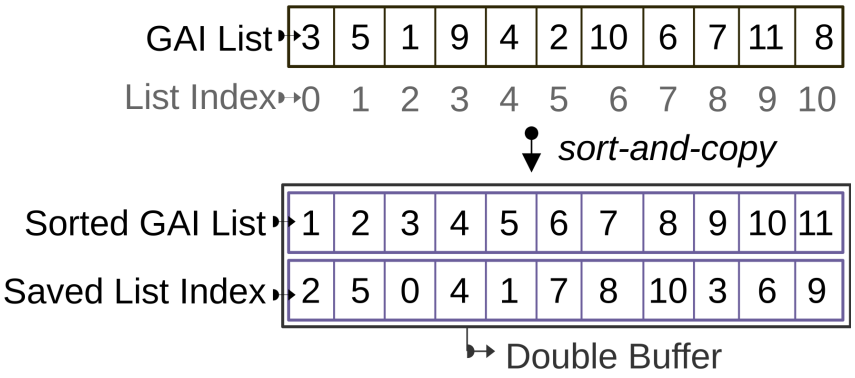


Figure 4.9: A GAI list is sorted and copied into an output double buffer that saves the list indices in its second row.

Next, the main rank of the reference bead sends its gathered GAI list to its neighbours and receives the same from its neighbours. Once the transfers are completed, the main rank executes a sort-and-copy operation on each received GAI list, requiring $\mathcal{O}(n \log n)$ accesses, copying the sorted items into an output double buffer. This operation takes care to save the index of every item in the second row of the output double buffer, as shown in figure 4.9.

Finally, the main rank iterates through its own GAI list via a *loop index*[9], searching for each iterated GAI in the sorted double buffer using the well-known binary search algorithm, which requires $\mathcal{O}(\log n)$ accesses per search. All GAIs are required to be found in this process, failing which an error is thrown. The result of each search gives the index in the received GAI list at which the iterated GAI was found[10], which is then saved at the current loop index in an array called the *scatter mapper*⋆. This concept is explained by figure 4.10 using an example. The mapping is processed for the left and right neighbours, resulting in two scatter mappers.

[9] This simply refers to the `for`-loop counter used to iterate through the list.

[10] The double buffer is required to propagate this information, since it preserves the indices of the sorted list.

DEFINITION : SCATTER MAPPER

An array used to record the order-preserving mapping between elements of an incoming message and the domains in the receiving bead to which they must be sent.

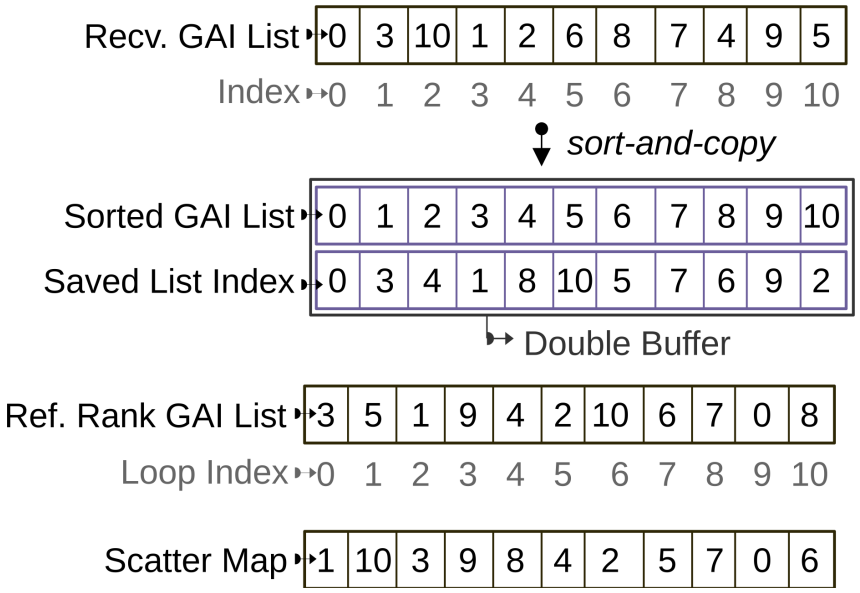


Figure 4.10: An example illustrating the concept of the scatter mapper. Here, the GAI 3 at loop index 0 in the reference rank's GAI list is found at index 1 in the received GAI list, which is saved at index 0 in the scatter mapper. Similarly, the GAI 5 at loop index 1 in the reference rank's GAI list is found at index 10 in the received GAI list, which is saved at index 1 in the scatter mapper. This procedure is repeated for all the GAIs in the reference rank's GAI list, resulting in the scatter mapper.

Communication Phase

During the communication phase, using the saved domain GAI counts and offsets, the sending bead's main rank gathers the vector array from its domain ranks in the same order as the GAI sub-lists in the twin-finding phase and sends it to the receiving bead's main rank. Upon receiving the vector array, the receiving bead's main rank copies the vectors from the receive buffer into a scatter buffer according to the mapping provided by its scatter mapper i.e., `scatterBuffer[ii] = receiveBuffer[scatterMapper[ii]]`. Finally, the contents of the scatter buffer are scattered to the domain ranks. This ensures that every domain rank of the receiving bead gets the vectors of its home atoms' twins in the correct order from the sending bead. This procedure is illustrated by figure 4.11.

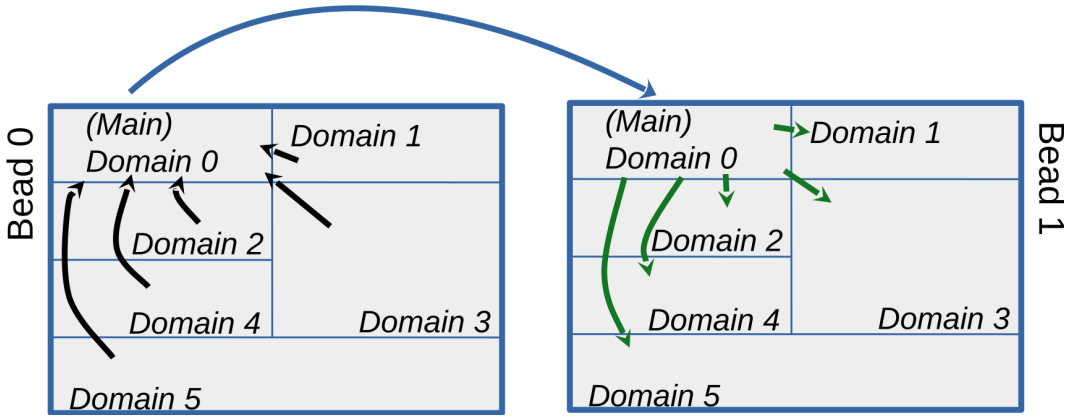


Figure 4.11: An illustration of the communication phase of the centralized twin-finding algorithm. The black arrows indicate the gathering of the vectors by the sending bead's main rank from its domain ranks. The blue arrow indicates the sending of a unified message by the sending bead's main rank to the receiving bead's main rank. The green lines indicate the scattering of the received vectors by the receiving bead's main rank to its domain ranks according to its scatter mapper.

The centralized twin-finding algorithm introduces one gather and one scatter operation in *each* of the three communication events, in *every* time step of the pMD algorithm. Theoretically, this should be detrimental to the parallel performance because each gather/scatter operation imposes an implicit barrier on the DD communicator, which increases the synchronization overhead and reduces parallelism.

On the other hand, the relatively low complexity of this algorithm allowed us to implement it as a first step towards tackling the problem of providing DD support for our pMD implementation. We tested the implementation for correctness using the outputs of simulations run without DD as references of comparison. Finally, we conducted a performance analysis of the implementation by benchmarking on the JUWELS Booster, the results of which are presented in subsection 4.3.5.

4.2.8 Decentralized Twin-Finding Algorithm

To eliminate the gather/scatter operations of the centralized twin-finding algorithm, we conceptualized the decentralized twin-finding algorithm, which makes use of point-to-point communication patterns only. Just like the former, the latter also has a twin-finding phase and a communication phase. In the twin-finding phase, which also occurs immediately upon repartitioning, associations are established between each domain of the reference bead and *one or more* domains of the reference bead's neighbours that

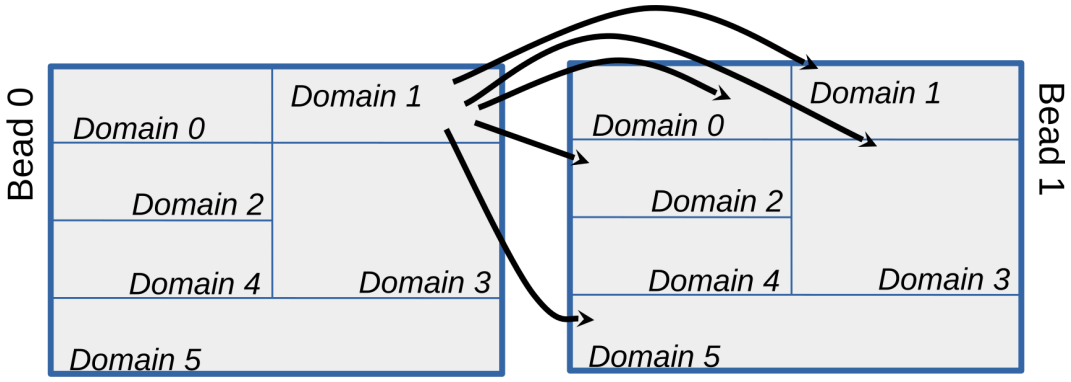


Figure 4.12: An illustration of the communication phase of the decentralized twin-finding algorithm. The black arrows indicate the direct and parallel message passing between inter-bead ranks and their communicating partners as per the associations made in the twin-finding phase.

contain at least one of its home atoms' twins. In the communication phase, multiple messages are exchanged parallely and directly according to the associations made in the twin-finding phase, as shown in figure 4.12.

Search Space Ranks

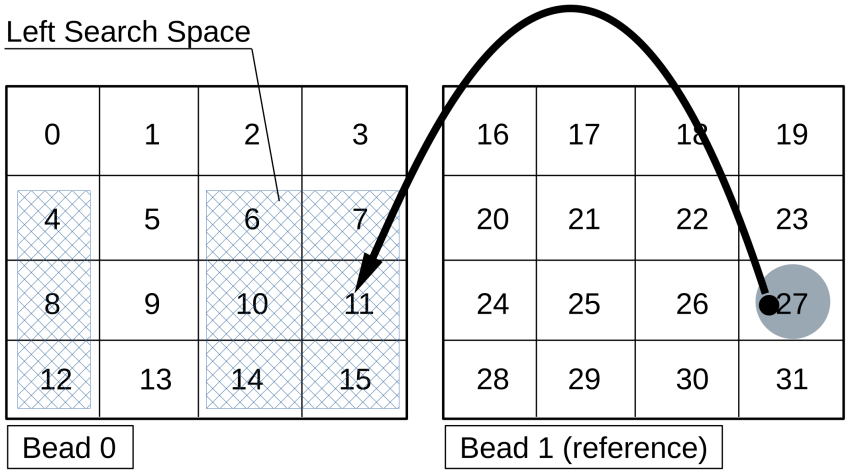


Figure 4.13: An example illustrating the construction of the left search space for rank 27 in a 2D with 16 PP ranks per bead. Here, rank 27 in bead 1 has twin rank 11 in bead 0. Therefore, the left search space of rank 27 is composed of rank 11 and those ranks that are adjacent to rank 11 i.e., ranks 6, 7, 4, 10, 11, 8, 14, 15 and 12. Ranks 4, 8 and 12 are adjacent to rank 11 by periodicity.

In this algorithm, since domain ranks must communicate with partners *outside* their DD communicator, a suitable MPI communicator must be created

before starting the simulation. Accounting for the possibility that PP and PME ranks may be interleaved, a new communicator called the *inter-bead* communicator is created at the start of the simulation by splitting GROMACS' world-level communicator based on the condition that a given rank must be a PP rank. This communicator identifies a set of MPI ranks given by $\{0, 1, 2, ...Nn_{pp} - 1\}$, where n_{pp} is the number of PP ranks per bead. For convenience, in this subsection, unless specified otherwise, the word *rank* will be used to mean an MPI rank in the inter-bead communicator.

To simplify the following description, the procedure for only one inter-bead direction will be considered, given that the same procedure applies for the other direction as well. To begin with, it is necessary to construct a *search space** for each rank, under the reasonable assumption that atoms will not cross two domains in any single re-partitioning event, which holds true if the maximum force is not too large. As per our definition, for 1D, 2D and 3D DDs, there can be 3, 9 and 27 search space ranks, respectively. Figure 4.13 illustrates the concept by means of an example.

DEFINITION : SEARCH SPACE

The set of ranks of those domains in the neighbouring bead, in which all of the reference domain's home atoms' twins are assumed to be found. We identified this to be the set of ranks that are adjacent to the reference rank's *twin rank* in the neighbouring bead, including the twin rank itself.

Twin-Finding Phase

In the twin-finding phase, since different ranks in a search space can have different home atom counts, the reference rank sends its home atom count to all ranks in its search space. Hence, it also receives the home atom count of each of its search space ranks. After this initial communication, it has an array of GAI counts and another of GAI offsets, which can be taken together to indicate distinct sub-arrays of a single, over-sized receiving buffer in which the GAI lists of the search space ranks are to be received. Next, it sends its GAI list to all ranks in its search space, and accordingly receives a GAI list from each rank in its search space.

In this algorithm, two kinds of mapping need to be processed, one for outgoing messages and another for incoming messages.

To process the mapping of outgoing messages, the GAI list of the reference rank is first sorted using the previously mentioned sort-and-copy technique,

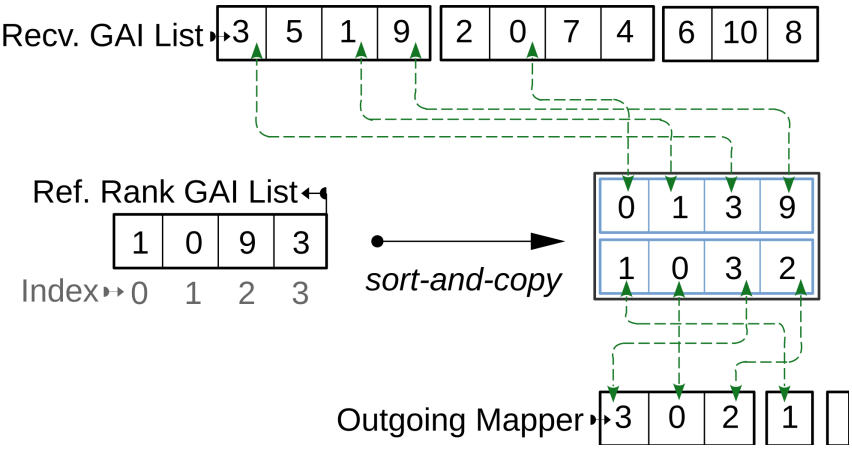


Figure 4.14: An example illustrating the processing of the outgoing mapper with search space size 3. Here, received GAI 3 is found at index 3 in the reference rank's GAI list, which is saved in the outgoing mapper. Received GAI 5 is not findable in the reference rank's GAI list, so control continues with the next iteration. Next, received GAI 1 is found at index 0 in the reference rank's GAI list, which is also saved in the outgoing mapper. This process is repeated for all the GAIs in the received GAI list. The mapper shows that the vectors at indices 3, 0 and 2 of the reference rank must be sent to the first search space rank, the vector at index 1 must be sent to the second search space rank, and no vectors need to be sent to the third search space rank.

using an output double buffer as explained in subsection 4.2.7 (and demonstrated in figure 4.9). Then, iterating through the received GAI list, the iterated GAI is searched for in the sorted double buffer using the binary search algorithm. If it cannot be found, it is not considered an error since another rank in the reference bead could have it. On the other hand, if it is found, the result of the search operation (which is the index in the reference rank's GAI list at which the iterated GAI was found), is recorded in the *outgoing mapper** in distinct sub-arrays that are reserved for the search space ranks, as illustrated in the example of figure 4.14.

DEFINITION : OUTGOING MAPPER

A device (an array with book-keeping structures) used to record the mapping of the contributions of the reference rank to its search space ranks.

In the centralized algorithm, the received vector array of the communication phase has the *same size and order* as the received GAI list of the twin-finding

phase. However, in this algorithm, the received GAI list of the twin-finding phase references all atoms in the search space ranks, whereas the received vector array of the communication phase contains *only the contributions that are relevant to the reference rank*. Moreover, the order of the contributions is determined by the outgoing mappings of the sending ranks in its search space. Therefore, the incoming mapping must take care to acknowledge these differences, and is consequentially a more involved procedure than in the centralized algorithm.

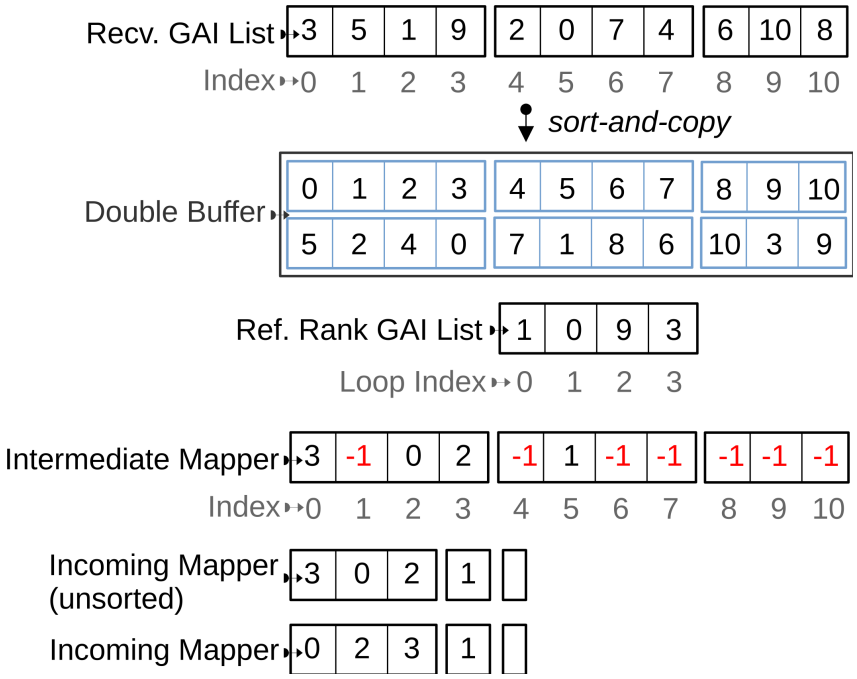


Figure 4.15: An example illustrating the processing of the incoming mapper with search space size 3. Here, reference rank GAI 1 is found at index 2 in the received GAI list. Hence, its loop index 0 is saved at index 2 in the intermediate mapper. Similarly, reference rank GAI 0 is found at index 5 in the received GAI list. Hence, its loop index 1 is saved at index 5 in the intermediate mapper. The procedure is repeated for GAIs 9 and 3. Next, the incoming mapper is extracted from the intermediate mapper. Finally, it is sorted within sub-arrays reserved for the search space ranks, resulting in the final incoming mapper.

DEFINITION : INTERMEDIATE MAPPER

A temporary mapper array used to record the contributions of a reference rank's search space ranks to it without explicit order information.

To begin with, an oversized array called the *intermediate mapper** is initialized[11] with the value -1. Next, the received GAI list is sorted using the sort-and-copy technique, resulting in the double buffer that has been previously discussed. Then, iterating through the GAI list of the reference rank via a loop index, the iterated GAI is searched for in the double buffer. If the iterated GAI could not be found, an error is thrown because it implies that an atom's twin does not exist in the search space, which could be a sign that the core assumption of the algorithm —that atoms will not cross two domains in any single re-partitioning event —was proven to be false. Now, at the location in the intermediate mapper accessed using the result of the search operation as index (which is the index in the received GAI list at which the iterated GAI was found), the loop index is saved.

At the end of this procedure, the intermediate mapper contains the information about the contributions of the individual search space ranks, with the background value -1 signalling atoms that are irrelevant to the reference rank. Accordingly, the values which are not -1 are extracted into an array known as the *incoming mapper**. Finally, this incoming mapper is sorted *within* the distinct sub-arrays that are reserved for the search space ranks. The incoming mapper then reflects both the contributions of the search space ranks as well as the order in which they are sent to the reference rank. This procedure is explained by means of an example in figure 4.15.

DEFINITION : INCOMING MAPPER

A device (an array with book-keeping structures) used to record the order-preserving mapping of the contributions of a reference rank's search space ranks to it.

At the end of the twin-finding phase, every rank knows which ranks in the neighbouring bead need which vectors from it by consulting its outgoing mapper. Similarly, each rank knows which ranks in the neighbouring bead will provide it with which vectors it needs and in what order, by consulting its incoming mapper.

Communication Phase

During the communication phase, while sending a vector array, a single, over-sized communication buffer is used. Reading from the vector array to be sent, vectors are copied into the communication buffer via the indices provided by the outgoing mapper i.e. `commBuffer[ii] = sendBuffer[outgoingMapper[ii]]`. Then, iterating over the search space ranks, if

[11] This "background" value works because GAIs are always positive.

the count of vectors to be sent to the iterated rank as reflected by the outgoing mapper is non-zero, a message is sent to the iterated rank using the distinct sub-array of the communication buffer reserved for it.

Similarly, while receiving a vector array, a single, over-sized communication buffer is used. Iterating over the search space ranks, if the count of vectors to be received from the iterated rank as indicated by the incoming mapper is non-zero, a message is received from the iterated rank in the distinct sub-array reserved for it. After all messages have been received, the output vector array of the function reads the vectors from the communication buffer via the indices provided by the incoming mapper i.e., `receiveBuffer[ incomingMapper[ii]] = commBuffer[ii]`. Ultimately, each rank in the receiving bead will acquire the vectors of its home atoms' twins from the sending bead.

In this algorithm, a rank could send or receive up to 27 messages in a communication event. Since asynchronous communication is employed, the order of completion of receiving/sending cannot be predicted. In code, this means that fixing the order of calls to `MPI_Wait` would necessarily reduce the asynchronicity in an unpredictable way. Therefore, to retain maximum asynchronicity, since waiting for the completion of communication takes place in two distinct phases, effort was taken to arrange `MPI_Requests` stage-wise in contiguous arrays such that a single `MPI_Waitall` can be used at each stage.

We implemented the decentralized twin-finding algorithm and, as in the case of the centralized twin-finding algorithm, tested it for correctness using the outputs of simulations run without DD as references for comparison. Finally, we conducted a performance analysis of the implementation by testing on the JUWELS Booster, whose results are presented in subsection 4.3.5.

4.2.9 pMD and GPU Offloading

The key procedures in a time step of a GROMACS simulation are the force computation and what is commonly referred to as the "update", which is the combined procedure of updating atomic coordinates (by numerical integration) and applying constraints. The force computation is the most time-consuming procedure, accounting for $95\% - 98\%$ of the total runtime (as ascertained by profiling, see subsection 4.3.4). It is comprised of largely independent part computations that can be performed in parallel, with their results being combined at the end to compute the overall force vector.

By means of command-line options given to the `mdrun` program, the user can specify which of these part computations are to be performed by CPUs and which by GPUs. The same applies for the update procedure as well. These options are: `-bonded` (bonded interactions), `-nb` (short-range, non-

bonded interactions), `-pme` (long-range, non-bonded interactions by PME) and `-update` (the update procedure).

DEFINITION : FORCE-OFFLOADING MODE

In this mode, only the force computations are offloaded to GPUs whereas the update is performed by CPUs.

Recall from subsection 4.2.3 that the non-polymer forces of the simulated system are computed by the function called `do_force`. Since this function internally handles the CPU-GPU data transfers, *force-offloading** works out of the box for our pMD implementation. As in the case of standard MD simulations, the user needs only specify any combination of the above command-line options in their `mdrun` call, followed each time by `gpu`. Of course, this requires a GPU-enabled GROMACS build, which is configured using the CMake option `-DGMX_GPU` followed by either `=CUDA`, `=OpenCL` or `=SYCL`, as per availability and requirement.

As a side note, it must be mentioned that since GROMACS v2020, we have the *GPU-resident* mode feature, in which both the force computation and the update are performed on GPUs [107, 113]. This eliminates the CPU-GPU transfer overheads, which, depending on the hardware, can potentially speed up simulations twofold [103]. However, the resident-mode can only be accessed when certain conditions are met e.g., in our pMD implementation code base at v2024, only the leap-frog integrator is supported (as ascertained by the initial test reported in subsection 4.3.6). Moreover, the new kernels 1-7 of the pMD algorithm as listed in subsection 4.2.3 have not been written for processing on GPUs, which means they must necessarily run on the CPUs. Therefore, currently, *pMD simulations do not yet support the GPU-resident mode*.

4.2.10 Support for Metadynamics using PLUMED

PLUMED (PLUgin for MolEcular Dynamics) is an open-source and a community-developed library for enhanced-sampling algorithms, free-energy methods and tools to analyze MD simulation data [87]. Some MD codes e.g., Amber, CP2K and LAMMPS, provide support for the PLUMED library natively, whereas others e.g., Quantum ESPRESSO and GROMACS, can be patched to provide support [108].

If we were to run a pMD simulation using an out-of-the-box PLUMED-patched GROMACS build, every bead would make independent calls to the PLUMED library at the required time points. Since these calls have access to the atomic coordinates of the reference bead only, computing a CV

across beads is impossible. To enable the definition of CVs that are functions of atomic coordinates of multiple beads, such as the end-to-end distance from equation (4.10), we have modified the `Custom` function class within the PLUMED library as follows.

Firstly, we introduced a new, optional keyword `MULTIREP` to allow the user to signal a multi-replica simulation. Secondly, we introduced the optional keyword `REPLICAS=...` that allows the user to provide a comma-separated list of indices of the *active beads** to the class. Since the MPI tasks of all active beads must participate in an `MPI_Allgather` call *every* time the `Custom` function is computed, using the `REPLICAS` keyword to limit the number of active beads can greatly ameliorate the potential performance loss associated with this all-to-all communication. Finally, we adapted the functionality of the `FUNC` keyword such that its associated expression may reference variable names suffixed with a bead index to refer to a quantity computed in that particular bead.

Definition : Active Bead

A bead is said to be active if its MPI tasks communicate their atomic coordinates to each other for the computation of CVs of the fictitious polymer.

Following the above changes in the PLUMED library, which were made in a yet unreleased version initialized from v2.9.0, it was also necessary to adapt the PLUMED patch for GROMACS such that the calls to PLUMED functions acknowledge our pMD implementation and its specific details. The examples shown in listing 4.1 highlight the differences between standard MD and pMD in the usage of the `Custom` function in a PLUMED input file. Figure 4.16 visualizes the example CV *polymer end-to-end distance* for pMD simulations.

```
# standard MD
com: COM ATOMS=1-22
p: POSITION ATOM=com cm: CUSTOM ARG=p.x,p.y,p.z
FUNC=sqrt(x*x + y*y + z*z) PERIODIC=NO

# Path MD
com: COM ATOMS=1-22
p: POSITION ATOM=com
e2e: CUSTOM MULTIREP ARG=p.x,p.y,p.z
FUNC=sqrt((x0-x7)*(x0-x7)+(y0-y7)*(y0-y7)+(z0-z7)
```

```
*(z0-z7)) PERIODIC=NO REPLICAS=0,7
```

Listing 4.1: An example demonstrating the difference between PLUMED inputs for standard MD and pMD. For the former, the distance of the centre of mass of atoms 1-22 from the origin is computed. For the latter, the distance between the centre of mass of atoms 1-22 in bead 0 and the centre of mass of the same set of atoms in bead 7 is computed, which is referred to as the polymer end-to-end distance (see figure 4.16 for an illustration of the CV for this input).

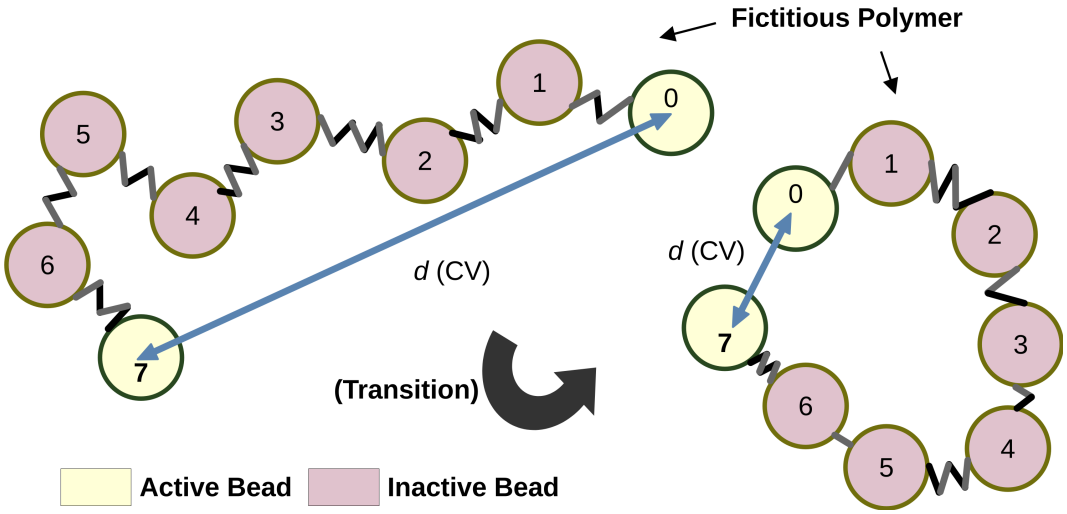


Figure 4.16: An illustration of an example CV for pMD simulations referred to as the polymer end-to-end distance. The distance between the centre of mass of a specific set of atoms in bead 0 and the centre of mass of the same set of atoms in bead 7 is monitored to identify very different states of the fictitious polymer.

After testing the functionality of the `Custom` class in isolation, we tested the entire software chain for correctness by running metadynamics simulations of simple test systems.

4.3 Testing

To quantitatively describe the overall performance of our pMD implementation, we conceptualized tests and performance models for its various components. However, owing to bug fixes, performance improvements etc., our pMD implementation evolved considerably during the course of this project. Therefore, as and when our pMD implementation advanced in its maturity, it was ”frozen” in version control as a series of tags. The results presented in this subsection are for pMD v1.3.10. A further release pMD

v1.4.0 is planned, which addresses minor bugs and integrates measures for performance improvement.

This section presents the test results and is organized as follows. We begin by listing all simulation systems used in our tests. Next, we present the tests relating to the correctness of our pMD implementation. Following this, we discuss the performance of the pMD kernels described in subsection 4.2.3. Next, we describe the performance analyses of our pMD implementation with and without the algorithms for DD support detailed in subsections 4.2.7 and 4.2.8. Then, we present the results of the performance tests of our pMD implementation using force-offloading to GPUs. Finally, we show the excellent weak-scaling performance of our pMD implementation with and without PLUMED using nearly the whole JUWELS Booster.

4.3.1 Systems used for testing and benchmarking

The following simulation systems have been used for the various tests described in this section.

Many-body LJ ("DROPLET") with ***27 atoms*** A "droplet" of 27 atoms with unit mass, unit σ and unit ε is distributed on the vertices of a regular cubic grid of side 4.5 nm before being perturbed slightly, randomly [12]. It is centred in a cubic simulation box of side 16.0 nm.

Acetone in water ("ACE") with $\sim \mathbf{1.4k}$ ***atoms*** An acetone molecule is centred in a cubic simulation box of side 2.4 nm, which is then filled with water molecules modeled using the Simple Point Charge (SPC) model, which is GROMACS' default solvent model. The force fields, starting configuration and other simulation parameters were obtained from Ref. [114].

Lysozyme in water ("LYSOBENZ") with $\sim \mathbf{50k}$, $\sim \mathbf{100k}$, $\sim \mathbf{150k}$ *and* $\sim \mathbf{200k}$ ***atoms*** The protein lysozyme T4 L99A variant in complex with benzene is placed in cubic simulation boxes of sides 8.0 nm, 10.2 nm, 11.6 nm and 12.7 nm, which, following solvation with SPC water molecules, results in systems with the above-mentioned atom counts. The force fields, starting configuration and other simulation parameters were obtained from Ref. [115].

Human adenosine receptor ("A2A-ZMA") with $\sim$ $\mathbf{150k}$ ***atoms*** The class A GPCR human adenosine receptor type 2A (hA2AR) in complex with its high-affinity antagonist ZM241385, embedded in a lipid bilayer and solvated in explicit water. The simulation box is sized $14.3 \times 10.8 \times 9.6$ nm. The force fields, starting configuration and other simulation parameters were obtained from Ref. [116].

[12] Computers cannot generate truly random values and thus make use of functions to generate *pseudo-random* values.

4.3.2 Correctness of the pMD Implementation

The initial phase of testing for correctness involved a series of highly simplified systems in which the total energy conservation were tested. Some examples are:

1. a single-atom NVE simulation with 480 beads,
2. a 64-atom LJ NVE simulation cooled to a temperature close to zero,
3. a two-atom simulation with a single bond between the atoms,
4. a three-atom simulation with a single angle between the atoms, etc.

These tests specifically targeted basic aspects of the implementation. After the initial phase, we conducted several higher-level tests ranging from simple to realistic simulation systems, as detailed in this subsection. These tests followed the given procedure:

1. energy minimization to as low an energy state as possible,
2. standard MD equilibration to target temperature, and (if applicable) target pressure,
3. LD simulation in which bead-starting configurations are generated by writing to the output trajectory in every frame,
4. LD pMD equilibration to target temperature, and
5. long NVT and/or NVE pMD simulation.

Here, for all equilibration and NVT simulations, LD was used via the *stochastic dynamics* (`sd`) integrator, as it was found to be effective in sampling the distribution given by equation (4.2). For all pMD simulations, the FDM stabilization was enabled with maximum frequency. The COMM correction was enabled only for the NVE pMD simulations. The timestep sizes were determined by trial and error. Unless otherwise specified, all simulations were conducted in single precision.

DROPLET

This simulated system represents the simplest test since it features only non-bonded interactions modeled using the LJ potential.

Following energy minimization, the simulated system was equilibrated to a temperature of 70 units[13] by means of an LD simulation (200,000 timesteps with $\Delta t = 1$ fs). Following this, a short LD simulation was conducted in which the configuration of the system was written to the output trajectory

[13]Here, we refer to "units" instead of Kelvin because the simulated system uses unit sigma and epsilon values i.e., the so-called *reduced units*.

in every timestep. Using each frame of this trajectory as the bead-starting configuration, a fictitious polymer of 480 beads was constructed.

Next, this fictitious polymer was equilibrated by means of an LD pMD simulation (500,000 timesteps with $\Delta t = 1$ fs) to 70 K. As illustrated by figure 4.17, the non-polymer potential energies of all beads were seen to cluster closely around a specific value. Moreover, the target temperature was stably maintained throughout the course of the simulation.

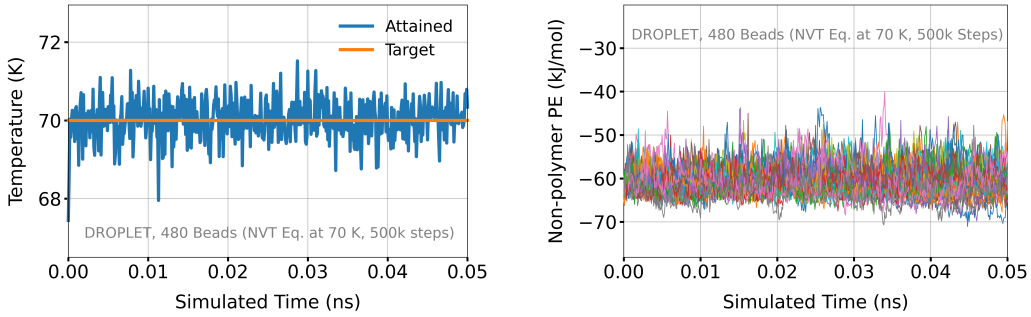


Figure 4.17: Left subfigure shows the variation of temperature during the LD pMD equilibration simulation of DROPLET. Right subfigure shows the non-polymer potential energies of all beads. The near-constant value is indicative of attainment of thermal equilibrium.

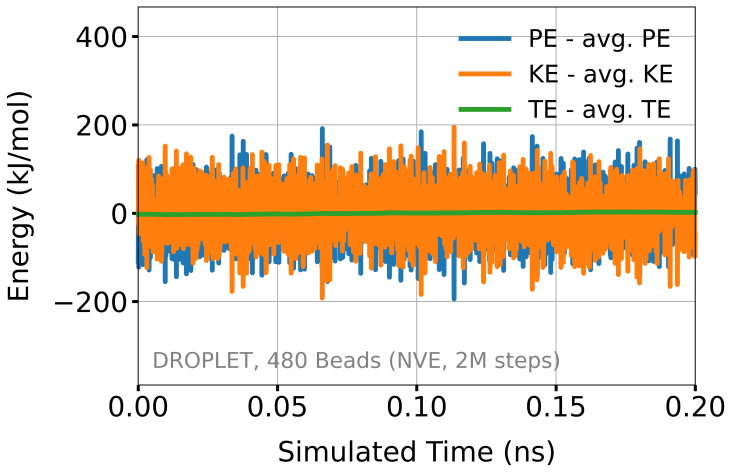


Figure 4.18: A plot showing the polymer potential, kinetic and total energies in an NVE simulation of DROPLET. It can be seen that the total energy is adequately conserved.

Finally, an NVE pMD simulation (2 million timesteps with $\Delta t = 0.1$ fs) was

performed with energies as shown in figure 4.18. With a total energy drift of 0.03%, it is clear that the total energy is conserved very well. However, it must be noted that a low timestep size ($\frac{1}{10}$th of that of a typical standard MD simulation) was found to be necessary for a stable simulation.

ACE

This simulated system represents a typical use case since it is modeled using bonds, angles, dihedrals, LJ potentials and Coulomb interactions. Moreover, it uses LINCS constraints for its hydrogen atoms[14].

Following energy minimization, the simulated system was equilibrated to 100 K by means of an LD simulation (100,000 timesteps and $\Delta t = 1$ fs) and then to 1 bar using the Parrinello-Rahman barostat by means of a second LD simulation (100,000 timesteps and $\Delta t = 1$ fs). Then, by means of an LD simulation writing to the output trajectory in every timestep, the bead-starting configurations for a 480-bead long fictitious polymer were generated.

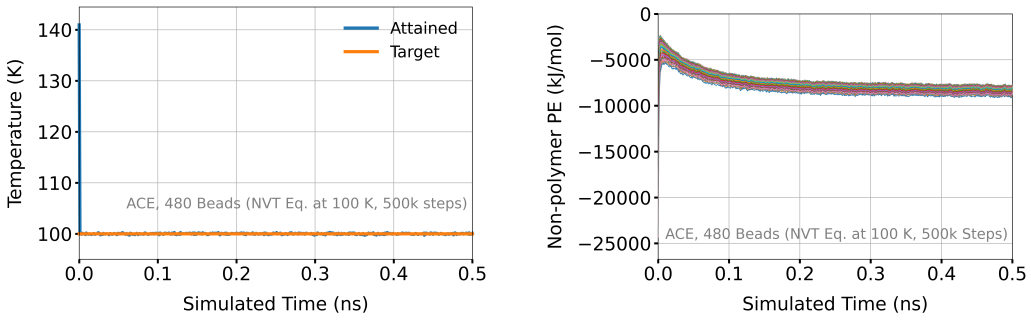


Figure 4.19: Left subfigure shows how the temperature is stably maintained during the LD pMD equilibration simulation of ACE. Right subfigure shows the non-polymer potential energies of all beads.

Next, an LD pMD simulation (500,000 timesteps with $\Delta t = 1$ fs) was performed to equilibrate the fictitious polymer to 100 K. As shown in figure 4.19, the non-polymer potential energies of all beads were clustered around a specific value and the target temperature was stably maintained. Moreover, the written trajectories were visually inspected to ascertain the absence of anomalies.

Following the equilibration, an NVT (LD) pMD simulation (2 million timesteps with $\Delta t = 1$ fs) was performed. The RMSD[15] values of the ace-

[14]Constraining atoms is a technique by which specific degrees of freedom of a molecule are fixed for computational efficiency, allowing thereby for simulations with larger timestep sizes.

[15]The Root Mean Square Deviation refers to the average deviation of the atoms of a (particularly structured) biomolecule from their corresponding images in the biomolecule's reference structure.

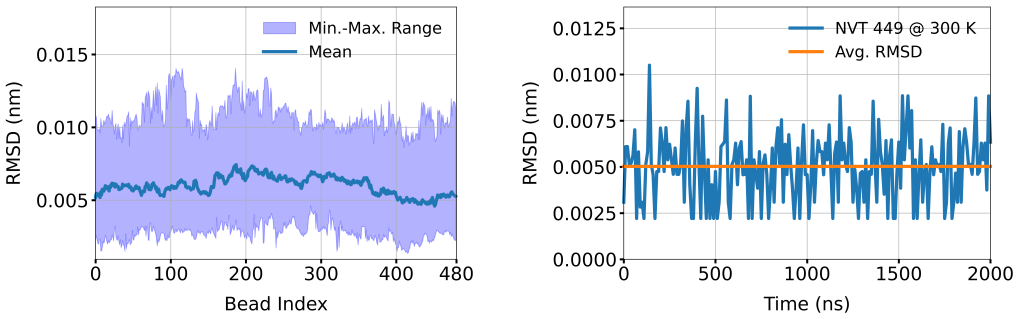


Figure 4.20: Left subfigure shows the mean, minimum and maximum RMSD values of all beads during an NVT (LD) pMD simulation of ACE. Right subfigure shows the instantaneous RMSD values for bead 450 of the fictitious polymer.

tone's "backbone" were computed for all beads, with results as shown in figure 4.20. Firstly, it is clear that the RMSD values have plateaued. Secondly, the maximum RMSD value is < 0.5 Å, which indicates minimal structural deviation i.e., the physical structure of the acetone molecule was preserved at all times during the simulation.

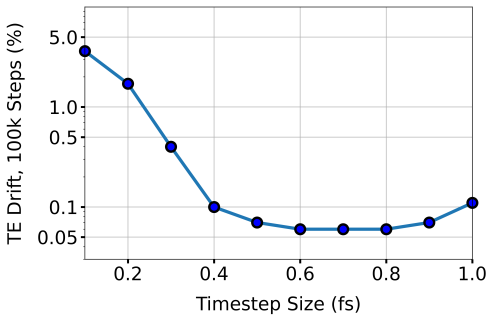


Figure 4.21: A plot showing the total energy drift of short NVE pMD simulations of ACE (with 480 beads and 100,000 timesteps) as a function of timestep Δt. As the timestep size is reduced, the total energy drift is seen to first fall to a minimum value before sharply rising to a high value. This can be taken to be indicative of increased numerical rounding errors at low timestep sizes.

Finally, an NVE pMD simulation (2 million timesteps with $\Delta t = 1$ fs) was performed, during which a rather high total energy drift of 3.4% was observed. Hoping to observe lower total energy drifts at lower timestep sizes, we performed short (100,000 timestep-long) NVE pMD simulations for various timestep sizes. The results (as shown in figure 4.21) are counter-

intuitive; as the timestep size decreased, the total energy drift dropped to a minimum value before sharply rising to a high value. One plausible explanation for this behaviour is increased numerical rounding errors at timestep sizes < 0.5 fs. Therefore, using our current implementation in single precision, the above total energy drift value is the lowest that can be expected.

Running the same simulation in double precision lowered the total energy drift to 0.2%. Moreover, in double precision, the total energy drift consistently decreases with the timestep size, which provides us an alternative for use cases in which a very low total energy drift is a necessity.

LYSOBENZ

This simulated system closely represents a practical use case of MD in the context of biological systems.

As per protocol, we began by performing the energy minimization of the simulated system. Next, it was equilibrated to 300 K by means of an LD simulation (100,000 timesteps and $\Delta t = 1$ fs) and then to 1 bar using the Parrinello-Rahman barostat by means of a second simulation (100,000 timesteps and $\Delta t = 1$ fs). Then, by means of an LD simulation writing to the output trajectory in every timestep, the bead-starting configurations for a 480-bead long fictitious polymer were generated.

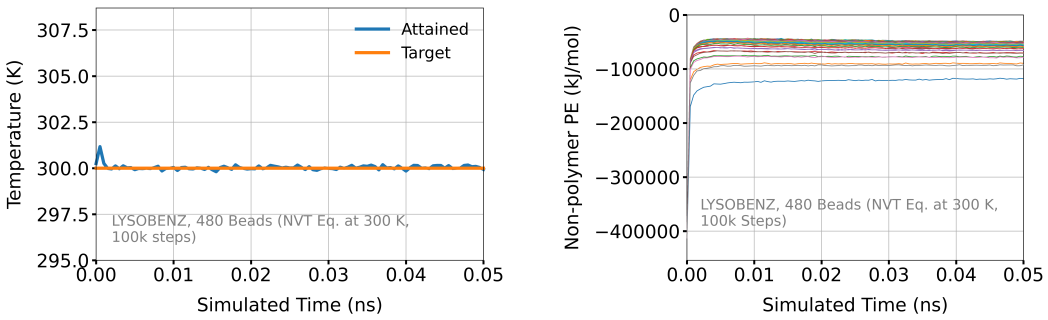


Figure 4.22: Left subfigure shows how the temperature is stably maintained during the LD pMD equilibration simulation of LYSOBENZ. Right subfigure shows the non-polymer potential energies of all beads.

Next, an LD pMD simulation (100,000 timesteps with $\Delta t = 0.5$ fs) was performed to equilibrate the fictitious polymer to 300 K. As shown in figure 4.22, the non-polymer potential energies of all beads were distributed within a certain range and the target temperature was stably maintained. Moreover, the written trajectories were visually inspected to ascertain the absence of anomalies.

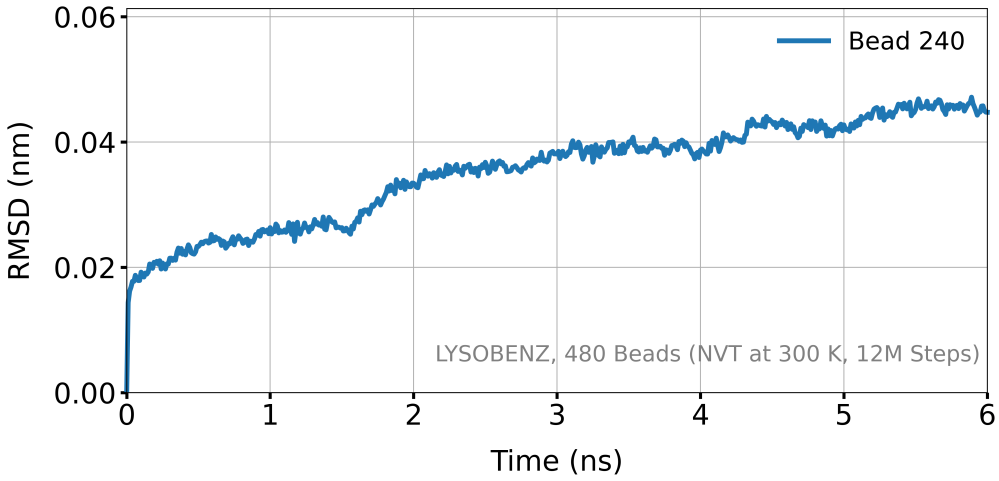


Figure 4.23: A plot showing the instantaneous RMSD values for bead 240 of the fictitious polymer. It can be seen that the values are slowly approaching a plateau and that the maximum value is < 1 Å, indicating that the physical structure of the protein is preserved at all times during the simulation.

Following the equilibration, an NVT (LD) pMD simulation (12 million timesteps with $\Delta t = 0.5$ fs) was performed using GPU force-offloading. The RMSD values of the protein's "backbone" were computed for all beads, with results as shown in figure 4.23. Although the values are slowly approaching a plateau towards the end of the simulation, it is clear that a longer simulation is necessary to observe the plateau. The maximum RMSD value is < 1 Å, which indicates that the physical structure of the protein was preserved at all times during the simulation.

Finally, an NVE pMD simulation (500,000 timesteps with $\Delta t = 0.5$ fs) was performed, during which a total energy drift of 3.4% was observed. Running the same simulation in double precision lowered the total energy drift to 0.05%.

4.3.3 Performance of pMD Kernels

The kernels 1-7 of the pMD algorithm are *memory-bound* functions i.e., their performance is limited solely by the memory bandwidth. Hence, their performance can be most appropriately quantified in relation to the peak read-write memory bandwidth. First, to estimate this roof for the JUWELS Cluster, we microbenchmarked the well-known DAXPY loop[16] using the utility Intel® MLC, obtaining a peak read-write memory bandwidth of 205 GB/s

[16]For n-dimensional arrays `x` and `y`, and scalar `a`, the DAXPY loop computes `y[i] = y[i] + a*x[i]` in double precision.

for the single JUWELS Cluster node.

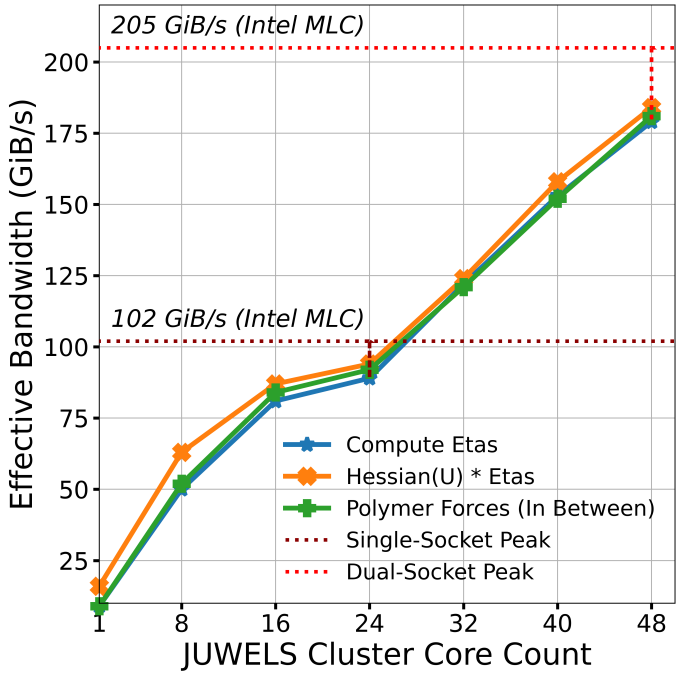


Figure 4.24: Strong-scaling performances of some of the pMD kernels within a single node of the JUWELS Cluster showing their peak effective bandwidths in relation to the peak read-write memory bandwidths attained by the utility Intel® MLC in the single- and dual-socket modes, which are represented by the dark red and red dotted lines, respectively.

Next, to test the performance of the newly implemented kernels, we wrote a microbenchmarking program for each kernel using the open-source microbenchmarking library TiXL [37]. In these programs, each kernel is called repeatedly with dummy input and output data and the wall-clock time of each call is measured using a monotonic clock. For each repetition, data is allocated afresh, the first word of each page is modified and each cache line is evicted from the caches using the intrinsic `_mm_clflush`. This is done to prevent timing page faults and simultaneously avoid having data in the caches at the time of the function call. The first 20 of 100 repetitions are not timed, allowing the program to adequately warm up.

Using the above programs, we conducted strong-scaling tests of all kernels on a single node of the JUWELS Cluster for a fixed, large array size of 1 GiB. The results of the test showed that the kernels scale very well with core count, as demonstrated for example kernels `computeEtas`, `computeHessianUTimesEtas` and `computePolymerForcesInBetween` in figure 4.24. Here, we see that e.g., the kernel `computeEtas` attained a peak effective bandwidth of 179 GB/s,

which yields an efficiency of 87% in relation to the roof. This way, the efficiencies of all kernels were found to be $87\% - 90\%$, on the basis of which we concluded that the kernels are well-optimized.

4.3.4 Benchmarking pMD without DD

At first, it may seem that modeling the performance of the pMD algorithm to gauge the performance of our implementation is complicated by having to estimate the performance of the internal GROMACS function `do_force`. However, since improving the performance of this function is *not* one of the goals of our work, the performance of our pMD implementation can simply be expressed in terms of it.

We began by running standard MD simulations using the LYSOBENZ system for atom counts of $\sim 50k$, $\sim 100k$, $\sim 150k$ and $\sim 200k$ on a single node of the JUWELS Booster. These simulations were conducted using one MPI rank with access to 12 OpenMP threads distributed across 2 NUMA domains with SMT disabled. The profiling tables in the simulation log files consistently showed that `do_force` accounts for 97% of the total runtime of the standard MD simulation.

Next, we profiled pMD simulations with the help of TiXL's MPI Spot Profiler using the same simulated system, with identical input options, starting configurations[17] and atom counts as above. These simulations were conducted with three beads, one MPI rank/bead, each MPI rank in a separate JUWELS Booster node and the same OpenMP thread count and affinity plan as above. The results revealed that `do_force` accounts for 96% of the total runtime of the pMD simulation.

Now, in standard MD, `do_force` is called once per time step. On the other hand (as detailed in subsection 4.2.3), in our pMD implementation, it is called three separate times per time step. Taking into account the fact that the function makes up the vast majority ($> 95\%$) of the runtimes of both standard MD and pMD simulations, a simple performance model of this pMD algorithm is given by

$$\begin{aligned} t_S &\approx n_{\text{steps}}\, t_{\text{do\_force}}, \\ t_P &\approx n_{\text{steps}}\, 3t_{\text{do\_force}}, \\ \therefore t_P &\approx 3t_S, \end{aligned}$$

where n_{steps} is the number of simulated time steps, $t_{\text{do\_force}}$ the time of `do_force`, t_P the runtime of a pMD simulation and t_S the runtime of its analogous standard MD simulation. This model provides us with the per-

[17] The same starting configuration was used for all beads.

formance indicator

$$\frac{t_P}{t_S} \approx 3, \tag{4.15}$$

according to which the performance of an implementation of the pMD algorithm described in subsection 4.2.3 is said to be optimal if its runtime is *at most thrice* that of its analogous standard MD simulation.

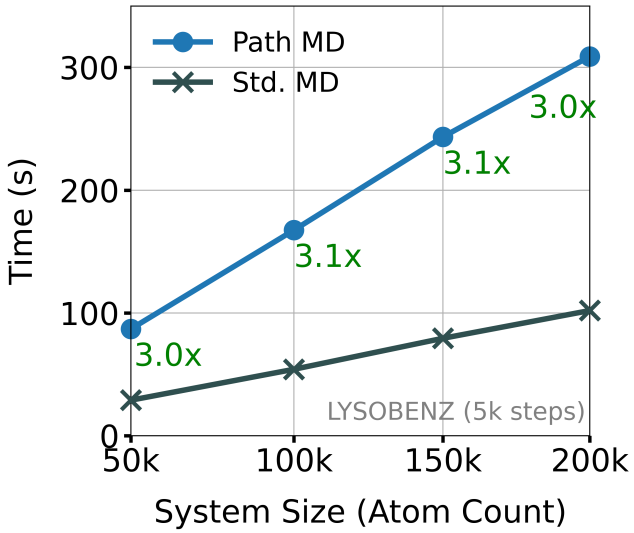


Figure 4.25: A plot relating the runtimes of pMD simulations of the system LYSOBENZ with various atom counts to the runtimes of their analogous standard MD simulations on the JUWELS Booster. The factor-of-three in runtime, which can be seen consistently for all atom counts, is a first indication of good performance.

To analyze the performance of our pMD implementation on the basis of the above indicator, we benchmarked standard MD and pMD simulations using the same simulated system and atom counts as in the profiling runs on the JUWELS Booster. Also, the inputs, simulation parameters and setup (including the OpenMP thread count and affinity plan) remained unchanged. Using separate PME ranks was disabled by passing `-npme 0` to `mdrun`. The simulations were run for 30,000 time steps, with timers and performance counters being reset after 25,000 time steps to avoid measuring the auto-tuning and optimization phase that takes place at the start of GROMACS simulations. Comparing the standard MD runtime to the *pMD runtime**, we observed the *factor-of-three* as shown in the plot of figure 4.25.

DEFINITION : PMD RUNTIME

The maximum runtime of all replicas of the multi-simulation that is the pMD simulation. The runtime of each replica is obtained from its GROMACS log file.

The close alignment of the experimental results with the prediction of the performance indicator from equation (4.15) should not be taken as a definitive indicator of optimal or even near-optimal performance. The low node count was chosen specifically to reduce MPI-parallel effects (e.g., communication time and synchronization overhead) that are notoriously difficult to accurately model. It is possible that a sub-optimally performing pMD implementation could perform similarly at this node count but significantly worse at higher node counts. Therefore, for a complete performance analysis, one must factor in the performance at various node counts, which we've done via our strong-scaling test reported in subsection 4.3.6 and our weak-scaling test reported in subsection 4.3.7. Nevertheless, the results of the above experiments can be understood to be the first indication of good performance.

4.3.5 Benchmarking pMD with DD

To analyze the performances of the centralized and the decentralized twin-finding algorithms and to compare them with one another, we performed strong-scaling tests of pMD simulations using these algorithms for the system LYSOBENZ (with $\sim 100k$ atoms) on the JUWELS Booster. To reproduce these tests for a different simulated system, we repeated them for A2A-ZMA, the results of which are presented separately (for clarity) in appendix section B.1.

Strong Scaling Within Single Node

In these strong-scaling tests, since we were interested in analyzing the performance of DD, it was necessary to minimize the noise introduced by the bead-level parallelism. Consequently, the size of the fictitious polymer was limited to three beads, with each bead being allocated a separate JUWELS Booster node. The core count per bead was increased successively in terms of the number of MPI tasks dedicated to DD. To eliminate the performance effects of OpenMP-based parallelism, each MPI task was assigned a single thread pinned to a single core. Since the two twin-finding algorithms pertain only to PP-based DD, allocation of separate PME ranks was disabled. As in the performance tests of the previous subsection, each simulation was

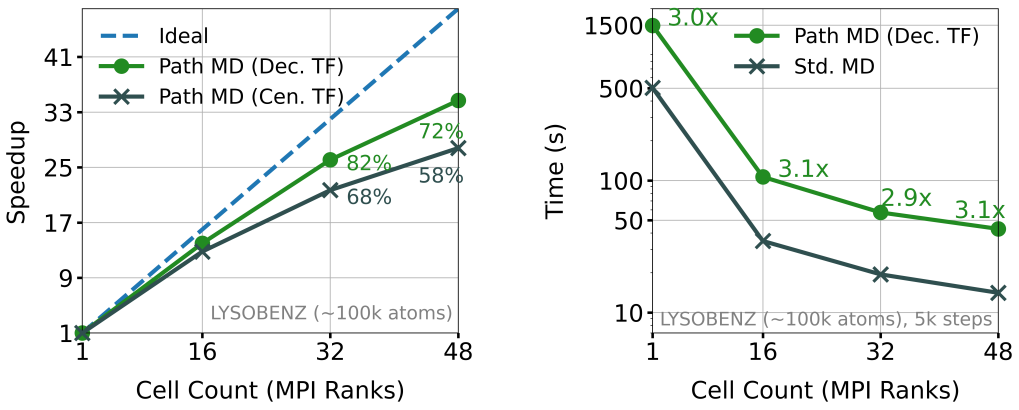


Figure 4.26: Left subfigure shows the strong-scaling performances of pMD simulations using DD via the centralized and the decentralized twin-finding algorithms. Right subfigure shows the runtime factor-of-three of pMD simulations using DD via the decentralized twin-finding algorithm with respect to their analogous standard MD simulations. In both subfigures, the simulations are of the system LYSOBENZ (with $\sim 100k$ atoms).

run for 30,000 time steps and the clocks and performance counters were reset after 25,000 time steps. Here, FDM stabilization was disabled[18].

The results are presented in figure 4.26. As originally hypothesized, we could confirm that the decentralized twin-finding algorithm performs significantly better than the centralized one, making it the algorithm of choice for DD support. Comparing the strong-scaling performance of our pMD implementation using DD via the decentralized twin-finding algorithm to that of standard MD (as shown for the analogous runs depicted in figure 4.4), it can be seen that they scale very similarly. Further, as seen in the right subfigure of figure 4.26, even when using DD (via the decentralized twin-finding algorithm), the desirable runtime factor-of-three could be observed.

Strong Scaling Beyond Single Node

To analyze the performance of our pMD implementation when transcending the single node, we also conducted a strong-scaling test using DD via the decentralized twin-finding algorithm with three beads and going up to 20 JUWELS Booster nodes per bead. Here, the allocation of separate PME ranks was enabled. The results of this test showed that good speedups could be achieved, with the simulation ultimately running 10 times faster, albeit at

[18]For completeness, FDM stabilization was enabled with maximum frequency for the tests reported in appendix section B.1.

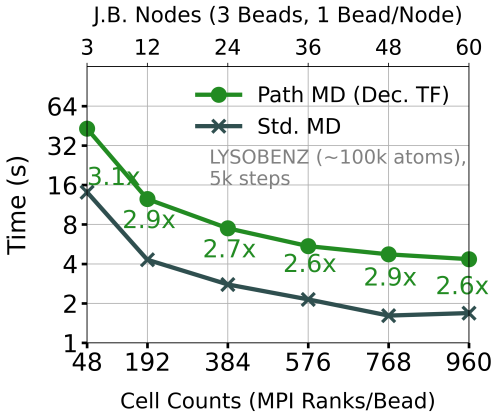


Figure 4.27: A plot demonstrating the runtime factor-of-three for pMD simulations of the system LYSOBENZ (with $\sim 100k$ atoms) using DD via the decentralized twin-finding algorithm with three beads, using up to 20 JUWELS Booster nodes per bead and with separate PME ranks enabled. Good speedups were obtained, albeit at progressively degrading strong-scaling efficiencies.

a low strong-scaling efficiency of 37%. Also here, the factor-of-three could be observed, as presented in figure 4.27.

Energy Consumption

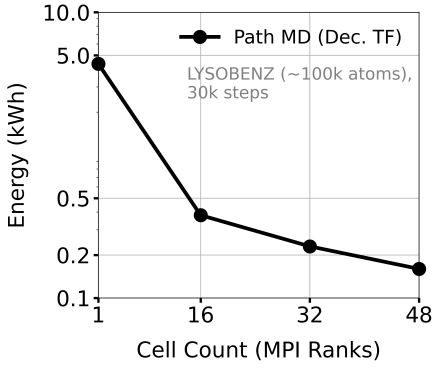


Figure 4.28: A plot relating the energy consumption of pMD simulations of the system LYSOBENZ (with $\sim 100k$ atoms) with three beads for 30,000 time steps to the number of MPI ranks given to DD (via the decentralized twin-finding algorithm). The energy consumption dropped by a factor of 27 from when DD was disabled to when it was enabled with as many MPI ranks as cores in one JUWELS Booster node.

Another interesting aspect of running pMD simulations using DD is the

sharp reduction in energy consumption. Using the monitoring infrastructure LLview developed by the Jülich Supercomputing Center [117], we were able to report the energy consumption of the strong-scaling test simulations. As shown in figure 4.28, the energy required to run a pMD simulation for 30,000 time steps reduced strongly and progressively as the number of MPI ranks dedicated to DD increased, with a total reduction by a factor of 27. Extrapolating from this information, running a pMD simulation for 10 million time steps would need ~ 1.453 MWh without DD and only ~ 53 kWh with DD at 48 MPI ranks per bead.

To conclude, the above results have made it clear that the decentralized twin-finding algorithm is preferable to the centralized twin-finding algorithm. Further, our pMD implementation showed excellent strong-scaling performance when using DD via the decentralized twin-finding algorithm. Moreover, the factor-of-three observed in all tests indicates that the performance of our pMD implementation is good at low node counts.

4.3.6 Benchmarking with GPU Offloading

As detailed in subsection 4.2.9, enabling GPU offloading in pMD simulations is achieved by providing the `mdrun` program any combination of the command-line options, `-bonded`, `-nb`, `-pme` and `-update`, followed each time by `gpu`. For this to work, a GPU-enabled GROMACS build is required, for which we built GROMACS with CUDA support. Since each combination can potentially have different performance implications, we performed pMD simulations of the A2A-ZMA system (with $\sim 150k$ atoms), once for each combination, on one node of the JUWELS Booster with access to four GPUs. The results of these runs are provided in table 4.1.

gpu	-nb	-bonded	-pme	-update
-nb		53.734	33.684	*Error1*
-bonded			*Error2*	*Error3*
-pme				*Error2*
-update	(*Symmetric Entries*)			

-nb -bonded -pme -update	*Error1*
-nb -bonded -pme	32.364

Error1	*Only the md integrator is supported.*
Error2	*Nonbonded interactions must also run on GPUs.*
Error3	*Short-ranged, non-bonded interactions must also run on GPUs.*

Table 4.1: Runtime (in seconds) with different GPU-offloading combinations. However, as can be seen, some combinations resulted in errors.

These simulations were performed for a fictitious polymer with four beads, with one bead/GPU. As in the tests conducted in subsections 4.3.4 and 4.3.5,

they were run for 30,000 time steps of which only the last 5,000 time steps were timed. Importantly, the *stochastic dynamics* (SD) integrator was used, which, we had established during the testing-for-correctness phase, can be used to sample the distribution of equation (4.2). However, this resulted in *Error1* for the combinations with `-update`, which tells us that this option is only supported for the default leap-frog integrator. As table 4.1 makes it clear, not all combinations are feasible.

Of the three viable combinations, It can be seen that `-nb gpu -bonded gpu -pme gpu` is the fastest[19]. We note that it is much faster than pMD using DD. Using DD, a pMD simulation of three beads across three nodes (with one bead/node) takes ~ 68 s, whereas using force-offloading to GPUs, the same pMD simulation but with four beads and a single node takes ~ 32 s. Neglecting scaling effects for the small difference in node counts, in the time it takes to simulate one bead in the former mode, *8 beads* can be simulated in the latter mode.

Increasing GPU Utilization and Throughput

DEFINITION : OVERLAY FACTOR

The number of beads processed by a single GPU.

Using the information from LLview, it was observed that, despite the excellent performance, the average GPU utilization was quite low, at 30%. Since there is information suggesting that running multiple simulations per GPU in parallel can increase overall GPU utilization, we decided to investigate it [103]. To this end, we conducted the "overlay" test —a series of pMD simulations of the system A2A-ZMA using the four GPUs of a single JUWELS Booster node in which we studied the effects of overlaying increasing numbers of beads on each GPU. Specifically, we were interested in seeing the relationship between the *overlay factor** and average GPU utilization and *throughput**. The test simulations were run after starting the CUDA Multi-Process Service (MPS) for better hardware utilization [103].

DEFINITION : PMD THROUGHPUT

The size of the fictitious polymer (viz., the number of beads) that can be simulated in a given time.

[19] From now on, any mention of force-offloading to GPUs must be taken to mean this combination.

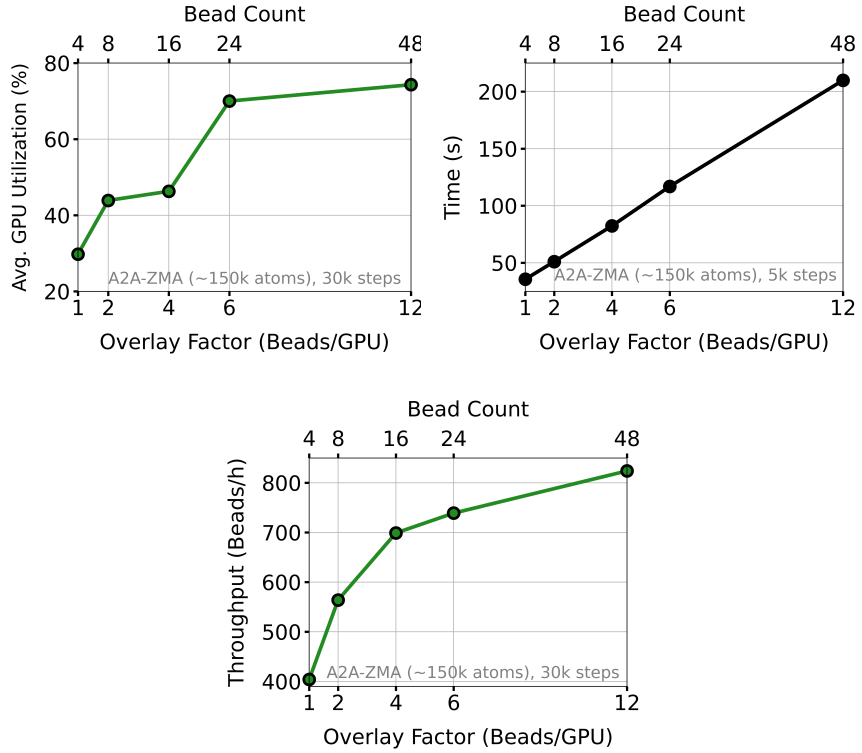


Figure 4.29: Subfigures showing the results of the overlay test conducted using the four GPUs of one JUWELS Booster node by running pMD simulations of the system A2A-ZMA (with $\sim 150k$ atoms) with four beads for 30,000 time steps. Top left subfigure shows how the average GPU utilization increases with the overlay factor, from 30% at 1 bead/GPU to 74% at 12 beads/GPU. Top right subfigure shows the runtime increasing with the overlay factor. Bottom subfigure shows the increase in throughput with increasing overlay factor in terms of number of beads simulated per hour.

The results of this test are visualized by the plots in figure 4.29. As expected, the average GPU utilization of the simulations increased with the overlay factor, from 30% at 1 bead/GPU to 74% at 12 beads/GPU. On the other hand, the runtime of the simulations also increased by a total factor of 6 in the same range. Nevertheless, as a result of better average GPU utilization, the throughput also increased sharply.

This information puts the user at the crossroads of throughput and speed; overlaying helps increase the GPU utilization (and hence throughput) significantly but also simultaneously increases the runtime noticeably. In practice, however, the resolution to this dilemma comes easily. Firstly, overlaying becomes unavoidable when the simulation requirement approaches the hardware-resource limit (viz., the number of GPUs available) and the

fictitious polymer has too many beads to be accommodated. Secondly, increasing the throughput is a good way of saving computing time, especially when it is a limited resource. For example, in the overlay test, we noticed that overlaying 12 beads/GPU effectively *halved* the core-hour usage as compared to when not overlaying.

Strong Scaling Across JUWELS Booster

To investigate the strong-scaling performance of our pMD implementation when using force-offloading to GPUs, we conducted a series of strong-scaling tests by running pMD simulations of the system A2A-ZMA in the size scales of small (with 384-2,400 beads), medium (with 5,760 beads) and large (with 9,600 beads) involving 8 to 800 JUWELS Booster nodes. The results of these tests are provided in the subfigures of figure 4.30.

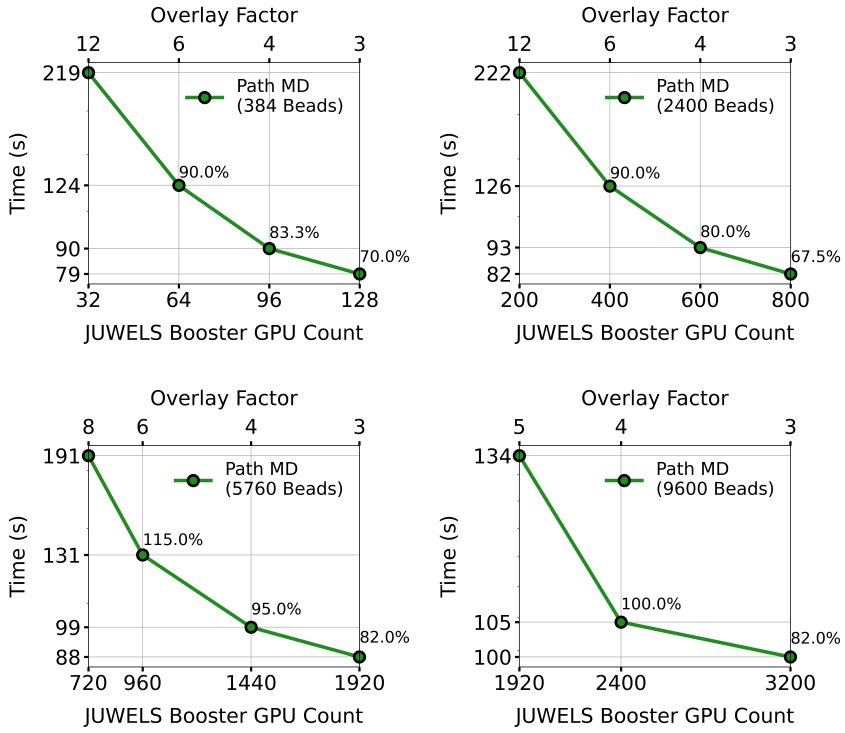


Figure 4.30: Subfigures showing the results of the strong-scaling tests of pMD simulations of the system A2A-ZMA (with $\sim 150k$ atoms) using the JUWELS Booster's GPUs. The top two subfigures represent the small scale, with 8 to 200 JUWELS Booster nodes. The bottom-left subfigure represents the medium scale, with 180 to 480 JUWELS Booster nodes. The bottom-right subfigure represents the large scale, with simulations using 480, 600 and 800 JUWELS Booster nodes.

To conclude, even without making use of the GPU-resident mode, we report excellent speedups of pMD simulations when using force-offloading to GPUs compared to when using DD. Following up on a valuable hint provided by previous research, we identified a way to increase the average GPU utilization and hence improve throughput on GPUs. Finally, our strong-scaling tests reveal the excellent scaling performance of the combination of our pMD implementation and GPU offloading, spanning multiple scales and large node counts.

4.3.7 Extreme Weak Scaling to 94% of JUWELS Booster

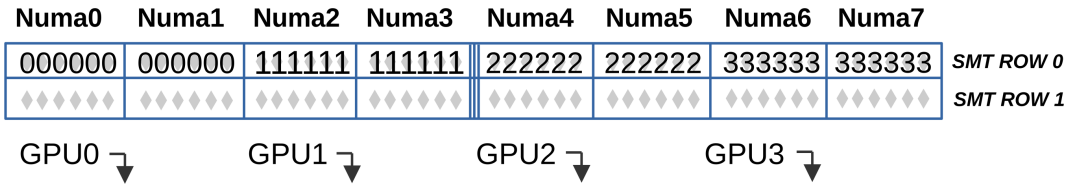


Figure 4.31: Illustration of the affinity policy used for the weak-scaling tests on the JUWELS Booster.

Perhaps the most remarkable characteristic of the pMD algorithm is that it converts the time-scale problem of standard MD i.e., "How long can we simulate?", into a size-scale problem i.e., "How many beads can we simulate?", by virtue of its theoretical foundation. This has a profound performance implication —the inherently serial nature of the time evolution in standard MD is exchanged for the inherently parallel approach of the pMD algorithm. The beads of the pMD algorithm's fictitious polymer represent work that can be very efficiently parallelized; by allocating hardware to every new bead, the fictitious polymer can be extended indefinitely. In theory, this makes it the perfect weak-scaling application.

To determine the veracity of the above statement in actual practice, we performed a large-scale weak-scaling test of our pMD implementation on the JUWELS Booster. In this test, pMD simulations of the system A2A-ZMA (with $\sim 150k$ atoms) were performed with force-offloading to GPUs without bead overlaying, in which the node count —and hence, automatically, the bead count —was progressively increased. In each node, each bead was allocated one GPU (for the force computations) and 12 closely placed threads (for the update) with SMT disabled and exactly one direct connection to the GPU. The affinity policy is as shown in figure 4.31. The results of the weak-scaling test are shown in figure 4.32. Going up to 875 JUWELS Booster nodes and making use of 3,500 GPUs and 42,000 CPUs, we observed at least 72% weak-scaling parallel efficiency, which (at this scale) is outstanding weak-scaling performance. Moreover, a plateauing of the efficiency can be observed at about 50% of the JUWELS Booster.

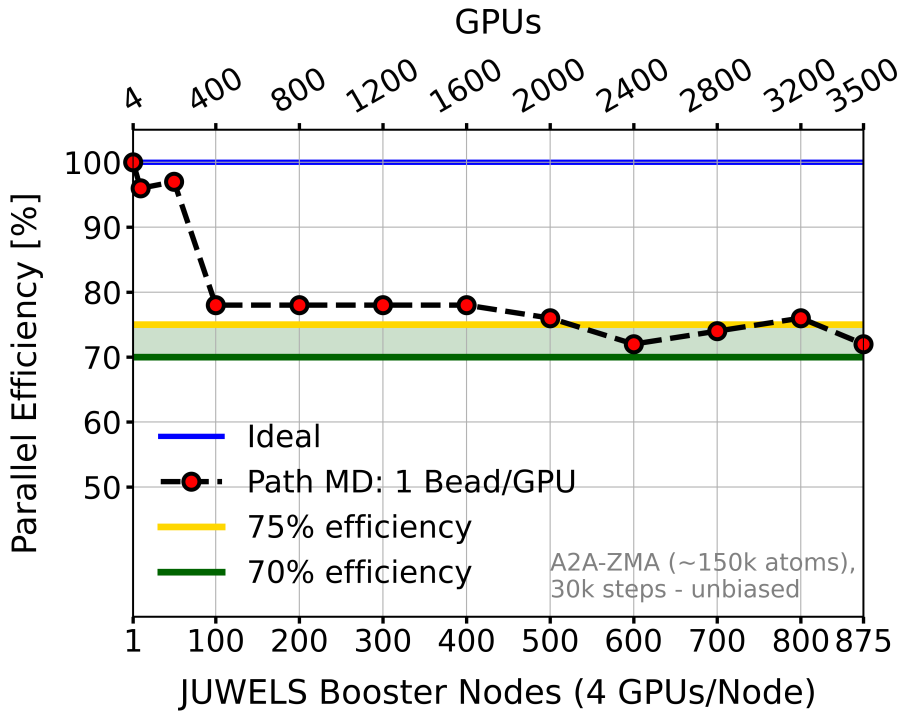


Figure 4.32: A plot showing the excellent weak-scaling performance of our pMD implementation on the JUWELS Booster, with force-offloading to GPUs.

As mentioned in subsection 4.2.1, our primary motivation to implement our pMD algorithm was to be able to carry out MoP simulations. Therefore, to look into the weak-scaling performance of metadynamics simulations performed using our pMD implementation coupled with the PLUMED library, we conducted a weak-scaling test identical to the above test with the only difference being the use of a PLUMED input file that defines a polymer end-to-end CV. This file is provided in the appendix section B.2. Of course, to correctly interpret and execute the instructions contained in this input file, this test had to be conducted using a GROMACS build that was modified using our PLUMED patch, linking with our modified version of PLUMED v2.9.0 that integrates the changes mentioned in subsection 4.2.10. Here, the results are quite similar to the tests run without PLUMED. They show the excellent weak-scaling performance of the combined software, with at least 70% weak-scaling parallel efficiency going up to 875 JUWELS Booster nodes and making use of 3,500 GPUs and 42,000 CPUs as shown in figure 4.33.

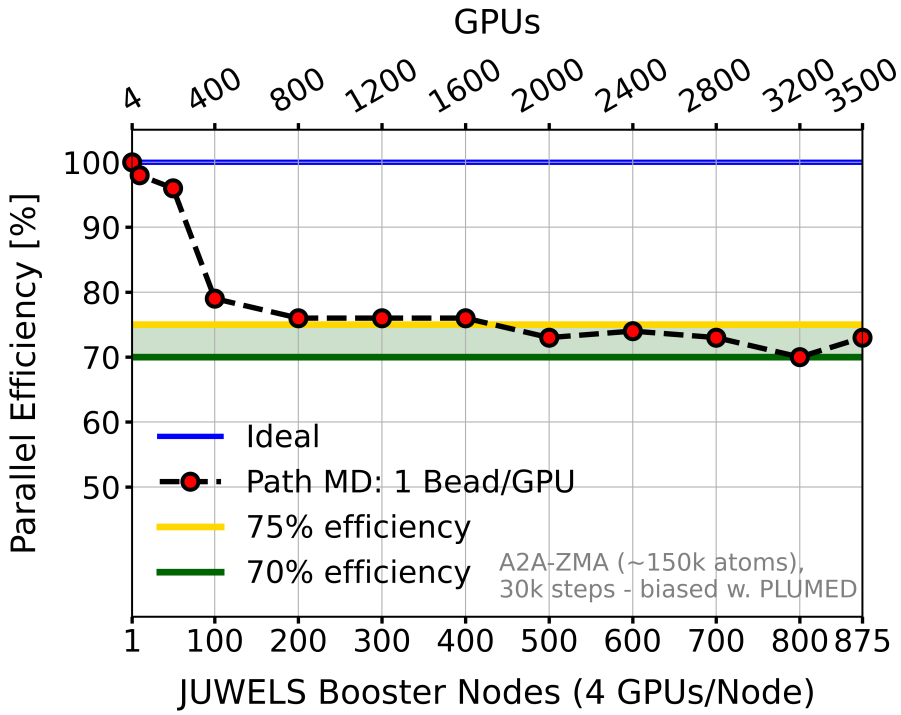


Figure 4.33: A plot showing the excellent weak-scaling performance of our pMD implementation on the JUWELS Booster, patched using our PLUMED patch and linking with our modified version of PLUMED v2.9.0, with force-offloading to GPUs.

4.4 Conclusion

In this work, we've designed a pMD algorithm to perform trajectory-space simulations and implemented it in GROMACS v2024. We've optimized its communication to reduce waiting time for all communicators, and its computational kernels for OpenMP-based auto-vectorization and shared-memory parallelism. During initial testing, we discovered the need for stabilization of an internal FDM computation and subsequently implemented an error-reduction strategy that reduces the total energy drift considerably. Moreover, we implemented a scheme to mitigate the error caused by COM motion, to increase the overall accuracy of the simulation. After the implementation was completed, we ran correctness tests using several systems that highlighted the strengths and limitations of our pMD implementation. By benchmarking our pMD implementation on the JUWELS Booster and with the help of the factor-of-three performance modeling, we demonstrated that it is efficient at low node counts.

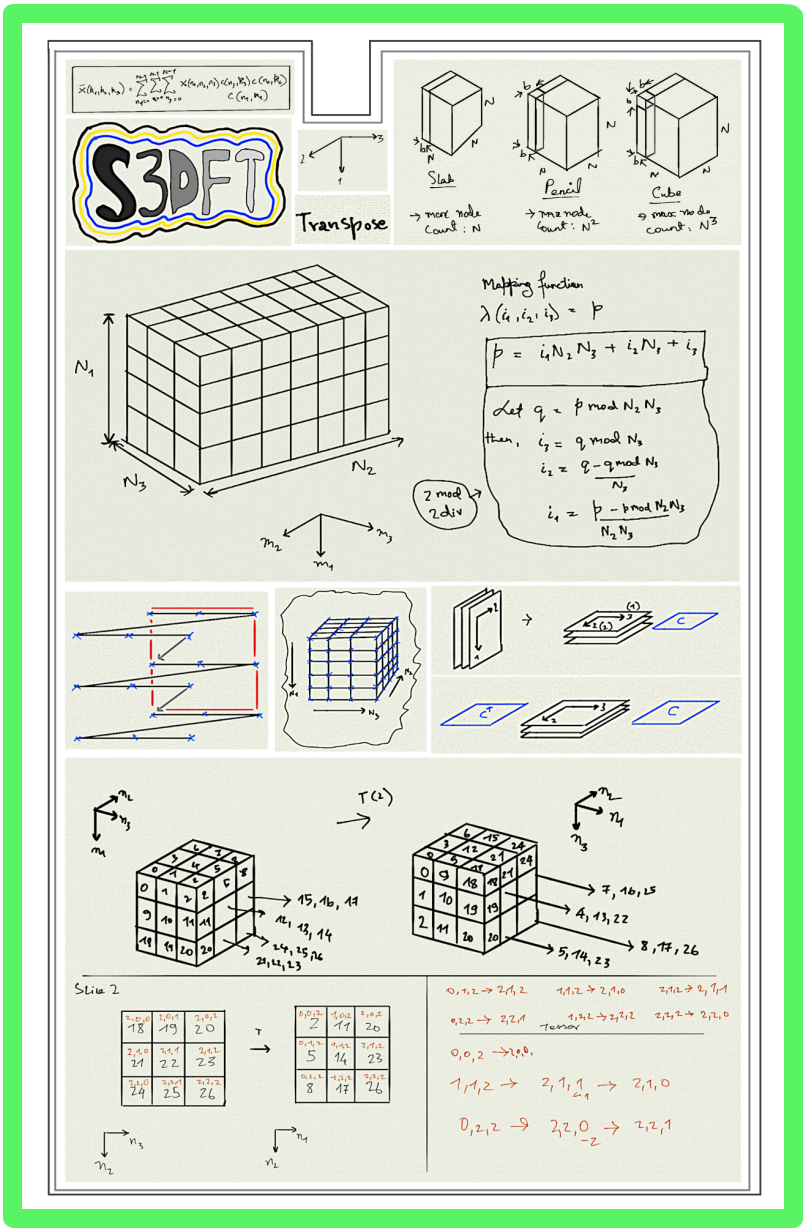
Since the pMD algorithm was conceived to ultimately serve as a tool to ap-

ply MoP, we've adapted our pMD implementation to make it compatible with the metadynamics engine available in the PLUMED library. Similarly, we've adapted the PLUMED patch for GROMACS to allow our pMD implementation to make the necessary PLUMED function calls. Moreover, we've adapted the PLUMED interface to signal, parameterize and control MoP simulations by means of the PLUMED `Custom` function, and implemented the necessary changes in a yet unreleased copy of the PLUMED library v2.9.0. Further, we've tested the adapted interface for correctness and conducted preliminary testing of the complete MoP machinery.

To be able to leverage the performance advantage offered by GROMACS' DD, we invented two algorithms to provide our pMD implementation with the extensions necessary for this mode of parallelism. Further, we implemented both algorithms and optimized them for high asynchronicity of communication. By benchmarking simulations with either algorithm on the JUWELS Booster, we observed that the decentralized twin-finding algorithm provides the best performance. In fact, it was shown that this algorithm meets the requirement of the factor-of-three model at low node counts. Moreover, using it, our pMD implementation showed very similar strong-scaling performance to standard MD, both within and beyond the single node. Applying DD across multiple nodes via the decentralized twin-finding algorithm resulted in significant performance improvements, albeit at low strong-scaling parallel efficiencies.

Finally, we've shown how our pMD implementation can make use of GROMACS' native force-offloading to GPUs and that it can be much faster than single-node DD on the JUWELS Booster. We've analyzed the performance of force-offloading to GPUs for hardware resource utilization, highlighted how simulation throughput can be increased and conducted extensive strong-scaling tests at various node counts to reveal the excellent strong-scaling parallel efficiency of our implementation in this mode. Most importantly, we've shown its excellent weak-scaling performance in large-scale tests going up to 875 nodes of the JUWELS Booster with at least 72% parallel efficiency even at large node counts. Furthermore, we conducted the same tests with PLUMED to demonstrate similar performance with a minimum 70% weak-scaling parallel efficiency.

This work has paved the road to studying complex biophysical processes by enhanced sampling via MoP. The resulting code has been shown to be high-performing and efficient, and its efficacy can now be determined by applying it to study real scientific systems.



For those who believe that things are best understood with a paper and pen.

Chapter 5

Scalability of 3D DFT by Block-Tensor-Matrix Multiplication on the JUWELS Cluster

This chapter is based on publication (A.) "List of Publications" (p. 4). Verbatim quotes of this publication are indicated as "..." (A.). They may extend over several pages and have been reformatted to fit in this monograph.

5.1 Introduction and Motivation

"Many fields of numerical simulation such as astrophysics, plasma physics and molecular dynamics (MD) involve computing the pair-wise long-range interactions between the physical system's constituents [118–120]. Some examples are gravitational forces, Van der Waals and electrostatic interactions. This computation is time-consuming and often restricts sizes and time scales. For example, in atomistic MD simulations of bio-physical systems, the sizes to be simulated can be very large —ranging up to 10^9 [38] particles —and computing the long-range interactions is typically responsible for 90 % of the total run-time. This problem is particularly relevant in the field of quantum mechanics-based MD simulations [73]. To limit the computational costs and improve scaling, techniques derived from the Ewald summation method are extensively used, which utilize the three dimensional Discrete Fourier Transform (3D DFT) —both to ensure the convergence of the calculation and to gain speed-up [61, 65, 121]. Therefore, improving the scalability of the parallel 3D DFT is very desirable as it will extend the scope of these simulations —enabling the study of larger and more complex systems

and for longer times —exploiting modern, massively parallel computer architectures [122].

The DFT operation is typically applied using any of the set of algorithms known collectively under the name of Fast Fourier Transform (FFT) [123]. Specifically, variants of the Cooley-Tukey FFT algorithm are the most commonly employed. They break the original DFT problem down into a tree of smaller DFT problems, which are sometimes solved recursively and more often non-recursively [124]. This results in a drastic reduction of arithmetic complexity from $O(N^2)$ of the naïve algorithm, down to $O(Nlog_2N)$. Similarly, the 3D FFT operation reduces the arithmetic complexity from $O(N^4)$ to $O(N^3log_2N)$. Depending on the size of the problem, this may considerably reduce the run-time of computer applications. However, in the context of distributed-memory computers, the run-time of the 3D FFT is dominated by communication, which can make up to 80-95% of it [125, 126], and so there is still interest in improving the scaling performance of the 3D FFT algorithm [125–128].

Reports suggest that the high fraction of communication time of the 3D FFT is due to its unavoidable use of all-to-all communications [125, 129]. Further, on the Fugaku supercomputer, it has been shown that 3D FFT algorithms that make use of point-to-point communication scale better than those that make use of all-to-all communication [126]. This indicates that one may profit from 3D DFT algorithms that achieve better scalability by means of alternative communication patterns.

In order to achieve better scalability by swapping all-to-all for point-to-point communication, in 2015, Sedukhin *et al*. studied the scalability of an alternative algorithm that makes use of point-to-point communication to compute the 3D DFT, albeit at the significantly higher arithmetic complexity of $O(N^4)$ as compared to that of $O(N^3 \log N)$ of the 3D FFT algorithm. Their work implemented a point-to-point orbital algorithm that targeted the IBM Blue Gene/Q computer, which is based on a 5D torus topology. Their benchmarking revealed worse overall performance of their implementation compared to that of the standard 3D FFT implementations of the time. However, the authors noted that, for a single node, their implementation of the core computational operation of the algorithm —the tensor-matrix multiplication —achieved 20% of the corresponding peak performance, and concluded with the speculation that a more efficient implementation could outperform the 3D FFT algorithm for very large node counts [130]. Since then, the computational power of CPUs has grown more than the speed of the interconnects between the nodes [131], a fact which leans in favour of this alternative approach.

Inspired by the work of Ref. [130], we have designed and implemented a

new 3D DFT algorithm that makes use of point-to-point communication, and benchmarked it on the JUWELS Cluster [32] at the Jülich Supercomputing Center. This cluster is based on the fat tree network topology, which is one of the most commonly adopted network topologies nowadays. Our algorithm —which we refer to as 3D DFT by block tensor-matrix multiplication —is based on specially adapted variants of Cannon's algorithm, which, in its original form, is an efficient distributed-memory matrix-matrix multiplication algorithm suited for square matrices [132–134]. We chose this algorithm because of its scalability [133] and the simplicity of its implementation. Our adaptations not only make it possible to use tensor operands, but also enable the utilization of the well-known strategy of overlapping communication and computation —by dedicating a thread to communication with the help of a custom work-sharing function for OpenMP-based multithreading —in an effort to hide the latency of communication. Further, by carefully applying only static scheduling, which almost entirely eliminates the use of processor interconnects in a non-uniform memory access (NUMA) setup, our implementation enables the efficient use of multithreading across NUMA domains. This avoids the use of distributed-memory parallelism within a shared-memory space, thereby increasing the efficiency of hardware resource utilization on the single node level.

We have implemented our algorithm as a C++ library that we named S3DFT and compared it with two competitive FFT implementations, namely, the FFTW3 library [124] and the Intel Math Kernel Library (MKL). This choice is motivated by the fact that these libraries are well-established, high-performing and arguably the most used worldwide in various applications[1]. The results demonstrate strong scaling of S3DFT up to the maximum number of $16,464$ cores considered. However, we found its overall performance to fall behind those of its competitors. A detailed analysis uncovered the source of the problem in two of its distributed-memory components that scale poorly owing to how their intrinsic communication patterns are mapped in subsets of the fat tree topology. Algorithmic variants of these components designed to overcome this problem significantly improved the strong-scaling performance of S3DFT, but did not lead to overall increase of its performance due to the overhead of the additional global data transpositions required.

Summarizing the main contributions of our work:

- We have designed and implemented a 3D DFT algorithm that uses only point-to-point communication.

[1]The Intel Math Kernel Library is the fastest and most-used math library for Intel-based systems (data from Evans Data Software Developer survey, 2022). The reference publication discussing the implementation of FFTW3 [124] has more than 3,200 citations in papers and more than 40 citations in patents.

- In this endeavour, we have designed, implemented and analyzed the performance of variants of Cannon's algorithm with adaptations that extend its use for block tensor-matrix multiplications on volumetrically decomposed domains.
- We have exposed a potential drawback when running block-wise parallel algorithms on machines with a fat tree network.
- We have compared the strong scaling performance of S3DFT with that of FFTW3 and Intel MKL on the pre-exascale JUWELS Cluster at the Jülich Supercomputing Center. This represents useful information for researchers who routinely use these popular 3D FFT libraries on similar machines.

The paper is organized as follows. Section 5.2 introduces the notation used throughout the paper, reviews key concepts needed to describe our 3D DFT algorithm, and describes the modeling of the overlapping of the communication and the computation used to select the combination of block size and node count in the performance analysis of the distributed-memory block tensor-matrix multiplication algorithm. Section 5.3 outlines the details of the actual implementation of S3DFT. Section 5.4 provides the specifications of the JUWELS Cluster that was used for the benchmarking. Section 5.5 presents a performance analysis of the core functions and provides insights into the overall performance of the algorithm. Finally, sections 5.6 and 5.7 discuss the results of comparisons of S3DFT with FFTW3 and Intel MKL and put forward our conclusions" (A.).

5.2 Theory

5.2.1 3D DFT by Block Tensor-Matrix Multiplication

The DFT for a sequence of 3D operand data $x(l,m,n) \in \mathbb{C}$ can be written as [130][2]

$$y(i,j,k) = \sum_{n=0}^{N-1}\sum_{m=0}^{N-1}\sum_{l=0}^{N-1} x(l,m,n)\; c(l,i)\; c(m,j)\; c(n,k) \tag{5.1}$$

where the coefficients

$$c(n_1,n_2) = \exp(-i\,\frac{2\pi}{N} n_1 n_2) \quad \forall\, 0 \le n_1, n_2 < N$$

define the DFT matrix $C \in \mathbb{C}^{N\times N}$. Making use of the formalism of Kolda *et al.* [135], the operand data $x(l,m,n)$ can be viewed as an order-3 tensor[3]

[2] In this work, we only consider the case in which the operand data $x(l,m,n)$ can be arranged as a cube, i.e., $0 \le l,m,n < N$. To extend the functionality to irregular cuboids, load balancing schemes must be additionally developed, which goes beyond the scope of this work.

[3] For brevity, henceforth, we shall take "tensor" to mean the order-3 tensor.

$X \in \mathbb{C}^{N\times N\times N}$ existing in a 3D space defined by orthogonal directions $1, 2, 3$, with the data $x(l, m, n)$ arranged as a cubic mesh. Now, we introduce a right product of X and a matrix $A \in \mathbb{C}^{N\times N}$ in terms of their mode-3 product as

$$Z_R = \rho_R(X, A) = X \times_3 A^T, \tag{5.2}$$

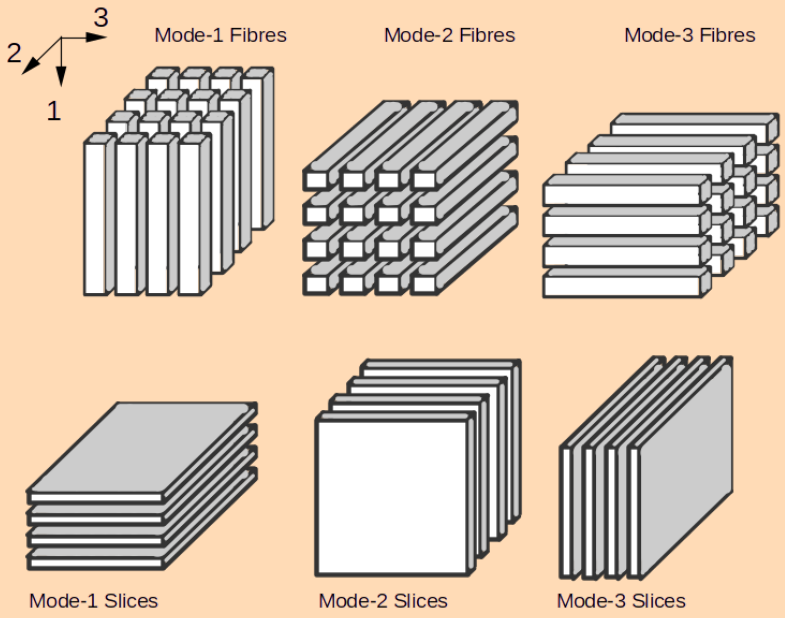
and a left product in terms of their mode-2 product* as

$$Z_L = \rho_L(X, A) = X \times_2 A^T, \tag{5.3}$$

where $Z_R, Z_L \in \mathbb{C}^{N\times N\times N}$." (A.)

DEFINITION : TENSOR

Multi-dimensional arrangement of real or complex values that generalizes vectors (*order-1 tensors*) and matrices (*order-2 tensors*) to arbitrary dimensions. Just as a matrix can be viewed in terms of its row and column vectors, so can a tensor be viewed in terms of its *fibres* and *slices* in at least six distinct ways.



"These products can be conceived as a set of independent matrix-matrix multiplications as detailed in algorithms 1 and 2. To visualize this, one can view X as a stack of N matrices $X^{(r)} \in \mathbb{C}^{N\times N}$ for $0 \leq r < N$, which we shall call the slices of the tensor, piled up along any of the 3 orthogonal directions. In practice, to ensure a contiguous memory layout for optimal data access, we fix the piling direction to direction 1, as illustrated in figure 5.1." (A.)

Algorithm 1 Procedure to perform the tensor-matrix multiplication ρ_R. Here, $Z_R^{(r)}$, $X^{(r)}$ and A are matrices. The tensors Z_R, X can be obtained by stacking their slices $Z_R^{(r)}$, $X^{(r)}$ along dimension 1 (see figure 5.1a).

1: **for** $r \leftarrow 0$ to N **do**
2: $\quad Z_R^{(r)} \leftarrow X^{(r)} A$
3: **end for**

Algorithm 2 Procedure to perform the tensor-matrix multiplication ρ_L. Here, $Z_L^{(r)}$, $X^{(r)}$ and A are matrices. The tensors Z_L, X are obtained by stacking the slices $Z_R^{(r)}$, $X^{(r)}$ along dimension 1 (see figure 5.1b).

1: **for** $r \leftarrow 0$ to N **do**
2: $\quad Z_L^{(r)} \leftarrow A^T X^{(r)}$
3: **end for**

DEFINITION : n-MODE PRODUCT

The n-mode product (referred to specifically as "mode-3" and "mode-2" products in (A.)) is obtained by multiplying each mode-n fibre of $X \in \mathbb{C}^{N \times N \times N}$ with a matrix $A \in \mathbb{C}^{N \times N}$ and is represented by $X \times_n A$. If X were to be *unfolded* into a matrix $X_{(n)}$ i.e., the data in X is arranged such that the mode-n fibres of X are the column vectors of $X_{(n)}$, then $X \times_n A = U X_{(n)}$.

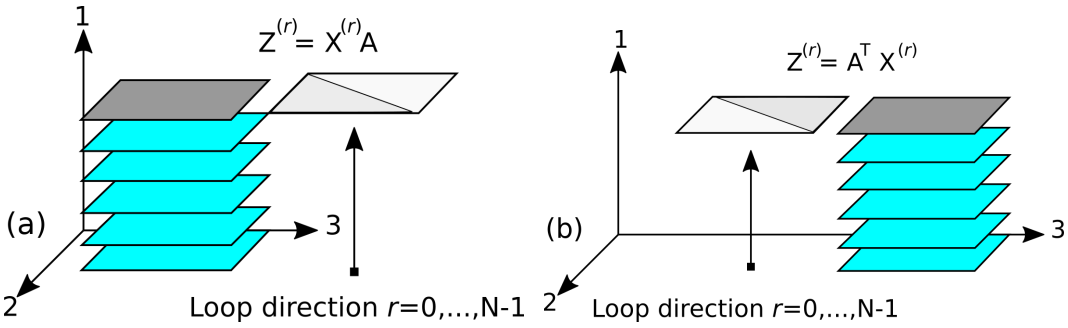


Figure 5.1: "Visualization of the procedures to compute the tensor-matrix multiplications (a) ρ_R and (b) ρ_L, as a set of independent matrix-matrix multiplications" (A.).

"Using the products defined by equations (5.2) and (5.3), it can be shown that the 3D DFT equation (5.1) can be rewritten in terms of tensor-matrix multiplications as

$$Y = \tau_2(\rho_R(\tau_2(\rho_L(\rho_R(X, C), C)), C)) \tag{5.4}$$

where $X, Y \in \mathbb{C}^{N\times N\times N}$ are the input and output tensors respectively, and $C \in \mathbb{C}^{N\times N}$ is the DFT matrix. The operation $\tau_k : \mathbb{C}^{N\times N\times N} \to \mathbb{C}^{N\times N\times N}$ indicates the transposition of the slices along piling direction $k \in \{1, 2, 3\}$ of the operand tensor. In this specific case, this is applied along direction $k = 2$.

In the implementation, the calculation of the transform via equation (5.4) can be performed in three stages [130]. In the first stage, the tensor-matrix multiplication of the input data with the DFT matrix is carried out as per algorithm 1,

$$\dot{Y} = \rho_R(X, C). \tag{5.5}$$

In the second stage, a similar procedure is performed, this time using the output of the first stage, and as per algorithm 2,

$$\ddot{Y} = \rho_L(\dot{Y}, C). \tag{5.6}$$

In the third stage, the piling direction for the tensor-matrix multiplication changes from 1 to 3. Consequently, in order to ensure a contiguous memory layout for the subsequent multiplication, a preliminary step must be performed in which $\ddot{Y}$ is subjected to the transpose operation represented by τ_2. After this, the final tensor-matrix multiplication is carried out as per algorithm 1. At the end, the transpose operation τ_2 is applied once more to arrange the result in the same spatial layout as the input tensor X. Putting everything together, the third and final stage implements the following operations:

$$Y = \tau_2(\rho_R(\tau_2(\ddot{Y}), C)). \tag{5.7}$$

The tensor Y contains the result of the forward DFT operation of equation (5.4)" (A.).

For brevity's sake, we have focused only on the forward transform and omitted an explanation of the inverse transform, which can be obtained in a straightforward and similar fashion using the inverse DFT matrix.

5.2.2 Adaptation of Cannon's Algorithm for Block Tensor-Matrix Multiplication

"In this subsection, we provide the designs of the procedures which make use of the basic idea of Cannon's algorithm [132] to perform the operations of equations (5.5), (5.6), (5.7), resulting in three unique distributed-memory block tensor-matrix multiplication algorithms.

In order to enable the use of as many Processing Elements (PEs) as possible, the tensors X and Y from equation (5.4) are subjected to the volumetric domain decomposition [126, 129, 130], resulting in a cubic mesh comprising p^3 PEs (see figure 5.2). Each PE(i, j, k) is accessible via indices $0 \leq i, j, k < p$, and has locally allocated operand and result block tensors

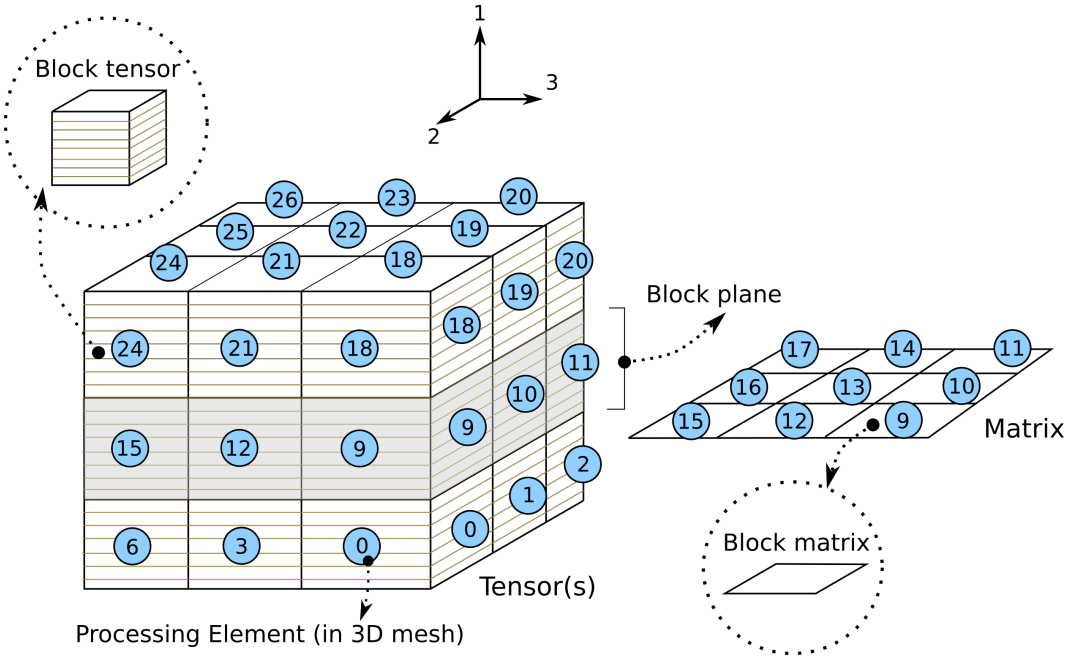


Figure 5.2: "A tensor and a matrix are broken down into p^3 and p^2 blocks (here, $p = 3$), respectively. Each block is located in the memory of its corresponding PE, as indicated by the number in the circles" (A.).

$X^{(b)}_{i,j,k}, Y^{(b)}_{i,j,k} \in \mathbb{C}^{b\times b\times b}$, and an operand block matrix $C^{(b)}_{j,k} \in \mathbb{C}^{b\times b}$, which is obtained by matrix decomposition. The block size is given by $b = N/p$.

Figure 5.2 is a schematic representation of volumetrically decomposed data with $p = 3$. In this example, it is shown how a tensor can be decomposed into p^3 block tensors, each of which is associated with and contained in the memory of an independent PE. Moreover, it can be seen how an operand matrix can be distributed to PEs in each block plane by decomposing it into p^2 block matrices and assigning each block to a PE. A pictorial definition of the block plane is also provided. To help the reader, in Table 5.1 we report the notation used in the following discussion of the algorithms to indicate the key elements depicted in figure 5.2" (A.).

DEFINITION : BLOCK PLANE

The plane containing all the block tensors that can be identified using a particular direction-1 index $0 \leq i < p$.

"We will start by focusing on the block tensor-matrix multiplication involved in the first stage, given by equation (5.5). From the description of ρ_R in algorithm 1 we recall that the slices of X along piling direction 1 are

Table 5.1: "Notation used in algorithms 3, 4, 5, 6 and 7. See figure 5.2 for a schematic representation of the PE mesh and the corresponding locally allocated block tensors and block matrices. More details are provided in the text" (A.).

PE(i,j,k)	A PE located at indices (i,j,k) in the cubic mesh
$X^{(b)}_{i,j,k}$	Locally allocated operand block tensor
$Y^{(b)}_{i,j,k}$	Locally allocated result block tensor
$C^{(b)}_{i,j}$	Locally allocated DFT block matrix
$X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$	Tensor matrix multiplication (algorithm 1)
$C^{(b)T}_{j,k} * X^{(b)}_{i,j,k}$	Tensor matrix multiplication (algorithm 2)

to be multiplied with C (see also figure 5.1a). Analogously, here, the block *tensors* in each plane orthogonal to piling direction 1 are to be multiplied with C. The multiplication of each such block plane with C can be executed independently, and hence, parallely. It is for this multiplication that we can utilize a scheme similar to that of the original Cannon's algorithm, with the essential deviation that one of the operand matrices and the result matrix are replaced by operand and result tensors, respectively. The adapted algorithm is composed of an alignment phase and a computation phase, with a total of p communication events. We define here a communication event as a set of multiple, parallel message passing calls[4]. The alignment phase consists of a single communication event involving $2p^2(p-1)$ parallel messages, whereas the computation phase consists of $p-1$ communication events, each involving $2p^3$ parallel messages respectively.

The algorithm for ρ_R is as outlined in algorithm 3. In the alignment phase, the block tensors $X^b_{i,j,k}$ and block matrices $C^b_{i,j}$ are redistributed to different PEs following the rules detailed in lines 1-6 of the procedure. In the computation phase, the operation in line 8, which we refer to as the local update, can be performed slice-wise as per algorithm 1, either sequentially, or, when additional computational resources are available to each PE, using a separate mode of parallelism, giving rise to multi-level parallelism. Additionally, the communication event (lines 10 and 11) and the local update can be executed in parallel. At the end of the computation phase, the block tensor $Y^{(b)}_{i,j,k}$ located in each PE contains the block result of the desired product. We observe here that the communication of block tensors (line 10) occurs between PEs that are neighbours along direction 3 of the PE mesh.

The algorithm for the multiplication of the second stage, given by equation (5.6), is outlined in algorithm 4. It is very similar to that of the first

[4]Conceptually, parallel messages are passed independently and simultaneously by all PEs and thus the duration of a single communication event is decided by the most time-consuming message passing.

Algorithm 3 "Procedure of the adapted Cannon's algorithm for the function $Y = \rho_R(X, C)$. See Table 5.1 for definitions" (A.).

▷ Alignment phase:
1: **if** $j \neq 0$ **then**
2: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-j) \bmod p)$
3: **end if**
4: **if** $k \neq 0$ **then**
5: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-k) \bmod p,k)$
6: **end if**
▷ Computation phase:
7: **for** $r \leftarrow 0$ to p **do**
8: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
9: **if** $r < p-1$ **then**
10: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-1) \bmod p)$
11: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
12: **end if**
13: **end for**

Algorithm 4 "Procedure of the adapted Cannon's algorithm for the function $Y = \rho_L(X, C)$. See Table 5.1 for definitions" (A.).

▷ Alignment phase:
1: **if** $k \neq 0$ **then**
2: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-k) \bmod p,k)$
3: **end if**
4: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,k,(j+p-k) \bmod p)$
▷ Computation phase:
5: **for** $r \leftarrow 0$ to p **do**
6: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + C^{(b)T}_{j,k} * X^{(b)}_{i,j,k}$
7: **if** $r < p-1$ **then**
8: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
9: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-1) \bmod p)$
10: **end if**
11: **end for**

stage, albeit with minor changes in the pattern of communication. Here, it is worth highlighting the fact that the communication of block tensors (line 8) occurs between PEs that are aligned along direction 2 of the PE mesh.

The algorithm for the third stage needs to incorporate the transpose operations represented by τ_2 in equation (5.7). Although these global transpositions are unavoidable, we can avoid communication events by transposing the PE mesh instead of the data itself. All the same, a local (i.e., not involv-

Algorithm 5 "Procedure of the adapted Cannon's algorithm for the function $Y = \tau_2(\rho_L(\tau_2(X), C))$. See Table 5.1 for definitions" (A.).

▷ First local transposition:
1: $X^{(b)}_{i,j,k} \leftarrow \tau_2(X^{(b)}_{i,j,k})$
▷ Alignment phase:
2: **if** $j \neq 0$ **then**
3: Send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}((i+p-j) \bmod p, j, k)$
4: **end if**
5: Send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(k, (j+p-k) \bmod p, i)$
▷ Computation phase:
6: **for** $r \leftarrow 0$ to p **do**
7: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
8: **if** $r < p-1$ **then**
9: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}((i+p-1) \bmod p, j, k)$
10: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i, (j+p-1) \bmod p, k)$
11: **end if**
12: **end for**
▷ Final local transposition:
13: $Y^{(b)}_{i,j,k} \leftarrow \tau_2(Y^{(b)}_{i,j,k})$

ing communication) data transpose function is still required to perform the transposition of the operand and result block tensors. The procedure of the adapted Cannon's algorithm for the third stage is provided in algorithm 5. Note that, in this case, the communication of block tensors (line 9) occurs between PEs that are aligned along direction 1 of the PE mesh" (A.).

5.2.3 Global Transpose Variants

"Our performance analysis of the distributed-memory block tensor-matrix multiplication functions (see subsection 5.5.2) showed that the direction along the PE mesh in which point-to-point communication between neighbours takes place crucially decides the time of communication. This is due to how the communication patterns of the algorithms are mapped into subsets of the fat tree topology. To understand this, let us consider a simplified visual example with $p = 2$ and a fat tree network with two servers per edge, as shown in figure 5.3. In the PE mesh, neighbours along all three directions are equidistant pairs. However, in the network, depending on the stride between the neighbours, messages may cross 1, 3 or 5 switches, which leads to three possible, distinct communication rates.

In practice, similarly, if p is large enough in relation to the number of nodes on the server level in the allocation, up to three distinct communication rates can be realized. In our tests, we observed that the communication rate is the

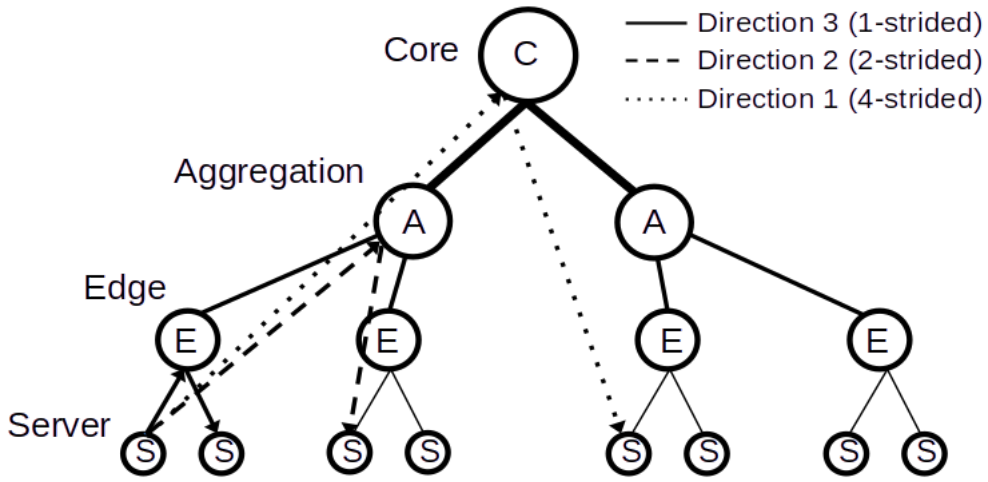


Figure 5.3: "Neighbours along directions 3, 2 and 1 in the PE mesh (not shown here) are 1-strided, 2-strided and 4-strided, respectively. This means that messages between neighbours along directions 3, 2 and 1 must cross 1, 3 and 5 switches, respectively. This leads to different communication rates" (A.).

highest for direction 3 in the PE mesh, followed by direction 2 and finally, direction 1.

From algorithms 3, 4 and 5, it can be seen that the computation phase involves $p - 1$ more communication events than the alignment phase, a number which scales linearly with the node count. To limit the communication to direction 3 as much as possible, we implemented variants of algorithms 4 and 5, in which the data is globally transposed in the alignment phase such that, in the computation phase, communication only occurs between neighbours along direction 3. This strategy has additional costs: (i) both procedures include an additional communication event after the computation phase, in which the transposed data is transposed back; (ii) in both procedures, local data transpositions become necessary before the alignment phase and after the computation phase. The procedures for these variants are provided in algorithms 6 and 7" (A.).

5.2.4 Modeling the overlapping of the communication and the computation

"In the computation phase of all the distributed-memory algorithms discussed in the previous subsections, the communication event and the local update can be overlapped to hide the latency of communication. To identify configurations of block size and node count for which the highest efficiency

Algorithm 6 "Procedure of the global transpose variant of the adapted Cannon's algorithm for the function $Y = \rho_L(X, C)$. See Table 5.1 for definitions" (A.).

▷ First local transposition:
1: $X^{(b)}_{i,j,k} \leftarrow \tau_1(X^{(b)}_{i,j,k})$
▷ Alignment phase:
2: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,k,(j+p-k) \bmod p)$
3: **if** $k \neq 0$ **then**
4: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-k) \bmod p,k)$
5: **end if**
▷ Computation phase:
6: **for** $r \leftarrow 0$ to p **do**
7: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
8: **if** $r < p-1$ **then**
9: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,j,(k+p-1) \bmod p)$
10: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,(j+p-1) \bmod p,k)$
11: **end if**
12: **end for**
▷ Global transpose:
13: send $Y^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i,k,j)$
▷ Final local transposition:
14: $Y^{(b)}_{i,j,k} \leftarrow \tau_1(Y^{(b)}_{i,j,k})$

of a block tensor-matrix multiplication algorithm can be attained, it is useful to separately model the time of the communication event and the time of the local update in the problem size range of interest, as a function of the block size. To this end, microbenchmark programs that exactly imitate the communication event and the local update can be used. The obtained times can then be fitted as polynomials. The points of intersection of the two fitted curves fulfill the condition of maximum efficiency of the overlapping under which the algorithm does not incur any communication overhead. This is also the case in block size intervals in which the time of the local update is always greater than that of communication. We used this approach to select the configurations of block size and node count in our performance analysis in subsection 5.5.2" (A.).

Algorithm 7 "Procedure of the global transpose variant of the adapted Cannon's algorithm for the function $Y = \tau_2(\rho_L(\tau_2(X), C))$. See Table 5.1 for definitions" (A.).

▷ First local transposition:
1: $X^{(b)}_{i,j,k} \leftarrow \tau_2(X^{(b)}_{i,j,k})$
▷ Alignment phase:
2: Send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(k, j, (i+p-j) \bmod p)$
3: **if** $k \neq 0$ **then**
4: Send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i, (j+p-k) \bmod p, k)$
5: **end if**
▷ Computation phase:
6: **for** $r \leftarrow 0$ to p **do**
7: $Y^{(b)}_{i,j,k} \leftarrow Y^{(b)}_{i,j,k} + X^{(b)}_{i,j,k} * C^{(b)}_{j,k}$
8: **if** $r < p-1$ **then**
9: send $X^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i, j, (k+p-1) \bmod p)$
10: send $C^{(b)}_{j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(i, (j+p-1) \bmod p, k)$
11: **end if**
12: **end for**
▷ Global transpose:
13: send $Y^{(b)}_{i,j,k}$: $\text{PE}(i,j,k) \rightarrow \text{PE}(k,j,i)$
▷ Final local transposition:
14: $Y^{(b)}_{i,j,k} \leftarrow \tau_2(Y^{(b)}_{i,j,k})$

5.3 Details of Implementation

"The 3D DFT by block tensor-matrix multiplication algorithm was implemented as a C++ library named S3DFT, which has a distributed-memory Application Programming Interface (API —the set of functions which are to be used to compute the 3D DFT) in both single and double precision. The library was built using the Intel compiler and the Intel MKL and Intel MPI libraries, which are part of the Intel OneAPI v2021.4.0 toolkit suite. It is open-source software available for use under the GNU Lesser General Public License v3 (LGPL) [136].

S3DFT combines the use of shared- and distributed-memory parallelism by means of an OpenMP/MPI hybrid approach. On the shared-memory level, the computational work is identified by a set of slices of the operand tensor. In all shared-memory functions, we use a custom work-sharing function, which (provided the number of threads is constant) always returns the same set of indices of operand tensor slices for any given thread. This tactic ensures that the association between operand data and threads does not change. Combined with proper run-time thread pinning, it strongly reduces

the number of non-local memory accesses in NUMA systems, which are significantly slower than local memory accesses. This design makes sure that S3DFT can be run with threads distributed across NUMA domains without a performance drop. The excellent scaling of S3DFT across NUMA domains is demonstrated by the performance analysis presented in subsection 5.5.1.

Initial profiling of the distributed-memory block tensor-matrix multiplication functions that implement algorithms 3, 4 and 5 (see subsection 5.5.2) revealed that the time of communication *significantly exceeds that of the local update*, which confirms our initial speculation that the communication (and not the computation) would be the bottleneck. To hide the latency of communication, we decided to dedicate one thread to communication. Our work-sharing function fulfills this requirement by returning an empty set for the communicating thread and distributing the slices of the operand tensor amongst the remaining threads in a round-robin fashion.

The local update mentioned in subsection 5.2.2 has been realized by a shared-memory tensor-matrix multiplication function, which computes the product as a set of parallel matrix-matrix multiplications as shown in algorithms 1 and 2. For this, we used the CBLAS implementation provided by the Intel MKL v2021.4.0. We see from the strong scaling of the function (figure 5.4b) that the difference in performance when all 48 cores are used and when 47 cores are used is small, with the performance decreasing only slightly from 88% to 85% of the single node peak performance. Hence, the use of a dedicated communication thread does not reduce the performance of the local update much" (A.).

5.4 Specifications of the JUWELS Cluster

"All benchmarks and tests in this work have been executed on the JUWELS Cluster [32], which uses a fat tree network with InfiniBand interconnects. The standard compute node has two cache-coherent NUMA domains. The salient specifications are listed in table 5.2, and the peak bandwidths as obtained with the Intel Advisor tool [137], in table 5.3. These numbers were used as reference values while analysing the performance of the core functions of the implementation.

Since Intel did not explicitly provide information on peak performance in terms of FLOP/s at the time of writing of this article [138], the peak performance of the compute node had to be estimated using the published specifications. The base frequency of the Intel Xeon® Platinum 8168 processor is 2.7 GHz [139]. However, when all cores are active and the use of AVX-512 instructions is maximized, the clock frequency drops to 2.5 GHz [139]. The processor is equipped with 2 AVX-512 Fused Multiply-Add (FMA) units per

Table 5.2: "Specifications of the standard compute node of the JUWELS cluster" (A.).

Processor	Intel Xeon® Platinum 8168 (Skylake)
CPU count	2 (sockets)
Core count	48 cores (24 cores per CPU)
SMT/HT	Available, 96 threads (48 threads per CPU)
Clock frequency	[1.2 - 3.7 GHz], base @ 2.7 GHz
Cache	L1 - 32 kB, L2 - 1 MB, L3 - 33 MB
DRAM	96 GB DDR4 @ 2666 MHz

Table 5.3: "Cache and memory bandwidths according to Intel Advisor" (A.).

	Bandwidth	
	1x NUMA	2x NUMA
L1	11.6 TB/s	23.2 TB/s
L2	5.5 TB/s	11.0 TB/s
L3	649 GB/s	1299 GB/s
DRAM	115 GB/s	230 GB/s

core, which yields a theoretical peak performance of 3840 GFLOP/s. Corroborating this estimation, Intel Advisor's roofline chart includes information about the double-precision FMA peak performance [137], which in this case is 3812 GFLOP/s. Henceforth, we refer to this value when we speak about the peak performance of the node"(A.).

5.5 Performance Analysis of Components

"In this section, we present the performance analysis of the core functions of the S3DFT implementation. For this analysis, the functions were microbenchmarked on the JUWELS Cluster in double precision. The open-source library TiXL, which is available under the LGPL v3, was used for this purpose [37].

The microbenchmark programs consist of (i) an initialization phase, in which operand/result data are allocated afresh, and each OpenMP thread (excepting the communication thread) accesses the first word of each memory page of its associated data —thereby ruling out the measuring of page-faults, (ii) an experiment phase, in which the function of interest is run, and (iii) a clean-up phase, in which all data is freed. Each benchmark test was concluded by performing 20 warm-up and 100 timed runs. The experiment phase of only the latter runs was used to measure the duration of the func-

tion of interest. The result was calculated as the geometric mean of these measurements.

Via mircobenchmarking, we present a breakdown of the run time of the main API function in table 5.4 for node count 343 (i.e., $p = 7$) and problem size $N = 4200$. Since the shared-memory tensor zeroing and transpose functions constitute a negligible fraction of the total run time, they have been excluded from the section. For completeness, a performance analysis of the local transpose function is still reported in C.1" (A.).

Table 5.4: "Breakdown of S3DFT's main API function for $p = 7$ and $N = 4200$" (A.).

Distributed-Memory Function	**Run Time Percentage**
Block Tensor-Matrix Multiplication (algorithm 3)	21%
Transp. Block Matrix-Tensor Multiplication (algorithm 4)	35%
Transp. Block Tensor-Matrix Multiplication (algorithm 5)	44%
Shared-Memory Function	
Tensor-Matrix Multiplication, Transp. Matrix-Tensor Multiplication	9%
Set Zero Tensor	0.8%
Transpose Tensor	1%

5.5.1 Shared-Memory Tensor-Matrix Multiplication

"This function performs the local update operation of the distributed-memory tensor-matrix multiplication as mentioned in subsection 5.2.2.

We began the analysis by running problem scaling tests using all available cores in the single and dual NUMA configurations to identify the problem sizes at which peak performances of the function can be expected. Next, we conducted strong scaling tests for these problem sizes. The results are reported in figure 5.4. We observed a peak performance of 1671 GFLOP/s at $N = 900$ and of 3373 GFLOP/s at $N = 1100$ for the single and dual NUMA configurations, respectively. This corresponds to 88% of the peak performance. As shown in the right panel of figure 5.4, the function scales well.

To estimate the corresponding effective bandwidth[5], we must first model the traffic and computation requirements of algorithm 1. For a tensor of side N, a computer can perform $8N^4$ floating point operations[6] after

[5]The effective bandwidth is calculated using the run time and the *a priori* data traffic estimation.

[6]Each complex number addition and multiplication involves at least 2 and 6 FLOPs respectively.

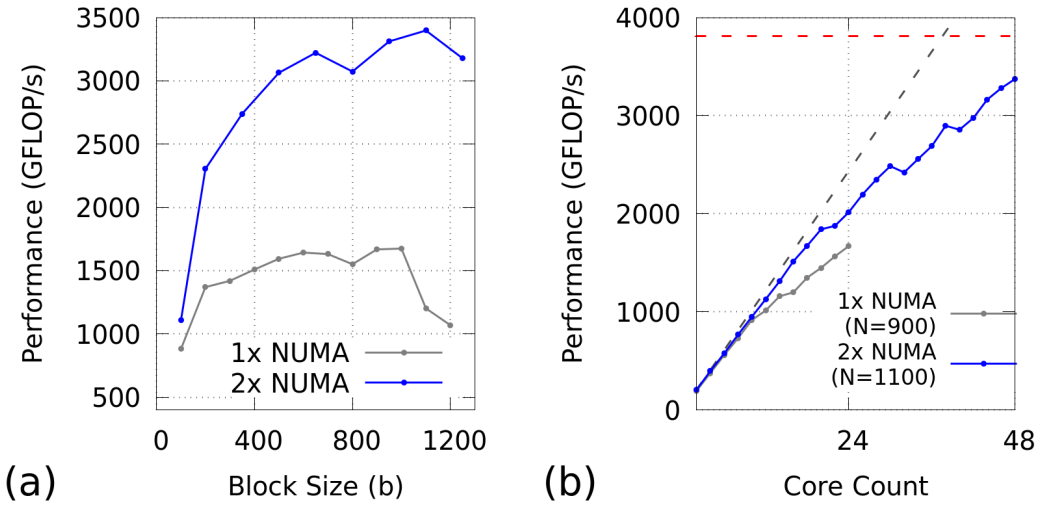


Figure 5.4: "Subfigures show (a) problem scaling as a function of block size and (b) strong scaling of the shared-memory tensor-matrix multiplication. Grey and blue curves are for single (24 cores) and dual (48 cores) NUMA confgurations, with $N = 900$ and $N = 1100$, respectively. The red line in subfigure (b) indicates the peak performance of the single node, while the grey dashed line indicates ideal linear scaling" (A.).

$2N^3(N+1)$ transfers. For double precision, we have the code balance given by $B_c = \frac{4(N+1)}{N} \approx 4$ B/FLOP. Using the roofline model, we can calculate the effective bandwidth as $b_s = B_c P$, where P is the attained performance [13, p. 66]. Following this, we can estimate peak effective bandwidths $b_s = 6.7$ TB/s and $b_s = 13.5$ TB/s for the single and dual NUMA configurations, respectively, which are greater than the corresponding L2-cache bandwidths listed in table 5.3. This can be taken to conclude that the function makes excellent use of caching" (A.).

5.5.2 Distributed-Memory Block Tensor-Matrix Multiplication

"In this analysis, we consider the implementations of algorithms 3, 4 and 5.

First, following the approach outlined in subsection 5.2.4, we identified the configurations of block size and node count for which the highest efficiency of the algorithm 3 can be reached. For this purpose, we designed microbenchmark programs which exactly imitate the communication event and the local update, respectively, and ran them for $p = 2, 3, 4, 5, 6$ and $b \in [100, 1300]$. As shown in figure 5.5, we then fitted the obtained times with cubic polynomials to model the time of the communication event for

the node count of interest (continuous curve), and a quartic polynomial for that of the local update (dotted curves). The optimal block sizes for the above-stated node counts were found to be $b > 750$ for the 2 MPI tasks/node (single NUMA) and $b > 600$ for the 1 MPI task/node (dual NUMA) configurations, respectively.

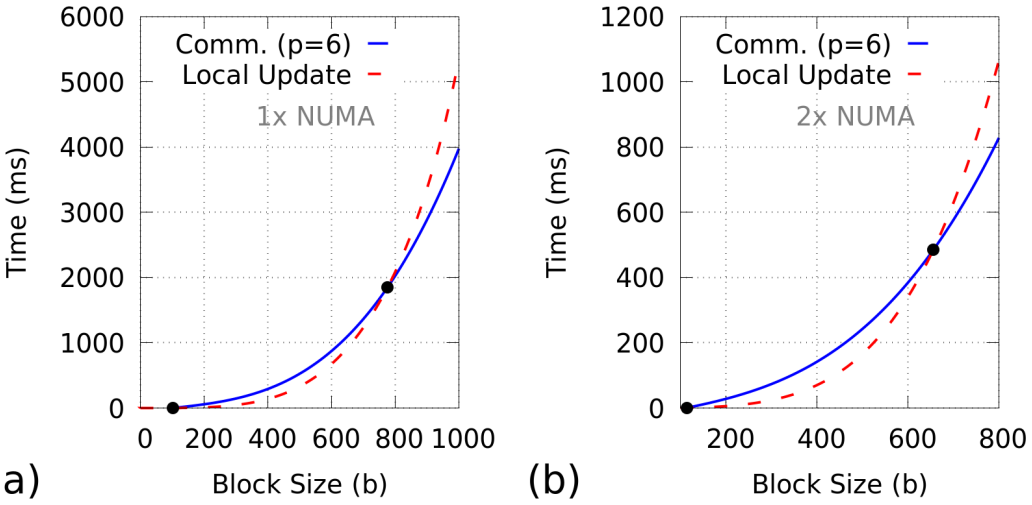


Figure 5.5: "Fitted curves model the time of the communication event (continuous lines) and that of the local update (dashed line) as functions of block size. The black dots indicate the intersections at which perfect overlapping can be expected. Subfigures (a) and (b) report results obtained in 2 MPI tasks/node and 1 MPI task/node configurations, respectively" (A.).

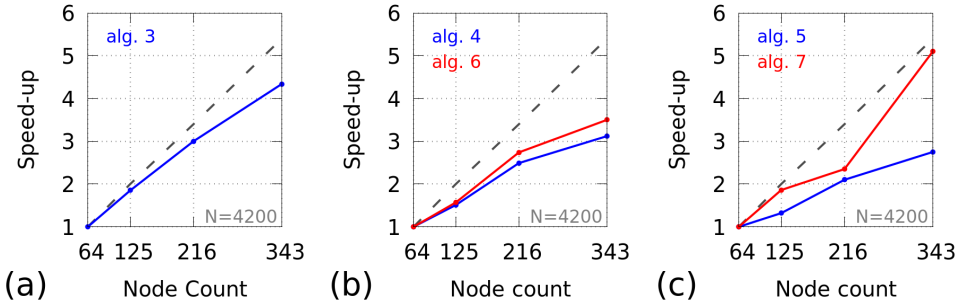


Figure 5.6: "The blue curves in subfigures (a), (b) and (c) show the strong scaling performance of the implementations of algorithm 3, algorithm 4 and algorithm 5, respectively, at problem size $N = 4200$, in the 1 MPI task/node configuration. The grey dashed lines indicate ideal linear scaling. The red curves in subfigures (b) and (c) show the results obtained using the global transpose variants 6 and 7, respectively" (A.).

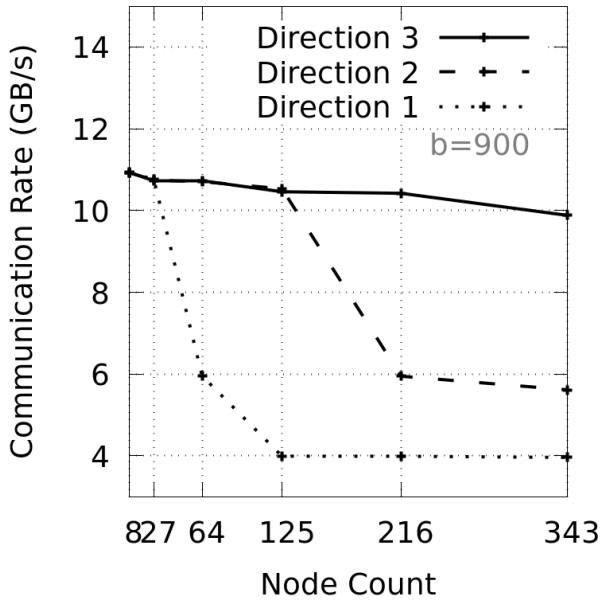


Figure 5.7: "Communication rate as a function of node count with $b = 900$. The continuous, dashed and dotted lines correspond to the communication rates of neighbours along directions 3, 2 and 1, respectively" (A.).

Next, we conducted strong scaling tests for the algorithms 3, 4 and 5 in the 1 MPI task/node configuration for the problem size $N = 4200$ and $p = 4, 5, 6, 7$, corresponding to block sizes $b = 1050, 840, 700, 600$, at which the latency of the communication event is expected to be completely hidden by the overlapping (see figure 5.5b). The results are reported in figure 5.6, showing parallel efficiencies[7] in the ranges of $81\% - 95\%$, $58\% - 77\%$ and $51\% - 68\%$ for algorithms 3, 4 and 5, respectively.

To understand the poor scaling of algorithms 4 and 5, we microbenchmarked the code of the communication event in the computation phase of the three block tensor-matrix multiplication algorithms. As shown in figure 5.7, the results revealed that the communication rate varies considerably with the direction along which the neighbours are identified, as the node count is increased. Interestingly, the communication rate between neighbours along direction 3 in the PE mesh is the highest, followed by direction 2 and finally, direction 1. As noted in subsection 5.2.2, in the computation phase of the algorithms 3, 4 and 5, most of the communication occurs between neighbours along directions 3, 2 and 1, respectively. This explains the scaling performance of each algorithm.

Finally, on another note, we observe that the performance of the local up-

[7]Here, the parallel efficiency has been evaluated relative to $p = 4$ case (i.e., with 64 nodes) which is the minimum number of nodes we could use for the given problem size due to memory limitations.

date reduces with reducing block size (see the left panel of figure 5.4), which warns us of a reduction in the shared-memory resource utilization as the node count scales up, which predicts an additional cause for the worsening of the strong-scaling performance of the above-mentioned algorithms" (A.).

5.5.3 Global Transpose Variants

"In an attempt to overcome the problem discussed in the previous section, we implemented variants of algorithms 4 and 5 in which the transposition of the PE mesh is replaced by the global transposition of the data, as outlined by algorithms 6 and 7. This resulted in improvements in the strong-scaling performance (see red curves in figures 5.6b and 5.6c), with parallel efficiencies of $65\% - 81\%$ and $70\% - 95\%$ for algorithms 6 and 7, respectively. However, the overall performance was found to suffer considerably due to the additional communication events required" (A.).

5.6 Comparison with Intel MKL and FFTW3

"Here, we present the results of the benchmarking of S3DFT against two competitive 3D FFT implementations: Intel MKL v2021.4.0[8] and FFTW3 v3.3.10. The benchmarking procedure is identical to that outlined in section 5.5, with the exception that 70 warm-up runs and 50 timed experiments were conducted. In the plots included in this subsection, we report the geometric mean of the run time. In the programs that recorded the performance of the cluster-based Intel MKL/FFTW3 libraries, multithreading was initialized as per the manual [140]. The FFTW-plan [124] was created in the initialization phase of the program using the flag `FFTW_MEASURE`. To be able to benchmark under reproducible conditions, contiguous node allocation was requested. Within a single compute node, a thread-placement policy of 1 thread/core was applied. Further, each thread was pinned to avoid thread migrations by the operating system during runtime.

Initial testing showed that both Intel MKL and FFTW3 performed much better when launched with 2 MPI tasks/node, which corresponds to 1 NUMA domain/MPI task. Therefore, their benchmarking programs were always run using this configuration. S3DFT was found to perform similarly in both the 1 MPI task/node and 2 MPI tasks/node configurations.

In what we call small problem scale, strong scaling comparisons were conducted for problem sizes $N = 120, 240, 480, 600, 840$. Similarly, in the large problem scale, we did the strong scaling comparisons for problem sizes $N = 2520, 3360, 4200$. Moreover, these comparisons were conducted two-

[8] We made use of the convenient FFTW3 wrapper interface provided by Intel MKL, which makes use of its implementation of cluster FFT functions.

fold, with S3DFT in the 1 MPI task/node and 2 MPI tasks/node configurations. In all these comparisons, we observed similar scaling behaviours and performances for all investigated problem sizes. The results for sizes $N = 840, 3360$ are provided in figures 5.8 and 5.9.

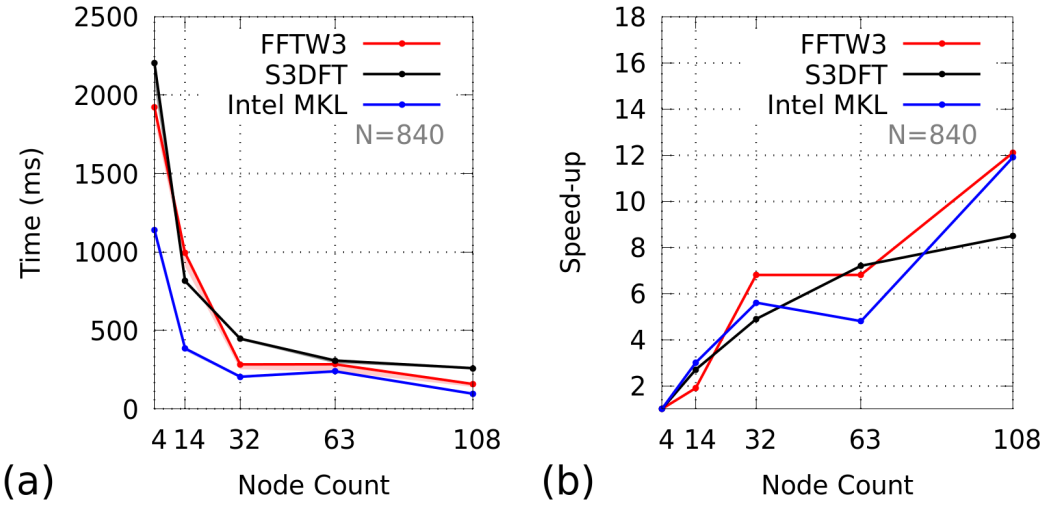


Figure 5.8: "Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node" (A.).

Results show that Intel MKL was consistently the fastest across all problem sizes and node counts, followed by FFTW3. On average, in the small problem scale and with S3DFT in the 2 MPI tasks/node configuration, Intel MKL was 2.5 times faster and FFTW3 was 1.8 times faster than S3DFT. With S3DFT in the 1 MPI task/node configuration, Intel MKL was 2.2 times faster and FFTW3 was 1.5 times faster than S3DFT. Frequently, for small node counts, S3DFT was found to be slightly faster than FFTW3 (see, e.g., figure 5.8a). In the large problem scale, Intel MKL was 3.7 times faster and FFTW3 was 2.3 times faster than S3DFT.

The benchmarking was repeated for the same problem scales, sizes, node counts and MPI tasks/node configurations with the implementations of algorithms 4 and 5 being replaced by the global transpose variants (GTV) 6 and 7. As can be seen in figures 5.10 and 5.11, the result of the comparison with FFTW3 and Intel MKL remained essentially unchanged, even in the large problem scale, despite the significant improvement of S3DFT's scaling. Specifically, in the small problem scale and with S3DFT in the 2 MPI tasks/node configuration, Intel MKL was 3.0 times faster and FFTW3 was

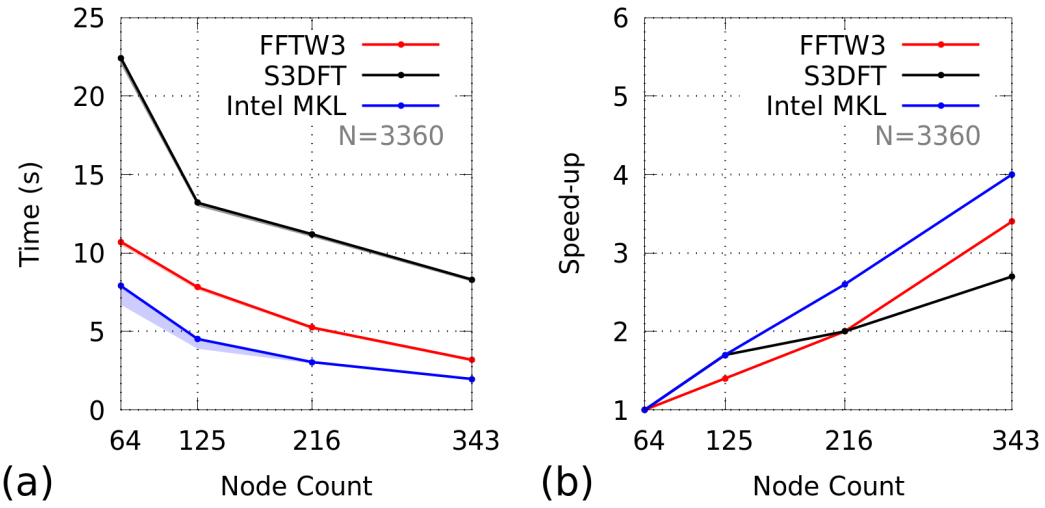


Figure 5.9: "Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node" (A.).

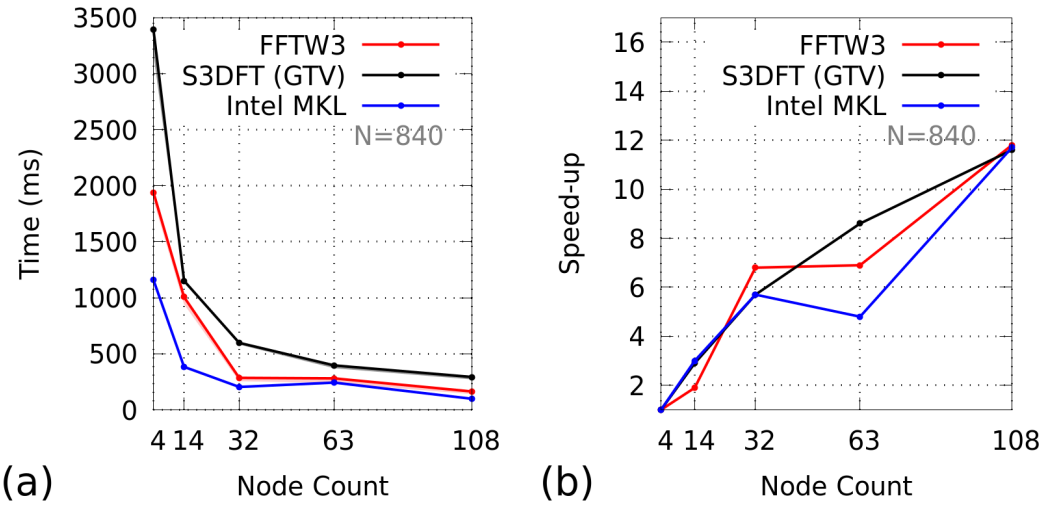


Figure 5.10: "In this comparison, S3DFT uses the global transpose variants as detailed in subsection 5.2.3. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node" (A.).

2.2 times faster than S3DFT. With S3DFT in the 1 MPI task/node configuration, Intel MKL was 2.4 times faster and FFTW3 was 1.7 times faster

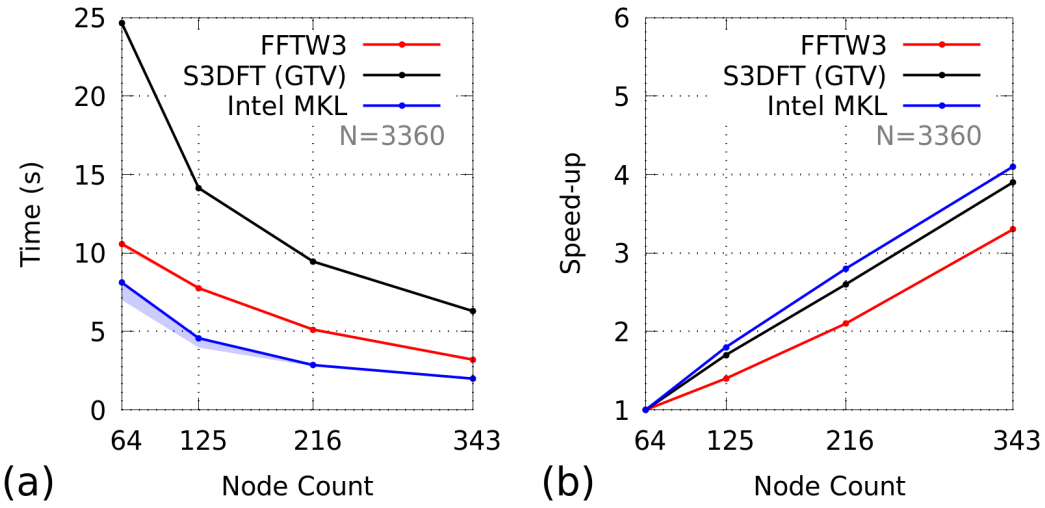


Figure 5.11: "In this comparison, S3DFT uses the global transpose variants as detailed in subsection 5.2.3. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node" (A.).

than S3DFT. In the large problem scale, Intel MKL was 3.6 times faster and FFTW3 was 2.2 times faster than S3DFT.

We note that other parallel 3D FFT implementations might have higher scaling limits than FFTW3 and Intel MKL. This is to be expected especially for libraries that make use of the pencil decomposition, such as FFTK [128], for which scaling up to $160k$ cores has been demonstrated. A comparison of S3DFT with FFTK is provided in C.2. Briefly, FFTK was found to be on average 4 times faster than S3DFT" (A.).

5.7 Conclusions

"We have presented a new parallel algorithm that exploits block tensor-matrix multiplication to compute the 3D DFT of a volumetrically decomposed, cubic domain using point-to-point communication. The algorithm has been implemented as a C++ library called S3DFT capable of utilizing shared-memory parallelism across multiple NUMA domains within a single node and distributed-memory parallelism across multiple nodes. In the process, we designed, developed and tested three adapted variants of Cannon's algorithm. These adaptations enable the use of tensor operands, realize multi-level parallelism and make efficient use of the technique of overlapping computation and communication with the help of a custom

work-sharing function for OpenMP-based multithreading. S3DFT has been optimized for the JUWELS Cluster and its core functions have been analyzed to show their efficiency. Finally, the performance of S3DFT has been compared with those of well-established, widely-used libraries for a wide range of problem sizes.

Our analysis by microbenchmarking has shown that S3DFT has a highly efficient implementation on the shared-memory level. On the other hand, we found that while one of the three required distributed-memory block tensor-matrix multiplication algorithms scales excellently, the other two scale poorly. We identified the origin of this problem in the mapping of their communication patterns in subsets of the fat tree topology: this result exposes a potential drawback of running block-wise parallel algorithms on systems with fat tree networks, which is caused by increased communication latencies along specific Cartesian directions in the PE mesh.

In an effort to improve the scalability of S3DFT, we designed algorithmic variants of the poorly scaling components in which the majority of the communication occurs between PEs aligned along the Cartesian direction in which the communication was found to be the fastest. This was made possible by performing global data transpositions instead of transposing the PE mesh, which comes at the cost of additional communication events. Although these variants improved the strong-scaling performance of S3DFT, its overall performance did not change considerably.

Our results let us speculate that at the current stage, the 3D DFT by block tensor-matrix multiplication is not a viable alternative to modern FFT-based approaches on computer clusters using fat tree networks. Different results might be expected on different computer clusters. For example, it is possible that S3DFT performs and scales considerably better on a computer cluster based on a network topology which is isomorphic to the PE mesh. Unfortunately, we did not have access to such a system to verify this hypothesis" (A.).

Chapter 6

Summary and Outlook

In this doctoral project, we have attempted to improve the scalability of MD simulations in two separate endeavours. In the first endeavour, based on evidence that the parallel 3D FFT becomes a bottleneck in QM/MM simulations at high core counts, we conducted a very specific investigation in which we looked into the possibility of improving the performance of the 3D DFT using an alternative parallel approach. In the second endeavour, we crafted a high-performing implementation of a massively parallel MD strategy called MoP in the GROMACS code and PLUMED library.

6.0.1 Improving the Performance of the 3D DFT

Researchers in the past have attempted to study the scalability of a 3D DFT algorithm that uses volumetric decomposition (with scaling limit at N^3, where N is the side length of cubic data) in an attempt to obtain better performance than the parallel 3D FFT algorithm, which is most commonly implemented using slab (scaling limit at N) or pencil (scaling limit at N^2) decompositions. Additionally, the algorithm they implemented made use of point-to-point communication instead of parallel 3D FFT's all-to-all communication. As a trade-off, the algorithm required more computational work to be done. The motivation of the investigators was to be able to scale to higher node counts (and therefore obtain better performance) and to eliminate all-to-all communication. The algorithm was implemented and tested on the IBM Blue Gene/Q (with a 5D torus network). However, their implementation could only make use of 20% of the hardware resources of the single node [130]. The authors concluded with the statement, "Based on the scalability evaluation, we can expect that highly-parallel implementation of the 3D DFT with 3D data decomposition can be faster than that of the 3D FFT with existing 2D data decomposition especially in the extreme case when the number of nodes is close to the size of problem, i.e., close to

N^3."

Five years after this pioneering investigation, we were inspired to look into the viability of such an alternative approach, albeit using adapted Cannon's algorithms for block tensor-matrix multiplications, which also make use of only point-to-point communication between neighbours. Moreover, we planned to conduct our investigation on the JUWELS Cluster, which uses a fat-tree network topology. Therefore, the algorithmic and network differences made our work a new investigation in its own right, instead of a re-investigation. Our work was motivated by findings in the literature that a major portion of the parallel 3D FFT's time is spent in participating in all-to-all communication, and by the fact that there is data to suggest that, depending on the cluster, point-to-point communication could scale better than all-to-all communication. We speculated that, even with the additional computational work, the communication would ultimately be the bottleneck.

We implemented our algorithm as a C++ library called S3DFT, which showed a much higher hardware resource utilization of 85% of the single node than the implementation of Ref. [130]. Moreover, as speculated, the communication was found to be the bottleneck of S3DFT. This means that improving the computational performance e.g., by porting to GPUs, *cannot* improve the overall performance of the algorithm. In spite of its highly efficient implementation, we observed unimpressive strong-scaling performance of our algorithm. Our analysis revealed poor scaling of two (of the three) adapted Cannon's algorithms for block tensor-matrix multiplications. Experiments showed that this was caused by a combination of the algorithmic requirement of communication in all directions of the PE mesh and the network characteristic that communication tended to be slower in certain directions. To combat this, we implemented algorithmic variants of the poorly functioning components that only communicate in that direction of the PE mesh that corresponds to the network's fastest communication rate. However, this came with the price of additional global transpose operations. The result improved the scalability of our algorithm considerably but also significantly worsened the initial performance.

State-of-the-art parallel 3D FFTs are still considerably faster than our algorithm. This may be attributed to their advanced optimization measures such as runtime performance tuning [124, 141], network topology-aware re-mapping and rotating the PE mesh to incur minimum performance loss [142] and using non-blocking all-to-all communication to overlap computation and communication [141]. Simply put, it has been possible to considerably optimize parallel 3D FFT's all-to-all communication. However, to the best of our knowledge, the same cannot be said of methods that make use of block tensor-matrix multiplication to compute the 3D DFT. There-

fore, future research in this direction would do well to focus on improving the communication of such methods.

6.0.2 High-Performance Implementation of MoP

Studying a physical phenomenon by means of cMD commonly involves navigating a rugged free-energy landscape with many local minima corresponding to metastable states of the simulated system. Studying the transition of the system from one metastable state to another arbitrary metastable state by natural dynamics, when they are separated by a large free-energy barrier, would require impractically long simulations. In this scenario, since we are interested in sampling the transition path but ultimately sample many paths that do not describe the transition, we say the simulation has low sampling efficiency. To increase the sampling efficiency, numerous specialized techniques have been developed that are collectively referred to as enhanced-sampling methods. One such technique is metadynamics, in which the Hamiltonian of the system is modified cumulatively by adding a bias potential, which accelerates the transition by lifting the simulated system out of the minimum.

The accumulation of bias potentials changes the natural dynamics of the system, which can only be recovered if a specially engineered bias potential is used. There have been some efforts to invent a generalized way to achieve this recovery but the proposed solution is prone to numerical instabilities. Here, as presented by Ref. [80], one could use a method called MoP that combines path-based MD with metadynamics since it provides a way to apply metadynamics in a manner that preserves the kinetics of the transition. Further, MoP can also be used to design optimal data-driven CVs for free energy calculations.

To be able to run MoP simulations of real-world systems of scientific interest, it was first necessary to provide a high-performing and efficient implementation of pMD, which samples a Boltzmann-like distribution in the enlarged phase space corresponding to all possible (same-sized) trajectories of the system under BD. The distribution references a potential energy function that effectively describes a fictitious polymer whose beads interact with one another in a harmonic fashion. By evolving the finite-temperature dynamics of this fictitious polymer, the technique generates an entire trajectory in each MD step (as opposed to a configuration in standard MD). This inherently parallel algorithm has the potential to be parallelized very efficiently and can scale to very large node counts.

First, we designed and implemented our pMD algorithm in GROMACS, which is a widely used and highly efficient MD code capable of comprehensive, multi-level parallelism. During initial testing, we identified issues

responsible for increased total energy drift and implemented solutions to mitigate them. Having run correctness tests using several simulation systems, we ascertained the proper functioning of our pMD implementation and highlighted its strengths and limitation. The tests revealed that NVE pMD simulations tend to show a relatively higher total energy drift (compared to standard MD) in single precision. Running the same tests in double precision, we showed that it can be used to obtain very low total energy drift values. Alternatively, it might be necessary to implement a four-point central-differences scheme to reduce the truncation error in the FDM computation, which would accordingly lead to much lower total energy drift values, also in single precision.

After preliminary tests of our pMD implementation for correctness, we conceptualized and implemented interface changes in the PLUMED library to enable the computation of CVs of the fictitious polymer. Accordingly, we made the necessary changes in our pMD implementation within GROMACS and adapted the PLUMED patch of GROMACS to enable metadynamics simulations of the fictitious polymer. After these changes were incorporated, we conducted preliminary tests of the complete MoP machinery.

Next, to improve the hardware resource utilization, we identified the obstacles towards making our pMD implementation compatible with GROMACS' system of DD and designed and implemented solutions as two different algorithms. We benchmarked these algorithms separately on the JUWELS Booster and identified the best-performing algorithm. This algorithm, named the decentralized twin-finding algorithm, shows excellent strong-scaling performance comparable to that of standard MD. Further, we came up with a performance model to compare the runtime of our pMD implementation with that of standard MD, using which we were able to confirm that the former is at least as efficient as the latter at low bead counts.

Moreover, we tested our pMD implementation with force-offloading to GPUs and analyzed its performance, thereby identifying a technique to improve the pMD throughput by increasing GPU utilization. We also conducted several strong-scaling tests of our pMD implementation in the force-offloading mode, which revealed excellent strong-scaling performances at various node-count scales from 8 to 800 nodes of the JUWELS Booster. Finally, we conducted large-scale weak-scaling tests of our implementation in the force-offloading mode with and without PLUMED, going up to 94% of the JUWELS Booster and making use of 3,500 GPUs and 42,000 CPUs. Here, we observed a minimum weak-scaling parallel efficiency of 72% and 70%, respectively, with the weak-scaling curves plateauing at these values at about 50% of the JUWELS Booster.

At this point, the road ahead for our MoP implementation is clear; it is time to test its functionality with regards to its primary objective viz., to study complex biological systems.

Appendix A

Appendix to "Introduction to Parallel Computing"

A.1 TiXL Tutorial

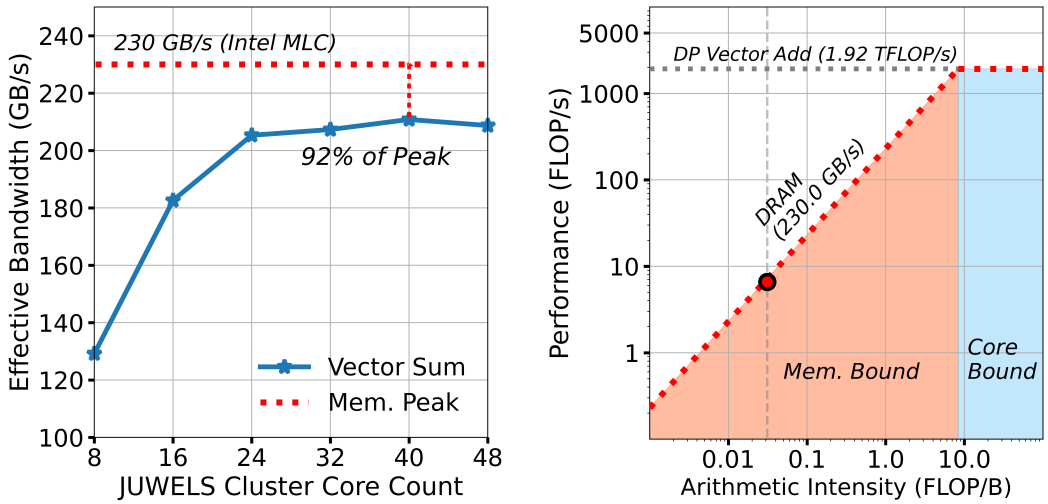


Figure A.1: Figures depicting the performance of the target function `vector_sum`. The left subfigure shows the strong-scaling performance of the function and the right subfigure depicts the location of the function's performance (red circle) in relation to its roofline.

Vector Sum The STREAM benchmark[143] operation that is placed leftmost on the roofline chart i.e., of the other operations, it has the highest code balance. To see how we can compute it, we refer to the core region within the function:

```
for (auto ii = std::size_t{}; ii < n; ++ii)
{
  x_[ii] = y_[ii] + z_[ii]; // fp: 1, traffic: 3+1
}
```

Clearly, it shows us how much computational work is to be done but only *partially* tells us how much data needs to be moved to perform the work. The remaining information must be acquired elsewhere —in the form of a data traffic model. For example, McCalpin uses a code balance model that does not account for write back and arrives at 24 B/FLOP in ref. [143]. Here, we consider cache line write back in our model. To perform each floating-point addition, at least three scalars `x_`, `y_` and `z_` have to be fetched. After the addition, `x_` must be written back to memory. Therefore, the data traffic per index is 4 `double` or 32 B, and the code balance is 32 B/FLOP. The total data traffic is computed over the array size as $32n$ B, using which we can compute the effective bandwidth.

We implemented the microbenchmark using TiXL and optimized the function using OpenMP-based multithreading and vectorization, and ran 50 experiments on a single node of the JUWELS Cluster. To ensure adequate workload, we used an array size amounting to 1 GB of memory. The results are presented in figure A.1, with the left subfigure showing the strong-scaling performance and the right subfigure showing the result of the semi-empirical roofline analysis. From these plots, it's clear that the implementation has come very close to its roofline.

Matrix-Vector Product The widely used function multiplying a matrix with a vector requires $\mathcal{O}(n^2)$ operations and involves the following operation

```
for (auto ii = std::size_t{}; ii < n; ++ii)
{
  for (auto jj = std::size_t{}; jj < n; ++jj)
  {
    // fp: 2, traffic: 3+1
    res[ii] += mat[ii * n + jj] * vec[jj];
  }
}
```

The estimation of data traffic is not as straightforward as in the case of `vector_sum`. Here, the data traffic must be estimated by looking at transfers over the *whole* procedure. To complete $2n^2$ floating-point operations,

1. operand matrix `mat` sized n^2 `double` must be fetched once,
2. operand vector `vec` sized `n double` must be fetched once, and,

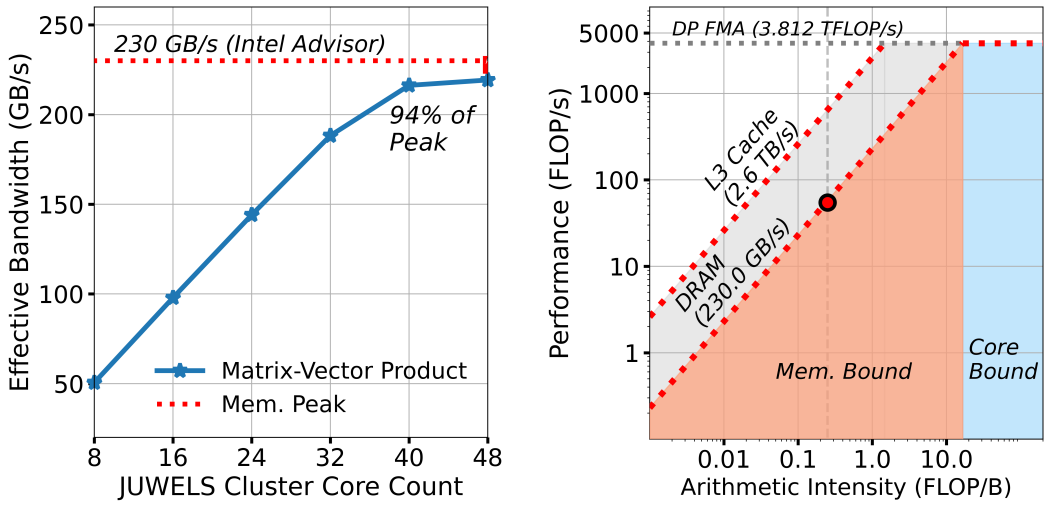


Figure A.2: Figures depicting the performance of the target function `matvec`. The left subfigure shows the strong-scaling performance of the function and the right subfigure depicts the location of the function's performance (red circle) in relation to its roofline.

3. result vector `res` sized `n double` must be fetched once and written back to memory once.

Using this information, the total data traffic is given by $8(\mathtt{n}^2 + 3\mathtt{n})$ GB, which translates to a code balance of $4+\frac{3}{2}\mathtt{n} \sim 4$ B/FLOP. There is some potential for caching as the operand vector is reused $\mathtt{n}$ times. However, the vast majority of the data does not benefit from being cached.

Having implemented this microbenchmark using TiXL and optimized the function using OpenMP-based multithreading and vectorization, we ran 50 experiments on a single node of the JUWELS Cluster using a problem size of $\mathtt{n}$=20000 (i.e., matrix size ~ 3.2 GB). The results are presented in figure A.2, with the left subfigure showing the strong-scaling performance and the right subfigure showing the result of the semi-empirical roofline analysis. From these plots, it's clear that the implementation has come very close to its roofline.

Bidirectional Open Ring Messaging The pattern is a common use case for MPI-parallel programs. It involves only point-to-point communication since each MPI rank sends a message to its left and right neighbours. This pattern is referred to as the "open" ring because the first rank has no left neighbour and the last rank, no right neighbour i.e., the ring is not cyclically closed. As the message passing occurs simultaneously, an implementation of this pattern has the potential to be very efficient.

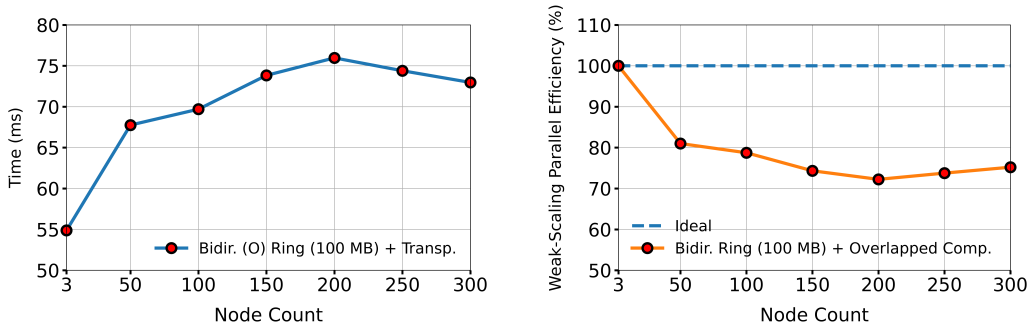


Figure A.3: Figures depicting the weak-scaling performance of the bidirectional open ring messaging program. The left subfigure shows the runtime and the right subfigure depicts the drop in weak-scaling parallel efficiency, followed by a series of plateaus as generalized by figure 2.10.

To demonstrate a pseudo-weak-scaling program, we wrote a microbenchmark using TiXL's MPI-based library that initiates asynchronous receiving and sending of square matrices first, transposes another same-sized matrix and finally waits for the messages to be either sent or to arrive. The (naive) transpose function is meant to represent perfectly scalable work since each rank operates on the same problem size and executes identical instructions. We ran this program on the JUWELS Cluster using up to 300 nodes with results shown in figure A.3.

Appendix B

Appendix to "Metadynamics of Paths"

B.1 Benchmarking pMD with DD (A2A-ZMA)

In subsection 4.3.5, we presented the results of strong-scaling tests of pMD simulations of the system LYSOBENZ (with $\sim 100k$ atoms) using the centralized and the decentralized twin-finding algorithms, to study their performances individually and to compare them. For completeness, we repeated these tests for a different system A2A-ZMA (with $\sim 150k$ atoms), the results of which are presented here. The tests were conducted with identical input options, simulation parameters and setup but with one change: the FDM stabilization was enabled with maximum frequency i.e. it was performed in every time step.

As seen in figures B.1, the results are almost identical to those obtained for the system LYSOBENZ. Similarly, figure B.2 shows the reduction in energy consumption as the number of MPI ranks dedicated to DD is increased. Here, we see a total reduction by a factor of 25.

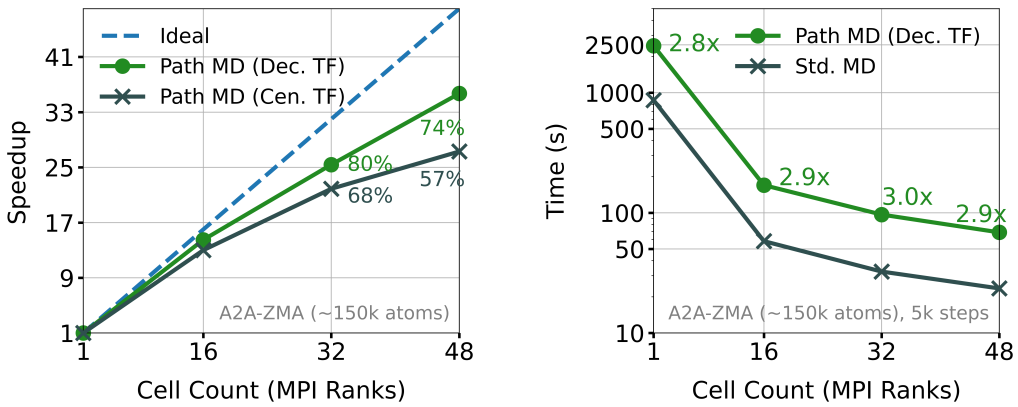


Figure B.1: Left subfigure shows the strong-scaling performances of pMD simulations using DD via the centralized and the decentralized twin-finding algorithms. Right subfigure shows the runtime factor-of-three of pMD simulations using DD via the decentralized twin-finding algorithm with respect to their analogous standard MD simulations. In both subfigures, the simulations are of the system A2A-ZMA (with $\sim 150k$ atoms) with FDM stabilization being performed in every time step.

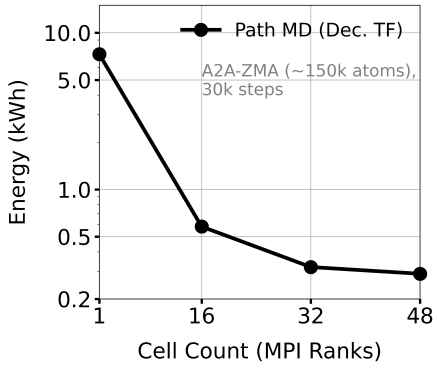


Figure B.2: A plot relating the energy consumption of pMD simulations of the system A2A-ZMA (with $\sim 150k$ atoms) with three beads for 30,000 time steps to the number of MPI ranks given to DD (via the decentralized twin-finding algorithm). The energy consumption dropped by a factor of 24 from when DD was disabled to when it was enabled with as many MPI ranks as cores in one JUWELS Booster node.

B.2 PLUMED Input for Weak-Scaling Tests

Here, we present the PLUMED input file that was used to conduct the large-scale weak-scaling tests on the JUWELS Booster as reported in subsection 4.3.7. In it, we define an end-to-end distance of the fictitious polymer for various bead counts. In each file, the placeholder 'XXX' must be replaced by the value $N - 1$, where N is the bead count.

```
# ------------ plumed.dat ------------
#RESTART

# PLUMED descriptors
# Compute COM and distance between atoms of the
ligand (ZMA) and the binding pocket (cBS). Atoms are
ordered as in the trajectory files and their
numbering starts from 1.
ZMA: COM
ATOMS=5227,5259,5231,5232,5234,5236,5238,5239,5242,5245,5247,
5248,5249,5250,5253,5254,5255,5256,5257,5258,5259,5260,5262,
5264,5266
cBS: COM
ATOMS=121,951,994,1013,1238,1254,2525,2545,2588,2618,2670,
3862,3936, 3981,4081,4092,4196,4263,4328
Dcom: DISTANCE ATOMS=ZMA,cBS

# end-to-end distance CV
CUSTOM ...
  LABEL=CV
  ARG=Dcom
  VAR=cv
  FUNC=(cvXXX-cv0)
  MULTIREP
  REPLICAS=0,XXX
  PERIODIC=NO
... CUSTOM

# Apply METAD bias
metad: METAD ARG=CV TEMP=310 PACE=500 HEIGHT=1.2
BIASFACTOR=15 SIGMA=0.05

# Print on a file named "COLVAR" every k steps.
PRINT ARG=* FILE=Colvar FMT=%20.15E STRIDE=500
```

Appendix C

Appendix to ”Scalability of 3D-DFT by Block Tensor-Matrix Multiplication on the JUWELS Cluster”

C.1 Performance Analysis of the Transpose Function

”Here, we take a closer look at the performance of our implementation of the local transpose function, τ of equation (5.7). The transpose function can be viewed as a streaming function because in a perfect implementation it would closely resemble a copy operation. Thus, one way to assess the performance of our implementation is to compare it to that of a suitably similar streaming function having an excellent memory bandwidth utilization. As a reference, we decided to use the DAXPY loop, by measuring the performance of the operation $Y = Y + aX$ on the target computer, where $Y, X \in \mathbb{R}^N$ are double-precision arrays and $a \in \mathbb{R}$ is a double-precision scalar. We performed the microbenchmarking with $N = 720$, at which size the performance of the loop was found to saturate. The black lines in figure C.1 illustrate the increase of the effective bandwidth of this reference loop as a function of the number of cores, for the single as well as dual NUMA configurations. The recorded corresponding peak performances are 103 GB/s and 202 GB/s, respectively. We used these values as reference to measure the efficiency of our implementation of the transpose function.

The performance of the naïve implementation of the transpose function is shown by the red curves in figure C.1 for the size $N = 720$. Building on

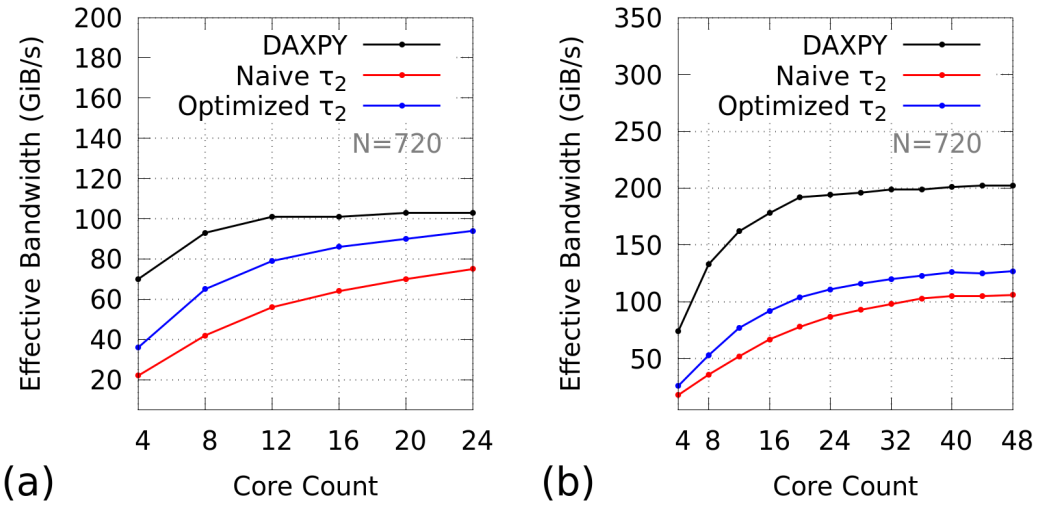


Figure C.1: "Subfigures show (a) strong scaling in single NUMA configuration and (b) strong scaling in dual NUMA configuration. In both subfigures, $N = 720$" (A.).

it, we improved the cache utilization by applying loop-blocking with the help of an intermediate array so small as to fit into the cache. The optimal blocking size was experimentally found to be 36 kiB. Upon optimization, only a small improvement in performance could be observed, as shown by the blue curves in figure C.1. Indeed, we found the performance of the näive implementation to be quite high, which we attributed to the size of the processor's L3-cache, by virtue of which good cache-line reuse can be achieved even for relatively large matrix sizes.

In the single NUMA configuration, the optimized transpose function attained a peak efficiency of 91% as compared to that of 73% of the naive function. However, we note that the efficiencies drop to 63% and 53% respectively, when the dual NUMA configuration is applied. This can be attributed to unavoidable non-local memory accesses arising from the fact that the functions make use of a different multithreading work-sharing plan as compared to the other kernels in the implementation. Although an adaptation of the algorithm to minimize these non-local memory accesses is conceivable, the expected performance gain did not justify its design and implementation within the scope of this study" (A.).

C.2 Comparison with FFTK

"Figures C.2 and C.3 show the benchmarking results of S3DFT (using the global transpose variants described in subsection 5.2.3) and FFTK. In the

small problem scale, with S3DFT in the 2 MPI tasks/node configuration, on average, FFTK was 3.8 times faster than S3DFT. In the large problem scale, with S3DFT in the 1 MPI task/node configuration, on average, FFTK was 4.3 times faster than S3DFT" (A.).

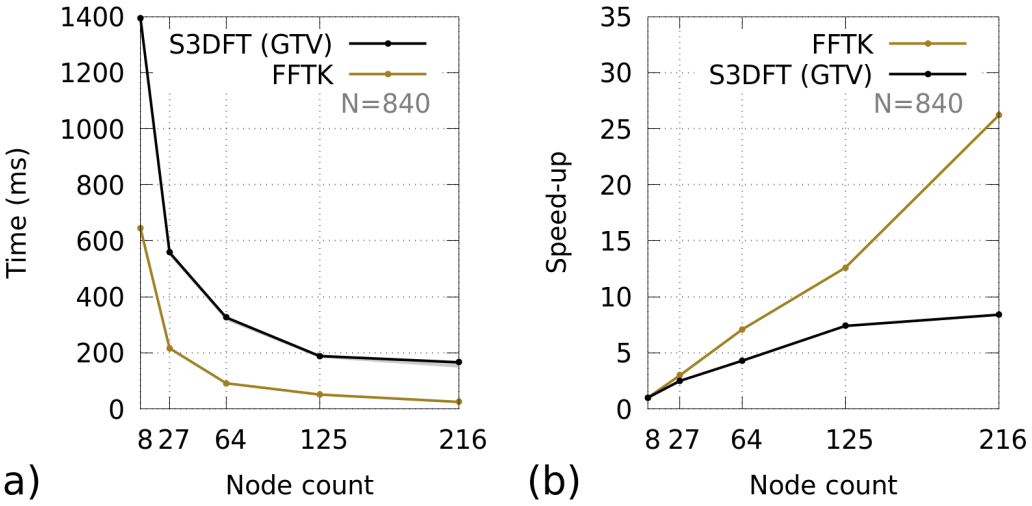


Figure C.2: "In this comparison, S3DFT uses the global transpose variants as detailed in subsection 5.2.3. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 840$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 2 MPI tasks/node" (A.).

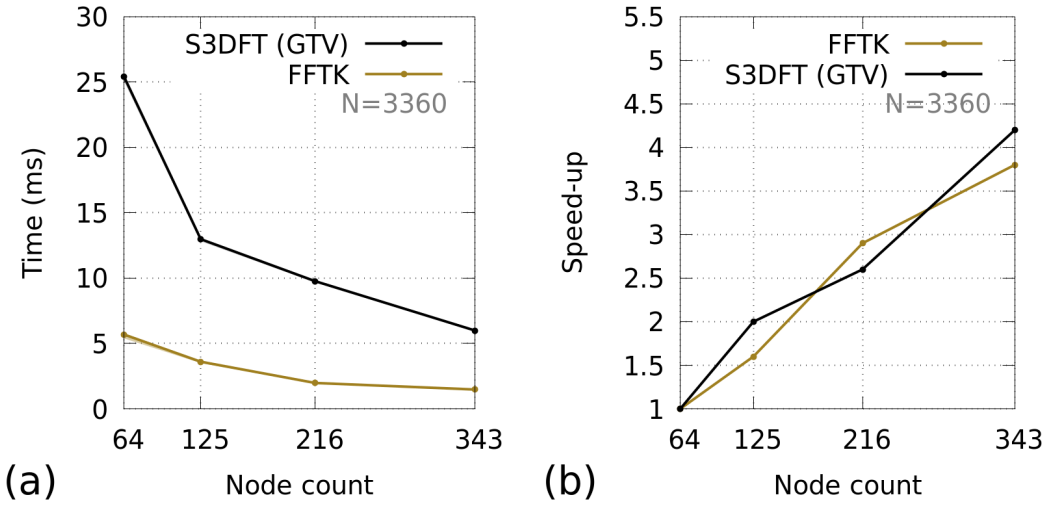


Figure C.3: "In this comparison, S3DFT uses the global transpose variants as detailed in subsection 5.2.3. Subfigures show (a) the time-to-solution and (b) the speed-up for $N = 3360$. In subfigure (a), the lower border of the shading underneath the lines indicates the minimum time. Here, S3DFT was run using 1 MPI task/node" (A.).

Bibliography

[1] Daan Frenkel and Berend Smit. Chapter 1 - introduction. In Daan Frenkel and Berend Smit, editors, *Understanding Molecular Simulation (Second Edition)*, pages 1–6. Academic Press, San Diego, second edition edition, 2002.

[2] Ben Cooke and Scott C. Schmidler. Statistical prediction and molecular dynamics simulation. *Biophysical Journal*, 95(10):4497–4511, Nov 2008.

[3] Arieh Warshel. Molecular dynamics simulations of biological reactions. *Accounts of Chemical Research*, 35(6):385–395, Jun 2002.

[4] Wei Ye, Wei Wang, Cheng Jiang, Qingfen Yu, and Haifeng Chen. Molecular dynamics simulations of amyloid fibrils: an ¡italic¿in silico¡/italic¿ approach. *Acta Biochimica et Biophysica Sinica*, 45(6):503–508, 2013.

[5] Maricarmen Hernández-Rodríguez, Martha C. Rosales-Hernández, Jessica E. Mendieta-Wejebe, Marlet Martínez-Archundia, and José Correa Basurto. Current tools and methods in molecular dynamics (md) simulations for drug design. *Current Medicinal Chemistry*, 23(34):3909–3924, 2016.

[6] Xuewei Liu, Danfeng Shi, Shuangyan Zhou, Hongli Liu, Huanxiang Liu, and Xiaojun Yao. Molecular dynamics simulations and novel drug discovery. *Expert Opin. Drug Discov.*, 13(1):23–37, January 2018.

[7] Marco De Vivo, Matteo Masetti, Giovanni Bottegoni, and Andrea Cavalli. Role of molecular dynamics and related methods in drug discovery. *Journal of Medicinal Chemistry*, 59(9):4035–4061, May 2016.

[8] Viacheslav Bolnykh. *Massively parallel quantum mechanical/molecular mechanical interface*. Dissertation, RWTH Aachen University, Aachen, 2019. Cotutelle-Dissertation. - Veröffentlicht auf dem Publikationsserver der RWTH Aachen University 2020; Dissertation, RWTH Aachen University, 2019. - Dissertation, The Cyprus Institue, 2019.

[9] Pratyush Tiwary, Vittorio Limongelli, Matteo Salvalaglio, and Michele Parrinello. Kinetics of protein–ligand unbinding: Predicting pathways, rates, and rate-limiting steps. *Proceedings of the National Academy of Sciences*, 112(5):E386–E391, 2015.

[10] Eric Vanden-Eijnden, Maddalena Venturoli, Giovanni Ciccotti, and Ron Elber. On the assumptions underlying milestoning. *The Journal of Chemical Physics*, 129(17):174102, 11 2008.

[11] P. G. Bolhuis and C. Dellago. Practical and conceptual path sampling issues. *The European Physical Journal Special Topics*, 224(12):2409–2427, Sep 2015.

[12] Dhiman Ray and Michele Parrinello. Kinetics from metadynamics: Principles, applications, and outlook. *Journal of Chemical Theory and Computation*, 19(17):5649–5670, Sep 2023.

[13] Georg Hager and Gerhard Wellein. *Introduction to High Peformance Computing for Scientists and Engineers*. CRC Press, 07 2010.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[15] Victor Eijkhout, Robert van de Geijn, and Edmond Chow. *Introduction to High Performance Scientific Computing*. Lulu.com, 01 2016.

[16] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[17] M. Dawson, S. Lakshmanamurthy, D. Bakhvalov, and N. Rotem. *Performance Analysis and Tuning on Modern CPUs: Squeeze the Last Bit of Performance from Your Application.* Amazon Digital Services LLC - Kdp, 2020.

[18] Intel Corp. Intel® celeron® processor g1820te. `https://www.intel.com/content/www/us/en/products/sku/78957/intel-celeron-processor-g1820te-2m-cache-2-20-ghz/specifications.html`, 2025. Last accessed: 24.02.2025.

[19] Intel Corp. Intel® core™ i7-5960x processor extreme edition. `https://www.intel.com/content/www/us/en/products/sku/82930/intel-core-i75960x-processor-extreme-edition-20m-cache-up-to-3-50-ghz/specifications.html`, 2025. Last accessed: 24.02.2025.

[20] Intel Corp. *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, 2024. Last accessed: 24.02.2025. Avail-

able at https://cdrdv2-public.intel.com/814198/248966-Optimization-Reference-Manual-V1-050.pdf.

[21] P. Gepner and M.F. Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13, 2006.

[22] Richard Vuduc, Jee Choi, Xuan Shi, Volodymyr Kindratenko, and Chaowei Yang. *A Brief History and Introduction to GPGPU*, pages 9–23. Springer US, Boston, MA, 2013.

[23] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

[24] Cristobal Navarro, Nancy Hitschfeld, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using gpu architectures. *Communications in Computational Physics*, 15:285–329, 09 2013.

[25] Ogier Maitre. *Understanding NVIDIA GPGPU hardware*, pages 15–34. Natural Computing Series, 07 2013.

[26] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[27] Dong-Joon Lim, Timothy R. Anderson, and Tom Shott. Technological forecasting of supercomputer development: The march to exascale computing. *Omega*, 51:128–135, 2015.

[28] Susanne Kunkel, Maximilian Schmidt, Jochen M. Eppler, Hans E. Plesser, Gen Masumoto, Jun Igarashi, Shin Ishii, Tomoki Fukai, Abigail Morrison, Markus Diesmann, and Moritz Helias. Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics*, 8, 2014.

[29] Stefan Kesselheim, Andreas Herten, Kai Krajsek, Jan Ebert, Jenia Jitsev, Mehdi Cherti, Michael Langguth, Bing Gong, Scarlet Stadtler, Amirpasha Mozaffari, Gabriele Cavallaro, Rocco Sedona, Alexander Schug, Alexandre Strube, Roshni Kamath, Martin G. Schultz, Morris Riedel, and Thomas Lippert. Juwels booster – a supercomputer for large-scale ai research. In Heike Jagode, Hartwig Anzt, Hatem Ltaief, and Piotr Luszczek, editors, *High Performance Computing*, pages 453–468, Cham, 2021. Springer International Publishing.

[30] Top500.org. Performance development. https://top500.org/statistics/perfdevel/, 2025. Last accessed: 26.02.2025.

[31] Top500.org. El capitan - hpe cray ex255a, amd 4th gen epyc 24c 1.8ghz, amd instinct mi300a, slingshot-11, toss. https://top500.org/statistics/perfdevel/, 2025. Last accessed: 26.02.2025.

[32] Damian Alvarez. Juwels cluster and booster: Exascale pathfinder with modular supercomputing architecture at juelich supercomputing centre. *Journal of large-scale research facilities JLSRF*, 7, 10 2021.

[33] Top500.org. Juwels module 1 - bull sequana x1000, xeon platinum 8168 24c 2.7ghz, mellanox edr infiniband/partec parastation clustersuite. https://top500.org/system/179424/, 2025. Last accessed: 25.02.2025.

[34] Top500.org. Juwels booster module - bull sequana xh2000 , amd epyc 7402 24c 2.8ghz, nvidia a100, mellanox hdr infiniband/partec parastation clustersuite. https://top500.org/system/179894/, 2025. Last accessed: 25.02.2025.

[35] Brian Lebiednik, Aman Mangal, and Niharika Tiwari. A survey and evaluation of data center network topologies, 2016.

[36] David Callahan, John Cocke, and Ken Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5:334–358, 05 2001.

[37] Nitin Malapally. Tixl: Timed experiments in a loop. Available on https://gitlab.com/anxiousprogrammer/tixl, 2021. Last accessed: 08.11.2022.

[38] M. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford Graduate Texts. OUP Oxford, 2010.

[39] GROMACS development team. *GROMACS Manual*, 2024. Last accessed: 24.01.2025. Available at https://manual.gromacs.org/2024.0/reference-manual/algorithms/replica-exchange.html.

[40] Q Spreiter and M Walter. Classical molecular dynamics simulation with the velocity verlet algorithm at strong external magnetic fields. *Journal of Computational Physics*, 152(1):102–119, 1999.

[41] I. P. Omelyan, I. M. Mryglod, and R. Folk. Optimized verlet-like algorithms for molecular dynamics simulations. *Phys. Rev. E*, 65:056706, May 2002.

[42] Şakir Erkoç. Empirical many-body potential energy functions used in computer simulations of condensed matter properties. *Physics Reports*, 278(2):79–105, 1997.

[43] K. Vanommeslaeghe and A.D. MacKerell. Charmm additive and polarizable force fields for biophysics and computer-aided drug design. *Biochimica et Biophysica Acta (BBA) - General Subjects*, 1850(5):861–871, 2015. Recent developments of molecular dynamics.

[44] Wei Kang, Fan Jiang, and Yun-Dong Wu. How to strike a conformational balance in protein force fields for molecular dynamics simulations? *WIREs Computational Molecular Science*, 12(3):e1578, 2022.

[45] Ming-Hong Hao and Harold A Scheragat. Designing potential energy functions for protein folding. *Current Opinion in Structural Biology*, 9(2):184–188, 1999.

[46] Benjamin Waldher, Jadwiga Kuta, Samuel Chen, Neil Henson, and Aurora E. Clark. Forcefit: A code to fit classical force fields to quantum mechanical potential energy surfaces. *Journal of Computational Chemistry*, 31(12):2307–2316, 2010.

[47] Pedro E M Lopes, Olgun Guvench, and Alexander D MacKerell, Jr. Current status of protein force fields for molecular dynamics simulations. *Methods Mol. Biol.*, 1215:47–71, 2015.

[48] Oliver T. Unke, Martin Stöhr, Stefan Ganscha, Thomas Unterthiner, Hartmut Maennel, Sergii Kashubin, Daniel Ahlin, Michael Gastegger, Leonardo Medrano Sandonas, Joshua T. Berryman, Alexandre Tkatchenko, and Klaus-Robert Müller. Biomolecular dynamics with machine-learned quantum-mechanical force fields trained on diverse chemical fragments. *Science Advances*, 10(14):eadn4397, 2024.

[49] Pan Zhang and Weitao Yang. Toward a general neural network force field for protein simulations: Refining the intramolecular interaction in protein. *J. Chem. Phys.*, 159(2), July 2023.

[50] Huziel E. Sauceda, Luis E. Gálvez-González, Stefan Chmiela, Lauro Oliver Paz-Borbón, Klaus-Robert Müller, and Alexandre Tkatchenko. Bigdml—towards accurate quantum machine learning force fields for materials. *Nature Communications*, 13(1):3733, Jun 2022.

[51] Pnina Dauber-Osguthorpe, Victoria A. Roberts, David J. Osguthorpe, Jon Wolff, Moniqe Genest, and Arnold T. Hagler. Structure and energetics of ligand binding to proteins: Escherichia coli dihydrofolate reductase-trimethoprim, a drug-receptor system. *Proteins: Structure, Function, and Bioinformatics*, 4(1):31–47, 1988.

[52] Chandler A. Becker, Francesca Tavazza, Zachary T. Trautt, and Robert A. Buarque de Macedo. Considerations for choosing and using force fields and interatomic potentials in materials science and engi-

neering. *Current Opinion in Solid State and Materials Science*, 17(6):277–283, 2013. Frontiers in Methods for Materials Simulations.

[53] Hendrik Heinz and Hadi Ramezani-Dakhel. Simulations of inorganic–bioorganic interfaces to discover new materials: insights, comparisons to experiment, challenges, and opportunities. *Chem. Soc. Rev.*, 45:412–448, 2016.

[54] Philip M. Morse. Diatomic molecules according to the wave mechanics. ii. vibrational levels. *Phys. Rev.*, 34:57–64, Jul 1929.

[55] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, and J. Hermans. *Interaction Models for Water in Relation to Protein Hydration*, pages 331–342. Springer Netherlands, Dordrecht, 1981.

[56] David M. Ferguson. Parameterization and evaluation of a flexible water model. *Journal of Computational Chemistry*, 16(4):501–511, 1995.

[57] Bernard R. Brooks, Robert E. Bruccoleri, Barry D. Olafson, David J. States, S. Swaminathan, and Martin Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[58] Teik-Cheng Lim. The relationship between lennard-jones (12-6) and morse potential functions. *Zeitschrift für Naturforschung A*, 58(11):615–617, 2003.

[59] González, M.A. Force fields and molecular dynamics simulations. *JDN*, 12:169–200, 2011.

[60] C. Pangali, M. Rao, and B.J. Berne. A monte carlo study of structural and thermodynamic properties of water: dependence on the system size and on the boundary conditions. *Molecular Physics*, 40(3):661–680, 1980.

[61] Abdulnour Y. Toukmaji and John A. Board. Ewald summation techniques in perspective: a survey. *Computer Physics Communications*, 95(2):73–92, 1996.

[62] Teodoro Laino, Fawzi Mohamed, Alessandro Laio, and Michele Parrinello. An efficient linear-scaling electrostatic coupling for treating periodic boundary conditions in qm/mm simulations. *Journal of Chemical Theory and Computation*, 2(5):1370–1378, 2006. PMID: 26626844.

[63] Scott E. Feller, Richard W. Pastor, Atipat Rojnuckarin, Stephen Bogusz, and Bernard R. Brooks. Effect of electrostatic force truncation on interfacial and transport properties of water. *The Journal of Physical Chemistry*, 100(42):17011–17020, 1996.

[64] Axel Arnold, Florian Fahrenberger, Christian Holm, Olaf Lenz, Matthias Bolten, Holger Dachsel, Rene Halver, Ivo Kabadshow, Franz Gähler, Frederik Heber, Julian Iseringhausen, Michael Hofmann, Michael Pippig, Daniel Potts, and Godehard Sutmann. Comparison of scalable fast methods for long-range interactions. *Phys. Rev. E*, 88:063308, Dec 2013.

[65] M. J. Harvey and G. De Fabritiis. An implementation of the smooth particle mesh ewald method on gpu hardware. *Journal of Chemical Theory and Computation*, 5(9):2371–2377, 2009. PMID: 26616618.

[66] Franziska Nestler. Fast ewald summation for electrostatic systems with charges and dipoles for various types of periodic boundary conditions. In *2017 International Conference on Sampling Theory and Applications (SampTA)*, pages 465–469, 2017.

[67] Hark B. Lee and Wei Cai. Ewald summation for coulomb interactions in a periodic supercell, 2009.

[68] Dominik Marx and Jürg Hutter. *Ab Initio Molecular Dynamics: Basic Theory and Advanced Methods*. Cambridge University Press, 2009.

[69] Frank Jensen. *Introduction to Computational Chemistry*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2006.

[70] Max Born. *Statistical Interpretation of Quantum Mechanics*, pages 89–99. Springer New York, New York, NY, 1968.

[71] Klaus Capelle. A bird's-eye view of density-functional theory. *Braz. J. Phys.*, 36, 12 2002.

[72] Eberhard Engel and Reiner M Dreizler. *Density functional theory*. Theoretical and mathematical physics. Springer, Berlin, Germany, 2011 edition, February 2011.

[73] Bharath Raghavan, Mirko Paulikat, Katya Ahmad, Lara Callea, Andrea Rizzi, Emiliano Ippoliti, Davide Mandelli, Laura Bonati, Marco De Vivo, and Paolo Carloni. Drug Design in the Exascale Era: A Perspective from Massively Parallel QM/MM Simulations. *Journal of Chemical Information and Modeling*, 63(12):3647–3658, jun 2023.

[74] P.G. Khalatur. 1.16 - molecular dynamics simulations in polymer science: Methods and main results. In Krzysztof Matyjaszewski and Martin Möller, editors, *Polymer Science: A Comprehensive Reference*, pages 417–460. Elsevier, Amsterdam, 2012.

[75] Bharath Raghavan. *High performance computing-based QM/MM simulations for drug design: application to the non-invasive diagnosis of IDH1-associated glioma*. Dissertation, RWTH Aachen University, Aachen,

2024. Veröffentlicht auf dem Publikationsserver der RWTH Aachen University; Dissertation, RWTH Aachen University, 2024.

[76] Lili Cao and Ulf Ryde. On the difference between additive and subtractive QM/MM calculations. *Front. Chem.*, 6, April 2018.

[77] Gerrit Groenhof. Introduction to QM/MM simulations. In *Methods in Molecular Biology*, Methods in molecular biology (Clifton, N.J.), pages 43–66. Humana Press, Totowa, NJ, 2013.

[78] Christopher I. Bayly, Piotr Cieplak, Wendy Cornell, and Peter A. Kollman. A well-behaved electrostatic potential based method using charge restraints for deriving atomic charges: the resp model. *The Journal of Physical Chemistry*, 97(40):10269–10280, Oct 1993.

[79] David Daniel Girardier, Hadrien Vroylandt, Sara Bonella, and Fabio Pietrucci. Inferring free-energy barriers and kinetic rates from molecular dynamics via underdamped langevin models. *The Journal of Chemical Physics*, 159(16):164111, 10 2023.

[80] Davide Mandelli, Barak Hirshberg, and Michele Parrinello. Metadynamics of paths. *Phys. Rev. Lett.*, 125:026001, Jul 2020.

[81] Omar Valsson, Pratyush Tiwary, and Michele Parrinello. Enhancing important fluctuations: Rare events and metadynamics from a conceptual viewpoint. *Annual Review of Physical Chemistry*, 67(Volume 67, 2016):159–184, 2016.

[82] Yi Isaac Yang, Qiang Shao, Jun Zhang, Lijiang Yang, and Yi Qin Gao. Enhanced sampling in molecular dynamics. *The Journal of Chemical Physics*, 151(7):070902, 08 2019.

[83] L. Donati, C. Hartmann, and B. G. Keller. Girsanov reweighting for path ensembles and markov state models. *The Journal of Chemical Physics*, 146(24):244112, 06 2017.

[84] Hendrik Jung, Kei-ichi Okazaki, and Gerhard Hummer. Transition path sampling of rare events by shooting from the top. *The Journal of Chemical Physics*, 147(15):152716, 08 2017.

[85] M. Parrinello and A. Rahman. Study of an f center in molten kcl. *The Journal of Chemical Physics*, 80(2):860–867, 01 1984.

[86] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. C. Berendsen. Gromacs: Fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, 2005.

[87] Massimiliano Bonomi, Davide Branduardi, Giovanni Bussi, Carlo Camilloni, Davide Provasi, Paolo Raiteri, Davide Donadio, Fabrizio

Marinelli, Fabio Pietrucci, Ricardo A. Broglia, and Michele Parrinello. Plumed: A portable plugin for free-energy calculations with molecular dynamics. *Computer Physics Communications*, 180(10):1961–1972, 2009.

[88] Anna Putrino, Daniel Sebastiani, and Michele Parrinello. Generalized variational density functional perturbation theory. *The Journal of Chemical Physics*, 113(17):7102–7109, 11 2000.

[89] Christoph Dellago, Peter G. Bolhuis, Félix S. Csajka, and David Chandler. Transition path sampling and the calculation of rate constants. *The Journal of Chemical Physics*, 108(5):1964–1977, 02 1998.

[90] Alessandro Laio and Michele Parrinello. Escaping free-energy minima. *Proceedings of the National Academy of Sciences*, 99(20):12562–12566, 2002.

[91] Giovanni Bussi, Alessandro Laio, and Michele Parrinello. Equilibrium free energies from nonequilibrium metadynamics. *Phys. Rev. Lett.*, 96:090601, Mar 2006.

[92] Alessandro Barducci, Giovanni Bussi, and Michele Parrinello. Well-tempered metadynamics: A smoothly converging and tunable free-energy method. *Phys. Rev. Lett.*, 100:020603, Jan 2008.

[93] James F. Dama, Michele Parrinello, and Gregory A. Voth. Well-tempered metadynamics converges asymptotically. *Phys. Rev. Lett.*, 112:240602, Jun 2014.

[94] M Bonomi, A Barducci, and M Parrinello. Reconstructing the equilibrium boltzmann distribution from well-tempered metadynamics. *J Comput Chem*, 30(11):1615–1621, August 2009.

[95] Davide Branduardi, Giovanni Bussi, and Michele Parrinello. Metadynamics with adaptive gaussians. *Journal of Chemical Theory and Computation*, 8(7):2247–2254, Jul 2012.

[96] Pratyush Tiwary and Michele Parrinello. A time-independent free energy estimator for metadynamics. *The Journal of Physical Chemistry B*, 119(3):736–742, Jan 2015.

[97] Letif Mones, Noam Bernstein, and Gábor Csányi. Exploration, sampling, and reconstruction of free energy surfaces with gaussian process regression. *J Chem Theory Comput*, 12(10):5100–5110, September 2016.

[98] Veselina Marinova and Matteo Salvalaglio. Time-independent free energies from metadynamics via mean force integration. *The Journal of Chemical Physics*, 151(16):164115, 10 2019.

[99] Michele Invernizzi and Michele Parrinello. Rethinking metadynamics: From bias potentials to probability distributions. *The Journal of Physical Chemistry Letters*, 11(7):2731–2736, Apr 2020.

[100] F. Giberti, B. Cheng, G. A. Tribello, and M. Ceriotti. Iterative unbiasing of quasi-equilibrium sampling. *Journal of Chemical Theory and Computation*, 16(1):100–107, Jan 2020.

[101] Lukas Müllender, Andrea Rizzi, Michele Parrinello, Paolo Carloni, and Davide Mandelli. Effective data-driven collective variables for free energy calculations from metadynamics of paths. *PNAS Nexus*, 3(4):pgae159, 04 2024.

[102] Aidan P. Thompson, H. Metin Aktulga, Richard Berger, Dan S. Bolintineanu, W. Michael Brown, Paul S. Crozier, Pieter J. in 't Veld, Axel Kohlmeyer, Stan G. Moore, Trung Dac Nguyen, Ray Shan, Mark J. Stevens, Julien Tranchida, Christian Trott, and Steven J. Plimpton. Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications*, 271:108171, 2022.

[103] Alan Gray and Szilárd Páll. Maximizing gromacs throughput with multiple simulations per gpu using mps and mig. `https://developer.nvidia.com/blog/maximizing-gromacs-throughput-with-multiple-simulations-per-gpu-using-mps-and-mig/`, 2021. Last accessed: 20.01.2025.

[104] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19–25, 2015.

[105] Carsten Kutzner, David van der Spoel, Martin Fechner, Erik Lindahl, Udo Schmitt, Bert de Groot, and Helmut Grubmüller. Speeding up parallel gromacs on high-latency networks. *Journal of computational chemistry*, 28:2075–84, 09 2007.

[106] Berk Hess, C. Kutzer, David van der Spoel, and Erik Lindahl. Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J Chem Theory Comput*, 4:435–447, 03 2008.

[107] Szilárd Páll, Artem Zhmurov, Paul Bauer, Mark Abraham, Magnus Lundborg, Alan Gray, Berk Hess, and Erik Lindahl. Heterogeneous parallelization and acceleration of molecular dynamics simulations in gromacs. *The Journal of Chemical Physics*, 153(13):134110, 10 2020.

[108] Alan Gray and Szilárd Páll. Maximizing gromacs throughput with multiple simulations per gpu using mps and mig.

https://www.plumed.org/doc-v2.9/user-doc/html/_installation.html. Last accessed: 20.01.2025.

[109] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, USA, 3 edition, 2007.

[110] Venkat Kapil, Jörg Behler, and Michele Ceriotti. High order path integrals made easy. *The Journal of Chemical Physics*, 145(23):234103, 12 2016.

[111] Wan Rong-Zheng, Li Song-Yan, and Fang Hai-Ping. Effect of center-of-mass motion removal in the molecular dynamics simulations. *Chinese Physics Letters*, 27(8):084702, aug 2010.

[112] Stephen C. Harvey, Robert K.-Z. Tan, and Thomas E. Cheatham III. The flying ice cube: Velocity rescaling in molecular dynamics leads to violation of energy equipartition. *Journal of Computational Chemistry*, 19(7):726–740, 1998.

[113] Alan Gray. Creating faster molecular dynamics simulations with gromacs 2020. https://developer.nvidia.com/blog/creating-faster-molecular-dynamics-simulations-with-gromacs-2020/, 2020. Last accessed: 28.03.2025.

[114] Andrea Levy, Sophia Johnson, Bharath Raghavan, Sonata Kvedaravičiūtė, and David Carrasco de Busturia. First steps: Acetone in water. https://mimic-project.org/en/latest/tutorials/acetone_tutorial.html. Last accessed: 24.03.2025.

[115] Riccardo Capelli, Paolo Carloni, and Michele Parrinello. Exhaustive search of ligand binding pathways via volume-based metadynamics. *The Journal of Physical Chemistry Letters*, 10(12):3495–3499, Jun 2019.

[116] Ruyin Cao, Alejandro Giorgetti, Andreas Bauer, Bernd Neumaier, Giulia Rossetti, and Paolo Carloni. Role of extracellular loops and membrane lipids for ligand recognition in the neuronal adenosine receptor type 2a: An enhanced sampling simulation study. *Molecules*, 23(10):2616, October 2018.

[117] Vitor Silva and Filipe Guimaraes. HPC system and job monitoring with LLview. In *RISC2 webinar series*. RISC2 webinar series, Online (Germany), 2022.

[118] Alessandro Campa, Thierry Dauxois, and Stefano Ruffo. Statistical mechanics and dynamics of solvable models with long-range interactions. *Physics Reports*, 480(3):57–159, 2009.

[119] Handan Yildirim, Jeronimo Matos, and Abdelkader Kara. Role of long-range interactions for the structure and energetics of olympicene radical adsorbed on au(111) and pt(111) surfaces. *The Journal of Physical Chemistry C*, 119(45):25408–25419, 2015.

[120] Bartosz Kohnke, Carsten Kutzner, and Helmut Grubmüller. A gpu-accelerated fast multipole method for gromacs: Performance and accuracy. *Journal of Chemical Theory and Computation*, 16(11):6938–6949, 2020. PMID: 33084336.

[121] Valery Weber, Costas Bekas, Teodoro Laino, Alessandro Curioni, Adam Bertsch, and Scott Futral. Shedding Light on Lithium/Air Batteries Using Millions of Threads on the BG/Q Supercomputer. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 735–744. IEEE, may 2014.

[122] Ariana Remmel. What exascale computing could mean for chemistry. *Chemical and Engineering News*, pages 29–33, sep 2022.

[123] D.N. Rockmore. The fft: an algorithm the whole family can use. *Computing in Science Engineering*, 2(1):60–64, 2000.

[124] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[125] Dmitry Pekurovsky. Ultrascalable fourier transfroms in three dimensions. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, New York, NY, USA, 2011. Association for Computing Machinery.

[126] Alan Ayala, Stanimire Tomov, Miroslav Stoyanov, and Jack Dongarra. Scalability issues in fft computation. In Victor Malyshkin, editor, *Parallel Computing Technologies*, pages 279–287, Cham, 2021. Springer International Publishing.

[127] Thomas Lippert, Klaus Schilling, Sven Trentmann, Federico Toschi, and Raffaele Tripiccione. Fft for the ape parallel computer. *International Journal of Modern Physics C*, 8(06):1317–1334, 1997.

[128] Anando G. Chatterjee, Mahendra K. Verma, Abhishek Kumar, Ravi Samtaney, Bilel Hadri, and Rooh Khurram. Scaling of a Fast Fourier Transform and a pseudo-spectral fluid solver up to 196608 cores. *Journal of Parallel and Distributed Computing*, 113:77–91, mar 2018.

[129] Jaewoon Jung, Chigusa Kobayashi, Toshiyuki Imamura, and Yuji Sugita. Parallel implementation of 3d fft with volumetric decomposition schemes for efficient molecular dynamics simulations. *Computer Physics Communications*, 200:57–65, 2016.

[130] Stanislav Sedukhin, T. Sakai, and Naohito Nakasato. 3d discrete transforms with cubical data decomposition on the ibm blue gene/q. *Proceedings of the 30th International Conference on Computers and Their Applications, CATA 2015*, pages 193–200, 01 2015.

[131] Tobias Klöffel, Gerald Mathias, and Bernd Meyer. Integrating state of the art compute, communication, and autotuning strategies to multiply the performance of the application programm cpmd for ab initio molecular dynamics simulations, 2020.

[132] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, USA, 1969. AAI7010025.

[133] Anshul Gupta and Vipin Kumar. Scalability of parallel algorithms for matrix multiplication. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 3, pages 115–123, 1993.

[134] Jean-Noël Quintin, Khalid Hasanov, and Alexey Lastovetsky. Hierarchical parallel matrix multiplication on large-scale distributed memory platforms. In *2013 42nd International Conference on Parallel Processing*, pages 754–762, 2013.

[135] Tamara Kolda and Brett Bader. Tensor decompositions and applications. *SIAM Review*, 51:455–500, 08 2009.

[136] Nitin Malapally. S3dft: Scalable 3d-dft. Available on `https://gitlab.com/anxiousprogrammer/s3dft`, 2021. Last accessed: 08.11.2022.

[137] Intel Corporation. *Intel(R) Advisor User Guide*, 2022.3 edition, 2022. Available at `https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top.html`.

[138] Intel Corporation. Where can i find information about flops per cycle for intel(r) processors? `https://www.intel.com/content/www/us/en/support/articles/000057415/processors.html`, 2021. Last accessed: 08.11.2022.

[139] Intel Corporation. *Intel(R) Xeon(R) Processor Scalable Family Specification Update*, 017 edition, October 2020. Available at `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf`.

[140] Matteo Frigo and G. Steven Johnson. *FFTW*. Massachusetts Institute of Technology, December 2020. Available at `http://www.fftw.org/fftw3.pdf`.

[141] Sukhyun Song and Jeffrey K. Hollingsworth. Designing and auto-tuning parallel 3-d fft for computation-communication overlap. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 181–192, New York, NY, USA, 2014. Association for Computing Machinery.

[142] Anthony Chan, Pavan Balaji, William Gropp, and Rajeev Thakur. Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems. In Ponnuswamy Sadayappan, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing - HiPC 2008*, pages 350–364, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[143] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1995.

Typeset January 20, 2026