Master in High Performance Computing

Optimization of postprocessing tools for understanding collective burst mechanism of angular jumps in liquid water

Supervisor(s):
Ivan Girotto, Ph.D. (ICTP - MHPC),
Natalia Manko, Ph.D. (ICTP - CMSP),
Ali Hassanali, Ph.D. (ICTP - CMSP)

Candidate:
Edward Ntim Gasu (Ph.D.)

10th edition
2023–2024

Abstract

This study aimed to optimize the computational efficiency of the automated protocol for detecting angular jumps in water molecules. The algorithm, which analyses dipole moment (DP), hydrogen-hydrogen (HH) vectors, and topological defects in hydrogen bonding across systems of varying sizes, faced performance bottlenecks as system size, trajectory length, and simulation time increased. These challenges led to frequent crashes and prolonged execution times. Initial profiling of the DP and HH extraction tool on the Leonardo Supercomputer, revealed high memory consumption (117 GB) and long execution times (4397 seconds). To address these challenges, the `MDAnalysis` package was initially leveraged for input processing and benchmarked against the custom input processing approach. Results showed that converting XYZ trajectory files to binary reduces memory usage and processing time. Among various binary file reading techniques implemented, using NumPy's `frombuffer` function performed best, reducing execution time and memory usage. A speedup by up to 265-fold in time compared to the original method, and 224-fold relative to `MDAnalysis` was recorded. Adapting optimized this new paradigm of input processing, the serial version of the DP and HH extraction protocol ran approximately 3-fold faster than the original in serial. Implementing multiprocessing for data parallelism with up to 8 processes further improved performance, achieving a 14-fold increase in speed while maintaining low memory usage ($\approx$ 93 MB) for all target analysis tasks. Performance analysis confirmed that parallel execution leading to the most optimal execution was achieved with 5 processing elements, beyond which a saturation was reached. This was due to resource contention during output processing. Quality checks on the molecular swing evaluation protocol were completed, incorporating updates to the jumps calculation protocol and plotting utilities. Additionally, the tool for calculating topological defects in hydrogen-bonded water molecules was enhanced by refactoring and optimized using `Numba's JIT` decorator along with the aforementioned coordinate generator. This reduced the serial execution time from 21 hours to 5 hours, representing a 4-fold speed-up. Using Jolib for data parallelism over 32 processes, further reduced execution time from 5 hours (18567.44 seconds) to an average of 697.23 seconds, representing a 25-fold speed-up, with about 6.2 GB maximum memory usage. Overall, this work identified computational bottlenecks presented by the two major tools of the automatized angular jumps detection protocol. By presenting optimized algorithms capable of processing large MD trajectories with minimal memory and time requirements, this work provides a solid foundation for enhancing the efficiency of post-processing after molecular dynamics simulation.

Acknowledgement

I would like to express my gratitude to the ICTP-MHPC Fellowship for sponsoring my enrollment and supporting my welfare throughout this specialization.
I am especially thankful to my supervisors; Dr. Ivan Girotto, Dr. Natalia Manko, and Dr. Ali Hassanali for their ideas, support, encouragement, critique, and guidance during my coursework and thesis.
My sincere thanks also go to the referee, Dr. Uriel Morzan (University of Buenos Aires, Argentina), for his time and support.

Contents

Chapter 1

Introduction

1.1 Background

Understanding the microscopic origins of collective reorientational motions in liquid water is crucial for advancing our knowledge of molecular dynamics (MD) and fluid behavior. Recent research by (1) has made significant strides in this area by employing an automated detection protocol to identify abrupt angular motions in MD simulations of water. This protocol enables a detailed analysis of how water molecules undergo reorientational changes, which is fundamental for various physical and chemical processes.

The automated detection protocol used in this study involved several key steps. Initially, MD simulations were performed on water molecules in cubic periodic boxes containing 1019, 3400, and 8152 molecules, over a period of 2 picoseconds (ps). The analysis relied on the dipole moment (DP) and hydrogen-hydrogen (HH) vectors to track angular changes in the molecular structure. The detection protocol comprised data filtering and preprocessing, derivative calculation of filtered vector time series, and cross-product calculation to determine the plane of rotation. Identifying peaks in the relevant quantities, the protocol successfully pinpointed the start and end points of each angular swing, providing outputs in terms of start time (t), duration (Δt), and magnitude ($\Delta\Theta$) for both the DP and HH vectors.

Despite the effectiveness of the initial implementation, significant computational performance issues were encountered as the system size and simulation time increased. The complexity of the dipole and hydrogen-hydrogen vector calculation tool depended on the number of atomic coordinates per timestep (m), resulting in a total complexity of $O(t \times m)$; the hydrogen bonding defect tool considered the number of coordinates processed and compared (w), as well as the number of timesteps (t) or frames, resulting in a total complexity of $O(t \times (m \times w))$. These complexities in time and memory (space) posed challenges while analyzing large-scale MD data.

To address these challenges, the main objective of this work was to enhance the computational efficiency of the analysis framework for handling extensive MD datasets, particularly those with large frame counts and simulations extending beyond 2 ps.

In MD simulations, the choice of file format for storing trajectory data is crucial for efficient data handling during post-processing. For example,

GROMACS, a widely used MD simulation software, uses two primary formats: TRR and XTC for storing trajectory data. TRR is a full-precision (float64) portable-binary format, while XTC is a reduced-precision (float32) portable-binary format designed for smaller file sizes. The XTC format offers very good compression of the data and is portable (2). Other MD simulation packages, such as DL-Poly, output several text-based file formats, including the resulting MD trajectory (commonly known as the HISTORY file).

Efficiently reading input data (9.1 GB XYZ formatted 1 ns MD trajectory) was for profiling the Python-based update of the original MATLAB-FORTRAN post-processing protocol used for understanding the collective burst mechanism of angular jumps in liquid water. The major tools in this protocol relied on the reading of the input trajectory file. Hence, this work explores the performance of various input data reading methods for a text (.xyz) output trajectory containing atomic coordinates across 125000 timesteps, generated from a 1 ns simulation of 1019 water molecules. Additionally, the advantages of converting the original text file into binary format, as described by Bressert (3), using the same dataset and a significantly larger one (129 GB), produced from a high-frequency (every 4 fs) coordinate production MD simulation of 8152 water molecules over 2 ns. The results of these input processing approaches were incorporated into a multiprocessing setup to parallelize the dipole (DP) and hydrogen-hydrogen (HH) vector extraction algorithm. Benchmarking analyses were performed to assess the parallel efficiency of the updated algorithms, enabling informed decisions for developing an optimized automated workflow. A second goal was to reduce the execution time of the hydrogen bonding defect analysis tool by adapting to efficient input processing and multiprocessing for improved performance.

1.2 Problem Identification

While the automated detection protocol effectively identifies angular changes in the molecular structure of water, its implementation faced challenges as system size and simulation time increased. The computational cost of calculating dipole moment (DP) and hydrogen-hydrogen (HH) vectors grew with the number of atomic coordinates per timestep and the number of timesteps, resulting in memory and performance bottlenecks. Addressing these bottlenecks became essential for the efficient analysis of large-scale molecular dynamics datasets.

Furthermore, hydrogen bonding defect analysis, another critical aspect of the study, also exhibited high computational complexity due to the large number of atomic coordinates per frame processed in each timestep. The main problem identified was the inefficiency in handling large datasets, particularly in terms of reading trajectory files and executing the angular jumps detection protocol. Optimizing the performance of the post-processing workflow was necessary to facilitate the study of reorientational motions in larger and longer MD simulations.

1.3 Hypothesis Formulation and Testing

Binary files, while more complex in terms of formatting and in some cases less portable compared to text files, have significant advantages in terms of file size

and read/write speeds, making them valuable for big data applications (3). This study hypothesizes that converting large text-based trajectory files (such as XYZ) to binary formats can significantly improve read/write performance and reduce memory overhead.

Optimization techniques envisaged to achieve this goal included code isolation, profiling to identify and optimize performance bottlenecks, ensuring that computational resources were utilized effectively. Additionally, high-performance computing (HPC) resources, including the Leonardo and Argo supercomputers, were leveraged for parallel processing using methods such as multiprocessing, among others. These techniques aimed to accelerate the processing of simulation data and improve overall efficiency. Furthermore, the angular detection algorithms were refined to enhance performance while maintaining the integrity of the original implementations. By implementing these optimizations, the aim was to improve the efficiency of processing large MD datasets, facilitating more detailed and timely analyses of molecular reorientational dynamics.

1.4 Thesis Structure and Relevance

This thesis is structured as follows: Chapter 2 begins by discussing the significance of calculating dipole (DP) and hydrogen-hydrogen (HH) vectors for understanding the physical properties of water. It then details the protocol used for analyzing the reorientational dynamics of liquid water, highlighting the main computational complexities involved in extracting DP and HH vectors as well as conducting hydrogen-bonding defect analyses.

Additionally, Chapter 2 presents preliminary findings on performance bottlenecks, based on execution time and memory usage metrics gathered from multiple runs of the DP and HH vector extraction tool. Furthermore, the state of the art of code optimization, traditional trajectory reading approaches (and their limitations), and performance measurements for Python based applications are reviewed. This chapter then concludes with the optimization objectives envisioned for the study.

Chapter 3 explores the optimization techniques applied to enhance the automated protocol's performance, particularly in managing large datasets and parallel processing. This includes descriptions of performance evaluation of serial implementations and parallel efficiency. Chapter 4 then presents the study outcomes, discussing insights into the performance gains achieved through optimization. Algorithmic details referenced in earlier chapters are provided in the appendices, while Chapter 5 summarizes key conclusions and suggests directions for future research.

One of the primary challenges to reproducibility in computational research is the frequent unavailability of the source code. Interactive software commonly used in data exploration does not typically log user actions in a reproducible manner. Even when code is written, the integration of multiple software packages often leaves portions of code undocumented. Addressing this requires either adapting current software practices or encouraging researchers to adopt more reproducible software systems. This work examines how advanced HPC techniques can enhance the performance of existing and translated code to analyze collective jumps in liquid water, thereby promoting adaptability,

collaborative development, and reproducibility.

This work contributes to ongoing research by proposing efficient post-processing methods for large-scale molecular dynamics (MD) simulation data. The optimized protocol for studying collective reorientational motions in liquid water opens new avenues for studying extensive and longer MD simulations, thereby providing fresh insights into molecular behaviors in complex fluid systems.

Chapter 2

Literature Review

2.1 Dipole moment and Hydrogen-Hydrogen Displacement in water

The calculation of dipole moments and hydrogen-hydrogen (H-H) vectors is essential in post-processing molecular dynamics (MD) simulations of water molecule trajectories for several key reasons.

First, the dipole moment of a water molecule serves as a measure of the separation of positive and negative charges (4). Given that water is a polar molecule, the direction and magnitude of its dipole moment provide valuable insights into the orientation and polarization of individual molecules during the simulation. This information is crucial for understanding water's role in solvation, hydrogen bonding, and electrostatic interactions. Furthermore, the overall dipole moment distribution in the system influences macroscopic properties such as the dielectric constant (5), permittivity, and polarizability (6), which are vital for studying water's behavior in various environments, particularly near charged surfaces, ions, or in confined spaces (7). Additionally, water dipoles significantly affect long-range electrostatic interactions, which are critical in simulations involving biological systems, membranes, and proteins, where water mediates interactions among biomolecules (8).

The calculation of hydrogen-hydrogen (H-H) vectors, representing the vector between the two hydrogen atoms in a water molecule, plays a key role in determining the orientation of water molecules. By tracking changes in this vector, researchers can gain insights into how water molecules rotate and reorient themselves, particularly in hydrogen-bonding networks or under the influence of external fields, such as electric fields (9). Given that hydrogen bonds are fundamental to water's structure and behavior, the computation of H-H vectors facilitates the identification of hydrogen bond angles and lengths, shedding light on the strength and directionality of the hydrogen bonding network. Moreover, calculating H-H vectors enables the detection of bonding defects or abnormalities within this network, which are crucial for understanding water's anomalous properties, including density anomalies and surface tension.

The dipole and H-H vectors are also pivotal for examining molecular dynamics under different conditions. They are essential in studying how

water molecules respond to external electric fields, shear forces, or other perturbations. In systems containing solutes, such as ions, proteins, or polymers, the orientation and dynamics of water dipoles and H-H vectors near these solutes help elucidate hydration effects, solvation structures, and even mechanisms of protein folding and unfolding (10).

Lastly, the temporal evolution of dipole moments and H-H vectors can be utilized to compute time correlation functions (11), providing insights into molecular relaxation processes, rotational diffusion, and viscosity. These vectors also play a significant role in the vibrational dynamics of water, which are often investigated through spectroscopic techniques like infrared (IR) and Raman spectroscopy (12).

In summary, the calculations of dipole moments and H-H vectors are central to understanding the structure, dynamics, and interactions of water molecules in molecular dynamics simulations.

2.2 Automatized Detection of Angular Jumps

Water reorientation dynamics include various processes happening at different time scales, from very fast vibrational motions to slower reorientation through sudden large-amplitude angular jumps (swings). Offei-Danso and coworkers (1) have previously described how each of these processes is involved in the collective burst mechanism of angular jumps in liquid water.

In order to track angular changes in water orientation, two body-fixed vectors, the dipole moment (DP) and the HH vector (see Fig. 2.1) are used. From the MD simulation, the DP and HH vector time series for each water molecule are extracted, referred to as $v(t)$, with t being time. A new time series, $v_F(t)$, was constructed by filtering $v(t)$ using a second-order low-pass digital Butterworth filter implemented in MATLAB.

Using the filtered time series, the derivative of $v_F(t)$ is computed using a finite difference method, and the cross product of this vector is calculated with $v_F(t)$ to obtain a new vector:

$$\mathbf{n}(t) = \mathbf{v}_F(t) \times \frac{d\mathbf{v}_F(t)}{dt}, \tag{2.1}$$

which corresponds to the vector perpendicular to the plane of rotation of the fixed-body vector $v_F(t)$. The start and end points of swing events are identified as large instantaneous changes in the direction of $\mathbf{n}(t)$. More precisely described by the following quantity:

$$q(t) = 1 - \frac{\mathbf{n}(t) \cdot \mathbf{n}(t + dt)}{|\mathbf{n}(t)||\mathbf{n}(t + dt)|}, \tag{2.2}$$

which is equal to zero during the swing. At the start and end of the swing, this quantity is non-zero, indicating a change in the plane of rotation. Consequently, the start and end times of angular swings were identified as maxima in $q(t)$, which are known to be consistent with extrema in the filtered time series. Finally, the duration of the swings is determined by the times between two peaks of $q(t)$, and the magnitude is calculated by computing the angle between the unfiltered DP or HH vector at the start and end point of the swing.

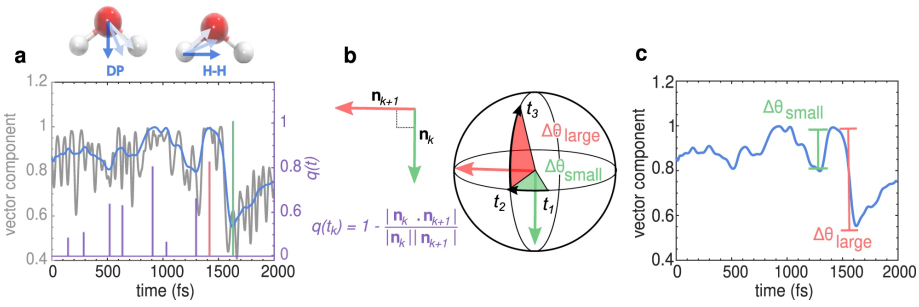


Figure 2.1: (a) Definition of dipole (DP) and HH vectors per molecule, with the x-component (DPx) time series over 2000 fs (gray), filtered to remove high-frequency oscillations (blue), and function $q(t)$ indicating angular swing points (purple). (b) Two successive swing events detected by $q(t)$ based on the change in direction of vector $n(t)$ (perpendicular to the rotation plane), shown for intervals (t_1, t_2) (green) and (t_2, t_3) (red). (c) Angular swings, $\Delta\theta_{\text{small}}$ (green) and $\Delta\theta_{\text{large}}$ (red), marked by extrema in the filtered DP vector time series.

The final output of the swing analysis protocol is the start time t, duration Δt, and magnitude $\Delta\theta$ for each angular swing detected.

2.2.1 Filtering of the Time Series

A 2nd-order low-pass digital Butterworth filter was applied to the time traces of the molecular dipole moments and HH-vectors. This filter, initially implemented in MATLAB, was designed to maintain a flat frequency response in the passband. The trajectories were sampled every 4 fs, with a cutoff frequency of 10 THz (equivalent to $\frac{1}{100}$ fs^{-1})—the frequency at which the filter's magnitude response reaches $\frac{1}{\sqrt{2}}$.

To further smooth the trajectories before automated detection of angular swings, a mean filter was applied using MATLAB's `smooth` function with a span of 25 data points (corresponding to a time window of 100 fs). During post-processing, a low-pass Butterworth filter with a cutoff frequency of 0.1 THz (equivalent to $\frac{1}{10000}$ fs^{-1}) was used on the time series of defect fractions and swing counts to capture fluctuations at a 10 ps timescale. This filtering approach was consistently applied across all water models and system sizes.

2.2.2 Analysis of Hydrogen Bonding Defects

Hydrogen bonds are analyzed by applying a geometrical criterion established by Luzar and Chandler (13). A pair of water molecules is considered to be hydrogen-bonded if two conditions are met. First, the distance $r_{O_D-O_A}$ between the donor oxygen (O_D) and the acceptor oxygen (O_A) atoms must be less than or equal to 3.5 Å. Second, the angle $\theta_{H_O-O_A}$, which is defined between the vector connecting the hydrogen atom (H_D) with its donor oxygen and the vector between O_D and O_A, must be less than 30°.

Again as described by Offei-Danso and coworkers (1), to identify defects in the hydrogen bonding network, the analysis proceeds through several

computational steps. The first step involves coordinate reading, where the trajectory file is parsed, and each frame of atomic coordinates is read. Each water molecule's position data is then extracted into arrays for further analysis.

Next, the distance calculation is performed with periodic boundary conditions (PBC). The function `dis` computes the minimum distance r_{ij} between two atoms, adjusting for PBC by utilizing the minimum image convention. This is given by the equation:

$$r_{ij} = \sqrt{\sum_k \left(v_{i,k} - v_{j,k} - L \times \text{round}\left(\frac{v_{i,k} - v_{j,k}}{L} \right) \right)^2} \tag{2.3}$$

where $v_{i,k}$ and $v_{j,k}$ are the coordinates of atoms i and j, respectively, and L is the box dimension.

The next step is angle calculation, performed by the `angle` function. This function computes the angle θ between two vectors, normalized for the PBC. Using the dot product, the angle is defined as:

$$\theta = \cos^{-1}\left(\frac{\vec{v}_1 \cdot \vec{v}_2}{|\vec{v}_1||\vec{v}_2|} \right) \tag{2.4}$$

where $\vec{v}_1$ and $\vec{v}_2$ are the vectors under consideration. This criterion allows the code to select molecular pairs with angles below the 30° threshold required for hydrogen bonding.

Following the distance and angle calculations, a defect matrix is generated. Using the established distance and angle criteria, this connectivity matrix is created, where entries are set to 1 if two molecules meet the hydrogen bonding requirements and 0 otherwise. The matrix is symmetric, representing the connectivity between particles.

Finally, the defect and index calculation is performed. The defect score for each molecule is computed by summing its connected partners within the matrix. Additionally, indices of hydrogen-bonded molecules are stored for further analysis, capturing the hydrogen bonding network at each time step.

2.3 Computational Complexity of the Automated Protocol

As described earlier, to detect angular swings in water orientational motions and analyze hydrogen bond dynamics that ensue involve a series of computational steps, each contributing to the overall complexity in space and time. Computational complexity is a way to describe how runtime or space requirements of an algorithm, grows as the size of input and the number of operations (and their frequency) performed on the input increase. To be able perform a satisfactory diagnosis of the sources of computational bottlenecks presented by scientific software, a review of the computational complexity of algorithmic formulations becomes essential. Also, a complete understanding of the computational complexity of the automated protocol will provide insights to improve computational efficiency. In the proceeding sections, the computational complexity of key processes, from initial trajectory processing through to hydrogen bonding defects analysis is vetted and described.

2.3.1 Trajectory Processing and Filtering

The initial step to detect angular jumps involves processing the trajectory data obtained from molecular dynamics simulations of water molecules to generate the time series of the dipole moment (DP) and HH vectors for each molecule. As seen earlier, this time series is filtered using a second-order low-pass Butterworth filter.

Trajectory data extraction involves parsing the atomic coordinates of each molecule across all frames, resulting in a complexity of $\mathcal{O}(N \cdot T)$, where N represents the number of water molecules and T the total number of time steps (each time step also known as a frame of the trajectory). For each frame, the algorithm must process the coordinates for each molecule individually. Next, Butterworth filtering, with a similar complexity of $\mathcal{O}(N \cdot T)$, requires iterating over the entire time series for each molecule to apply the filter function, smoothing the data and enabling more accurate capturing of angular jumps.

2.3.2 Dipole, HH Vector and Swing Magnitude Calculation

To compute the dipole (DP) and HH vectors for each molecule, vector operations were used. In addition, to identify angular jumps (also known as swings) derivatives, cros-products, cosine similarity (geodesic changes) computations using either DP or HH vectors are leveraged to estimate the magnitude and duration of the molecular re-orientations.

The complexity of the derivative calculation is $\mathcal{O}(N \cdot T)$, requiring finite differences over the entire time series for each molecule, while the complexity of the cross product calculation is also $\mathcal{O}(N \cdot T)$, as the cross product must be computed at each time step for every molecule.

2.3.3 Defect Calculation and Hydrogen Bond Network Analysis

Defect detection requires calculating distances and angles for each pair of neighboring molecules and evaluating these against established hydrogen bond criteria. The distance calculation, which includes adjustments for periodic boundary conditions (PBC), has a complexity of $\mathcal{O}(N^2)$ for each time step, as each molecule must be compared with every other molecule. Similarly, the angle calculation also exhibits a complexity of $\mathcal{O}(N^2)$ per time step, as angles are computed for each pair that satisfies the distance criterion. The generation of the defect matrix, representing hydrogen bonding status, likewise has a complexity of $\mathcal{O}(N^2)$ per time step due to the creation of a connectivity matrix of size $N \times N$. Finally, the defect and index calculation, which derives each molecule's defect score and bonding indices from the connectivity matrix, also has a complexity of $\mathcal{O}(N^2)$ per time step. For T timesteps, the overall computational complexity becomes $\mathcal{O}(N^2) \cdot T$. The computational complexity of each process is summarized in Table 2.1.

Process	Complexity
Trajectory Data Extraction	$\mathcal{O}(N \cdot T)$
Smoothing & Butterworth Filtering	$\mathcal{O}(N \cdot T)$
∇ and $\times$ Calculations	$\mathcal{O}(N \cdot T)$
$\|\mathbf{a} - \mathbf{b}\|$, Θ and Defect Matrix	$\mathcal{O}(N^2 \cdot T)$
Defect and Index Calculation	$\mathcal{O}(N^2 \cdot T)$

Table 2.1: **Summary of computational complexity for major steps in the automated protocol.** Here, N represents the number of molecules, T is the number of time steps, $\|\mathbf{a} - \mathbf{b}\|$ is the Euclidean distance, ∇ is the gradient operator, and $\times$ denotes the cross-product.

2.4 Code Review

The transition of the protocol from FORTRAN and MATLAB to Python was motivated by Python's readability, extensive scientific libraries, and robust community support, all of which contribute to its adaptability for high-performance computing (HPC) applications. This conversion was envisioned to create an interactive Python-based protocol providing a foundation for further optimizations and analysis scalability. Specifically, such a translation provides the following advantages for the analysis protocol: first, python's readability and accessible syntax lower the barrier for entry for other researchers to use and modify the code which also aligns with the objective to update the application. As Python's flexibility supports the development of user-friendly, interactive interfaces this language shift enhances collaborative work and enables researchers without specialized programming knowledge to contribute or adapt the protocol (14).

Secondly, the updated protocol also benefits from cross-Platform compatibility since by moving from MATLAB and FORTRAN to Python, the tools of the protocol could possibly be run across different operating systems, including, cloud computing environments, such as Google Colab, compute clusters and supercomputers, without specific licensing restrictions. This benefit is particularly significant when running simulations on high-performance computing resources, as Python's compatibility with Linux systems ensures consistency and reproducibility of results (15). Another objective would be to take advantage of NumPy and Scipy efficiency in relation to data handling. Computational operations with both `NumPy` and `SciPy` achieve high-performance capabilities due to their core implementation in C and C++. `NumPy` array operations are primarily written in C, enabling fast element-wise operations, efficient memory usage, and performance comparable to C-level code, while still offering Python's high-level interface. Also, `NumPy` wraps highly optimized libraries like BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package) (16), commonly implemented in C or Fortran, to handle complex linear algebra efficiently. Building on `NumPy`, `SciPy` extends functionality with modules for scientific computing tasks such as optimization, integration, and eigenvalue problems. Many parts of `SciPy` are implemented in C, C++, and Fortran, leveraging libraries like FFTPACK for fast Fourier transforms. This use of low-level language implementation enables both `NumPy` and `SciPy` to manage large-scale computations efficiently, making

them highly suitable for high-performance computing (HPC) and scientific applications. The original automated protocol's dependency on MATLAB for vector manipulation for instance, gets replaced with Python's optimized NumPy and SciPy libraries, for handling large arrays and performing signal processing tasks like Butterworth filtering and sorting among others (17). The computational efficiency and streamlined data handling afforded by these libraries promised direct support of the language transition goals of the tools of the protocol.

The enhanced visualization provides the possibilities of post-processing of data for useful information using Python's visualization libraries (e.g., Matplotlib, Seaborn, Plotly) allow for the creation of detailed and interactive plots, which could enhance the visual analysis of angular swings and reorientation dynamics. Clear visual feedback can facilitate debugging or data validation in HPC and as well, more reproducible insights can be obtained from adapting the protocol to understand the dynamic properties of water molecules in other systems of physical and biophysical interest.

Ultimately, scalability, community support, and reusability are easily achieved by transitioning the tools of the jumps detection protocol into python. It is noteworthy that, Python's compatibility with parallel processing libraries, such as Dask (18) and `multiprocessing`, supports efficient workload distribution, facilitating optimization strategies to manage larger datasets effectively. Also, transitioning to an open-source language while engaging a scientific community usually enhances adaptability. This adaptability will not only enhances the protocol's longevity but also ensures that researchers can benefit from ongoing improvements and insights driven by the community, promoting a sustainable research tool that evolves with advancements in the field.

Lastly, Python's general-purpose nature and modular structure provide future adaptability for the code, making it suitable for other molecular or rotational dynamics analyses. This versatility corroborates translation objectives while improving performance and usability, as Python's framework establishes a solid foundation for continued innovation and interdisciplinary applications.

To assess the initial performance of the Python version of the protocol, time and memory benchmarks were performed on the Booster module of the Leonardo supercomputer. The Booster Module consists of 3,456 custom BullSequana X2135 "Da Vinci" blade servers (nodes), each equipped with an Intel Xeon 8358 CPU (32 cores at 2.6 GHz), 512 GB RAM, 4 NVidia custom Ampere GPUs (64 GB HBM2), 2 NVidia HDR InfiniBand network adapters (two 100 Gbit/s ports), delivering a peak performance of 89.4 TFLOPs per node (19). Focusing on the dipole and hydrogen-hydrogen (H-H) vector extraction code and using Python's `time` and `memory_profiler` packages in an Anaconda environment installed with Python 3.11.5, both execution time and memory usage measurements (see Appendix A.1) showed that, an average of 4,406 seconds (approximately 1 hour and 22 minutes), along with a memory footprint of 117.07 GB were required for successful serial execution. These results highlighted significant memory demands, pointing to key limitations in the protocol's efficiency when applied to large-scale datasets or extended simulation times. Reducing these bottlenecks were envisaged as central to optimization efforts.

2.5 Code Optimization to Enhance Computational Performance

In performance evaluation for optimization, benchmarking of code could be time and resource consuming. The "expensive cost", limits the number of trial and error iterations engineers can perform in a given budget. An interesting and versatile approach to optimization is to set objective(s) to reduce cost in time. This is generally regarded as a single objective optimization. The goals of multi-objective optimization encompasses technical cross-platform, energy and hardware costs.

To reduce cost for software optimization for the title protocol, a probe into the rationale for high memory consumption during DP and HH vector extraction for instance, could be a crucial entry point. This is because execution instability could emerge in applications projecting such memory-bound performance bottlenecks when run in computing environments fitted with low memory architectures. It is not obvious immediately from a test run what questions could be raised without the use of standardized approaches for application benchmarking for optimization. Usually, in scientific application development, the hot spots represent a small fraction of the total source lines. Extraction of the performance hot spots of an application as isolated fragments (codelets) is broadly termed as code isolation. As a standard approach to application development, code isoaltion reduces benchmarking costs and allows piecewise optimization of an application. Codelets can be modified, compiled, run and measured independently of the original application. Codelets could correspond to loops, serially executed functions, and computationally expensive operations among others.

Breaking an application into independent codelets provides multiple benefits. Executing isolated codelets instead of whole applications is faster and enables piecewise evaluation and optimization as mentioned earlier. Indeed different codelets may expose different performance bottlenecks and react differently to standalone optimizations. With code isolation, they can be individually modified to evaluate the payoff of new optimizations and tune performance at a fine-grain level. It is noteworthy that, practically, the entry point to code isolation is that, source loops for instance are outlined and instrumented with profiling probes to pinpoint sources of performance bottlenecks.

Furthermore, exploiting codelet similarities follows from code isolation. This approach help reduces the search space for HPC development. Reducing the search space in this context is to find a minimal set of benchmarks that could faithfully capture and represent the original behavior of the entire application. Generally, there are two important sources of redundancy that could be exploited for optimization purposes; first is multiple invocations and second, similar computation kernels. Codelets that are repeatedly invoked with the same context in an application's lifetime have the same running time for each invocation. In an application where a single codelet is called a thousand times, measuring only a few invocations achieves significant gains in costly benchmarking times. And also, there is no need to measure multiple copies of the same code.

As shown earlier in the protocol description, input processing is crucial

for efficient execution of the protocol for both DP and HH vector extraction and hydrogen bonding defects analysis. Therefore, piecewise measurements and probing of the input processing codelet(s) of the protocol is important for reducing benchmarking cost, paving way for hot spot identification and overall exploitation of potential payoffs of envisioned optimizations.

To improve computational performance and scalability, a series of targeted optimization strategies are proposed, implemented and measured. Algorithmic improvements are iteratively addressed where inefficient segments are replaced by introducing more efficient versions. For instance, replacing less efficient "vanilla" data handling structures, with available standard library versions including the adoption of NumPy arrays for vectorized operations.

Parallel computing techniques in Python are primarily employed to enhance performance. Libraries like `multiprocessing` and `Dask` facilitate the distribution of workloads across multiple processors, thereby reducing processing times for large datasets and improving scalability. It is important to note, however, that efficient parallel implementation is not always achieved by merely parallelizing individual steps of a serial program. Instead, optimal performance often requires rethinking the algorithm as a whole to design a solution better suited for parallel execution.

2.6 MDAnalysis for Data Handling

MDAnalysis is a Python-based library designed to efficiently read, analyze, and manipulate molecular dynamics (MD) trajectories. Supporting multiple trajectory formats, it allows seamless integration of diverse datasets, making it an invaluable tool in HPC environments. Its object-oriented API simplifies molecular data manipulation, providing a streamlined and flexible framework for a range of MD tasks.

In addition to MDAnalysis, several other software packages (CHARMM (20), VMD (21), and Gromacs (22)) were tested on representative tasks available across all packages. Using a 10-ns trajectory with 10,000 frames of a fatty acid-binding protein (PDB ID: 1IFC) in water (13,051 atoms), the analysis included calculating time series for three distances, two dihedral angles, the RMSD relative to the initial conformation, and the protein's radius of gyration. Additionally, the density of water oxygens around the protein was computed from the same trajectory, and the radial distribution of 3,476 water oxygens in a urea solution was calculated from 2,000 frames of a trajectory with 11,492 atoms. Performance was assessed on an Intel Xeon E5420 CPU at 2.5 GHz.

The results indicate that MDAnalysis performs competitively and, in some cases, surpasses other software when conducting simple analyses, such as distance measurements, dihedral calculations, density mapping, and radius of gyration determination. However, more computationally intensive tasks, particularly those that involve large array manipulations (e.g., radial distribution functions), may exhibit performance limitations in MDAnalysis (23). This observation suggests that the library's efficiency varies depending on the nature of the analysis task.

Given this task-specific performance metrics, MDAnalysis routines present untapped potential for optimizing post-processing toolkits in HPC applications. While the authors suggested improvements for complex analysis tasks through

the use of Cython or direct C implementations without NumPy, it remains uncertain if further optimization within the NumPy-based backend (itself C/C++-supported) could offer similar gains. And it is important to note that, despite NumPy's speed benefits from its C/C++ foundation, handling large arrays—common in HPC settings—may not fully exploit multi-core architectures without further optimization or parallel processing.

Thus, exploring a corresponding MDAnalysis input processing codelet (specifically the XYZ-trajectory reader) for computational performance details is crucial. This would enable a deeper understanding of performance bottlenecks and help answer the open question: "What are some of the most computationally efficient approaches to process input data for NumPy-based MD post-processing tasks (whether serial or parallel) of the automated protocol when managing very large datasets in an HPC environment?"

2.7 Performance Analysis of Parallel Python Applications

The performance of Python-based applications in parallel computing environments is typically measured through benchmarks focusing on key metrics such as execution time and memory usage. These metrics provide insights into how well Python applications perform under various computational and communication demands, helping identify bottlenecks and areas for optimization. Studies demonstrate that detailed performance analysis can significantly aid in understanding scalability limits and highlight areas needing further improvements for HPC applications.

Several tools facilitate parallel computing in Python, with the `multiprocessing` module from Python's standard library and the external `MPI4Py` library being widely used. MPI4Py is particularly valuable as it offers an object-oriented interface for MPI-based communication, comparable to C++, enabling efficient communication of both standard Python objects and NumPy arrays (24; 25). Its close-to-native C or Fortran communication speed has made it a key tool in HPC contexts, where Python's high-level expressiveness is combined with optimized C-based operations (e.g., with BLAS, LAPACK, and NumPy libraries). Applications like GPAW, a computational chemistry package, demonstrate this combination's effectiveness, achieving around 25% of peak performance with strong scalability across tens of thousands of cores (26; 27).

Scalability assessments, such as those conducted by Enkovaara et al. (2011) (27), involve benchmarking applications like GPAW with increasing input sizes to observe performance trends under larger computational loads. These tests revealed insights into the computational and memory efficiency of GPAW when run on HPC systems with high core counts. A test case involving H_2 molecule calculations to demonstrate that Python-based HPC version of GPAW could achieve competitive performance on large-scale systems (28) was similarly carried out by Wagner et al. (2017) (29).

In their work, methods to instrument the MPI runtime for Python were developed, enabling detailed performance analysis analogous to that for C/Fortran applications such as GPAW. Using Extrae, a tracing tool that adds support for Python, the authors successfully applied these techniques

to analyze both the traditional C/Fortran and the Python versions of the GPAW application on the MareNostrum (30) supercomputer at the Barcelona Supercomputing Center. Their approach allowed for capturing fine-grained details of process behaviors, pinpointing efficiencies, and correlating them with source code—achieving a level of analysis previously restricted to lower-level languages. Tracing outcomes demonstrated that equivalent performance information as for traditional C or Fortran applications were recorded, therefore, offering the same profound analysis capabilities now for Python, as well.

Basic profiling tools, such as Python's `profile` and `cProfile` modules, offer function- and line-level timing but are limited in depth for parallel applications (31). More advanced profiling and tracing, especially for `multiprocessing`-based Python applications, remains under development. For example, tracing Python applications using the `multiprocessing` module has been partially implemented in Extrae, with functionality to track processes and emit lifecycle events. However, support for Python's `multiprocessing.Pool` (32) module is still evolving, suggesting that while current tools provide foundational support, further development is needed to fully leverage this tool for performance assessments of codes using Pool objects.

2.8 Optimization Objectives

Having demonstrated the significance of programming language transition, the primary objectives of optimization were focused on reducing execution time, reducing memory consumption, and enhancing user accessibility of the Python version of the automatized protocol for analyzing angular jumps and hydrogen bond defects in liquid water. Quantitative targets included reducing execution time by an unspecified (but satisfactory percentage) and capping memory usage within a set threshold such that, the automated protocol could run smoothly on laptops (or a compute node in an HPC cluster), cloud computing devices (e.g. Google Colab), as well as other systems with low memory budgets. Beyond performance, usability improvements aimed to minimize latency, allow for the processing of larger simulation datasets, and ensure adaptability to evolving research needs. Additionally, the emphasis on maintainability and extensibility required that the protocol could be placed in a state where it can be supported with much efficient improvements and modifications, making it responsive to new scientific inquiries and computational advancements.

Chapter 3

Methods

3.1 Computational Resources, and Profiling

Unless otherwise specified, a 1 ns (1 million steps) MD simulation trajectory of 1019 water molecules, stored in .xyz format (water_lmp_3057_10-001.xyz, 9.2 GB), was used throughout this work. The dataset, generated from an MD production run, consists of the positions of the water molecules stored every 8 fs, resulting in 125,000 frames. Each frame's metadata includes the atom count and header information, which specifies labels "Atoms" and "Timestep: n." Atom types are labeled as "1" for oxygen and "2" for hydrogen, followed by atomic coordinates separated by space delimiters.

Additionally, a Jupyter notebook containing all analysis tools of the automated protocol was provided by Dr. Natalia Manko. The DPHH vector extraction tool was profiled using the `time` and `memory_profiler` packages wrapped around the main function of the script (see Appendix A.1). The Booster module on the Leonardo supercomputer, consisting of 3,456 custom BullSequana X2135 "Da Vinci" blade servers, was used for iterative executions, unit testing, profiling, benchmarking and performance assessments. Each node is equipped with an Intel Xeon 8358 CPU (32 cores at 2.6 GHz), 512 GB RAM, 4 Nvidia custom Ampere GPUs (64 GB HBM2), and 2 Nvidia HDR InfiniBand network adapters (two 100 Gbit/s ports), offering a peak performance of 89.4 TFLOPs per node (19).

Aggregated system performance statistics were collected during code executions using the `perf` package. Specifically, the performance metrics included cycle counts, instruction counts, cache references, cache misses, branch predictions, branch mispredictions, task-clock timing, faults, minor faults, context switches, and process migrations. These statistics were gathered for the original and optimized codelets (both for serial executions and for parallelized versions).

3.2 Optimization Approach

To enhance performance, code fragments were first inspected and profiled to identify and confirm bottlenecks. Initial optimizations included intuitive best practices for improving code readability and operability. Following this,

hypotheses were formulated and tested through algorithmic modifications, iterative profiling, and benchmarking with two datasets: the main dataset described earlier, and a second dataset featuring eight times the system size with atomic coordinates stored every 4 fs over a 2 ns production simulation with GROMACS-2023.

Hypothesis Testing and Bottleneck Identification: Input trajectory processing for DP and HH vector extraction, as well as defects analysis tools, was analyzed to detect bottlenecks contributing to high memory consumption and increased execution times. The MDAnalysis package was evaluated for its potential benefits in handling XYZ input processing efficiently.

Probing Binary Input Processing: A binary input processing approach was also conceived and tested. To facilitate this, a Python script was created to convert the trajectory data into a binary stream. After conversion, the script included a binary reader for file integrity assurance, comparing the first and last frames of the binary file with the original .xyz format. Alternative binary input processing functions were implemented and tested to identify the most efficient option for minimizing memory usage and optimizing input processing speed. This evaluation aimed to select an approach that balanced low memory demands with faster processing, enhancing performance in both serial and parallel implementations of the DP and HH vector extraction and hydrogen bonding defects analysis tools.

Parallelization and Further Optimizations: To improve computational efficiency, code execution and file output operations were refactored for concurrent execution using Python's multiprocessing package and Joblib. For the defects analysis tool, functions were refactored and optimized for serial execution using binary input processing and Numba JIT decorators. Execution time and memory usage were iteratively measured to identify which functions provided significant performance gains when decorated with Numba. The serial execution was also adapted to support data parallelism using the Joblib library. The parallel version was profiled and compared to the serial version to analyze computational gains emerging from optimization choices.

3.3 File Reading Methods

3.3.1 XYZ File Reading Methods

The following methods were used to read XYZ files:

- **Original-XYZ Reading:** Reads the XYZ file line-by-line using a custom function. This straightforward approach extracts atomic coordinates but may be less efficient for large files due to its line-by-line processing (see Appendix B.1.2).

- **Reading with MDAnalysis:** Utilizes the MDAnalysis package to load the entire XYZ trajectory into memory and retrieve atomic coordinates. This method provides improved performance and memory efficiency, particularly for large datasets, by leveraging specialized tools designed for handling molecular dynamics data (see Appendix B.1.3.

3.3.2 XYZ to Binary File Conversion

The conversion of XYZ files to binary format is managed by the following method:

- **XYZ to Binary Conversion:** This method reads an XYZ file line-by-line, extracts the number of atoms, header information, and atomic coordinates, and writes this data into a binary file. The conversion format includes storing the number of atoms, header length, and atomic details as integers and floating-point numbers for efficient storage and retrieval (see Appendix B.1.4).

3.3.3 Binary File Reading Methods

The following methods were evaluated for reading binary files of atomic data, focusing on memory usage, processing speed, and scalability:

- **Chunked Reading with Loop:** This method reads and processes (unpacking in a for-loop) the binary file in chunks, where for each timestep, the size of each chunk is explicitly controlled by a predefined parameter (`chunk_size`) (33). Coordinates are extracted incrementally, allowing for memory-efficient processing of large files. However, this approach can involve additional overhead from repeated file I/O operations.
- **Full Reading with Loop:** This method reads full coordinate data for each timestep into memory at once while unpacking data in a for-loop, without chunking (33). While this reduces file I/O operations and can improve processing speed, it requires sufficient memory to store all the data for a single frame before processing.
- **Chunked Reading with `np.frombuffer`:** Similar to "Chunked Reading with Loop," this method processes the binary file in predefined chunks. However, it utilizes NumPy's optimized `frombuffer` function (34)to decode the binary data into arrays directly, which accelerates processing and improves efficiency while maintaining memory control.
- **Full Reading with `np.frombuffer`:** This method loads the entire coordinate data for a frame into memory and then uses NumPy's `frombuffer` (34) for vectorized conversion to arrays. It benefits from reduced overhead and efficient array manipulation but requires sufficient memory to handle large frames in their entirety.

Each method provides different trade-offs between speed, memory usage, and complexity, with choices depending on the specific needs of the data processing functions and available resources. Detailed descriptions have been included in Appendix B.1.5.

3.4 Benchmarking Procedure

An execution time and memory usage wrapper was used to monitor the global execution times and memory usage. The results were visualized using

matplotlib to compare read times and memory usage across methods. The plot includes bars representing time and a line graph depicting memory usage, providing a comprehensive view of the performance characteristics of each file reading approach. Each method was benchmarked for both time and memory usage:

- **Time Benchmarks:** The execution time for each method was measured using the `time.time()` function, with methods for reading the XYZ and binary files evaluated separately.
- **Memory Benchmarks:** Memory usage was assessed using the classic `memory_usage` function from the `memory_profiler` package, capturing peak memory consumption during file reading operations.

3.5 Performance Analysis

3.5.1 Speedup and Parallel Efficiency Calculation

Speedup quantifies the performance improvement of a given method relative to a baseline. It is calculated as the ratio of the execution time of the baseline method (e.g., Original-XYZ) to the execution time of the method being compared. The formula used for speedup calculation is:

$$\text{Speedup} = \frac{\text{Execution Time of Baseline Method}}{\text{Execution Time of Compared Method}}$$

For this analysis, the fastest method was used as the baseline. The relative speedup (RSU) was computed as the inverse of the speedup value, to show performance differences.

Parallel efficiency evaluates how efficiently a parallel algorithm utilizes multiple processes compared to a serial implementation. It is defined as the ratio of the achieved speedup to the number of processes used. Mathematically, the parallel efficiency $E(p)$ for a given number of processes p is expressed as:

$$E(p) = \frac{\text{Speedup}}{p} = \frac{T_{\text{serial}}}{p \times T_{\text{parallel}}(p)}$$

where:T_{serial} is the average execution time of the serial version of the code and $T_{\text{parallel}}(p)$ is the average execution time of the parallel code using p processes.

This efficiency metric was calculated for the optimized scripts (`dphhbinwl.py` and `dphhbinfb.py`), and the results were compared across varying numbers of processes to evaluate scalability and parallel performance.

3.6 Quality Checks

Angular jumps (swings) were computed using a dedicated tool, which required minimal modifications since it exhibited neither computational nor I/O bottlenecks. Quality checks were mostly done output file comparison. The visualization tool was modified to use `kdeplot` instead of `distplot`. The probability distribution plot of angular jumps over time was utilized as the final quality check to validate outputs from all optimized code implementations.

Chapter 4

Results and Discussion

4.1 Profiling Results for Input Processing

A comparison is herein made for the performance and memory usage of the different input processing methods tested, with particular emphasis on speedup factors achieved. The fastest method, was `Bin (Full, FB)`, serves as a benchmark against which relative speedup is assessed.

Test outcomes revealed that binary file reading using `np.frombuffer` (FB) was the most efficient in both full-reading and chunked (with looped unpacking) configurations. For comparison, the `Bin (Chunk, FB)` method was only slightly slower than the fastest, while the custom XYZ processing approach was over 265 times slower. The `MDAnalysis-XYZ` method performed similarly in memory usage to the binary reading methods but was significantly slower.

Table 4.1 summarizes the time, memory usage, and relative speedup for each method.

Table 4.1: Average Time, Memory Usage, and Relative speedup (RSU) for Each Method

Method	**Time (s) ± SD**	**Memory (MB) ± SD**	**RSU**
Original - XYZ	425.06 ± 2.50	87928.53 ± 2.03	1.0
MDAnalysis - XYZ	358.46 ± 0.85	5987.31 ± 0.40	1.2
Bin (Chunk, WL)	200.67 ± 11.74	5991.23 ± 0.11	2.1
Bin (Full, WL)	186.15 ± 3.20	5991.22 ± 0.10	2.3
Bin (Chunk, FB)	3.06 ± 0.04	6002.71 ± 0.12	139.0
Bin (Full, FB)	1.60 ± 0.04	6002.75 ± 0.11	265.7

The `Bin (Full, WL)` and `Bin (Chunk, WL)` methods also achieved comparatively fast execution while maintaining low memory usage similar to `MDAnalysis-XYZ`. The efficiency of the fastest methods can be attributed to the elimination of intermediate list storage for coordinates, instead directly reading data into arrays, which ensured better cache locality and reduced memory overhead. As a result, optimized binary methods effectively address the high memory and time demands observed for custom XYZ file processing. The $\approx 5987.31 \pm 0.40$ MB recorded for both `MDAnalysis-XYZ` and the binary input processing methods, can be attributed to the size of coordinates stored

in memory after reading was completed. A visual summary presented in

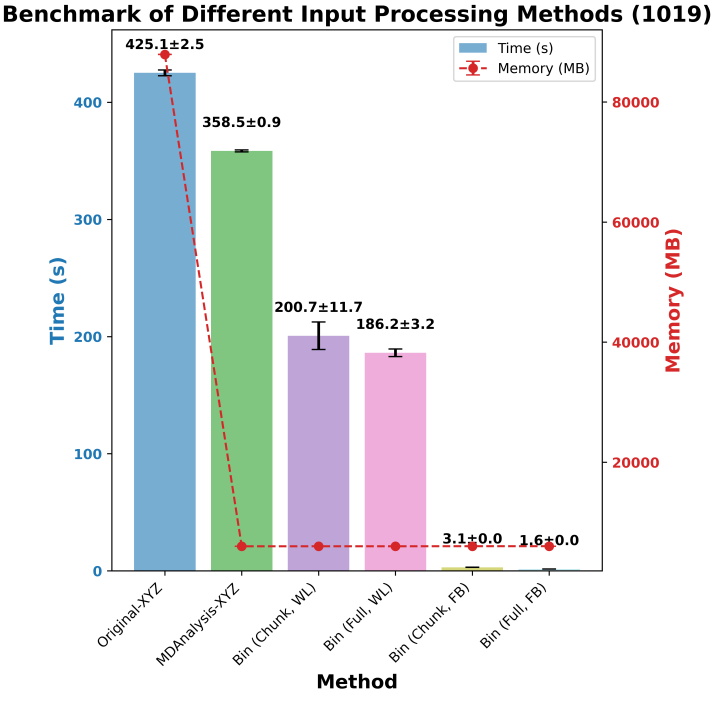


Figure 4.1: Time and memory usage for different file reading methods

figure 4.1, clearly shows that optimized binary reading techniques, particularly `np.frombuffer` implementations, provide substantial performance gains in both time and memory for large-scale input processing. The observed speedups in Table 4.1 underscore the potential impact of using binary data structures to achieve computational and memory efficiency.

4.2 Data Types and `np.frombuffer` Method

In order to understand the time cost of looping on unsigned integer and float32 data types, a comparative analysis was implemented and tested along with the `np.frombuffer` (FB) function.

It was observed that, time cost for processing 5 GB or less of data of any of the aforementioned types using NumPy's `frombuffer` function remained constantly around a few seconds less than 1 second. While using for-loops to process unsigned integer presented a linear growth in time, doing the same with float32 data type presented a log (n)-type time complexity (Figure 4.2, and Appendix C.1).

Since atomic coordinates are mostly stored in either high precision or low precision formats, this analysis clarified why using the `np.frombuffer` function to process binary atomic coordinates with or without chunking per timestep recorded fairly similar processing efficiencies.

It is noteworthy that since this function interprets a buffer as a 1-dimensional

array (34), a conversion may be required if the custom data is not initially in a binary format. Additionally, the use of for-loops in data processing could be costly when dealing with very large arrays of unsigned integers as well as floats.

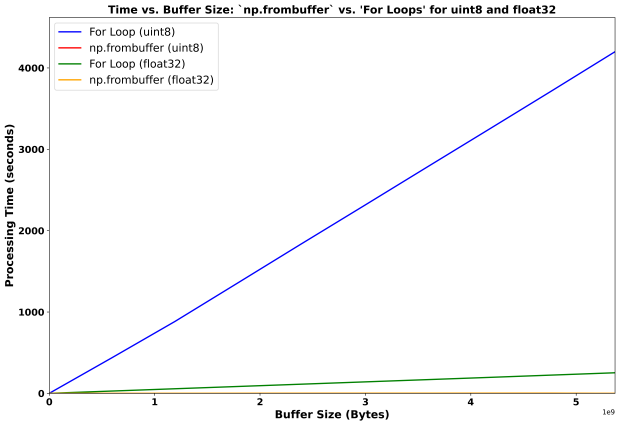


Figure 4.2: Processing times comparison for input processing with a `for-loop` and `np.frombuffer` function. The overlap of the yellow and red lines (brown) shows time versus size invariance of `np.frombuffer` when used to process different input data-types

4.3 Processing Large Data

To test the validity of the binary input processing efficiency of the `np.frombuffer` method for HPC application developments, a larger trajectory was generated through an MD simulation of 8152 water molecules for 2 ns with a 4 fs frequent storage of atomic coordinates in GROMACS. The compressed trajectory was converted to a single .gro file which was subsequently saved in both xyz (129 GB) (see Appendix C.2 for conversion details) and binary (92 GB) (see Appendix B.1.4 for conversion details) formats. For the xyz input, the xyz reader of the MDAnalysis package was chosen for comparison to the binary input processing methods. Again, a visual summary of the results of execution times and memory requirements for each method is presented in figure 4.3

The processing performance of both file formats (xyz and binary) was assessed by measuring the average time, memory usage, and relative speedup (RSU) of different data processing methods. Table 4.2 presents the results of this comparison, which includes the methods tested, their corresponding time (mean and standard deviation), memory usage (mean and standard deviation), and relative speedup relative to the MDAnalysis-XYZ method.

The results clearly show the significant performance improvements achieved with binary input processing methods, particularly when using the chunked and full block binary data reading configurations. Once again, the `Bin (Full, FB)` method demonstrated the highest relative speedup (RSU) of 144.76, processing

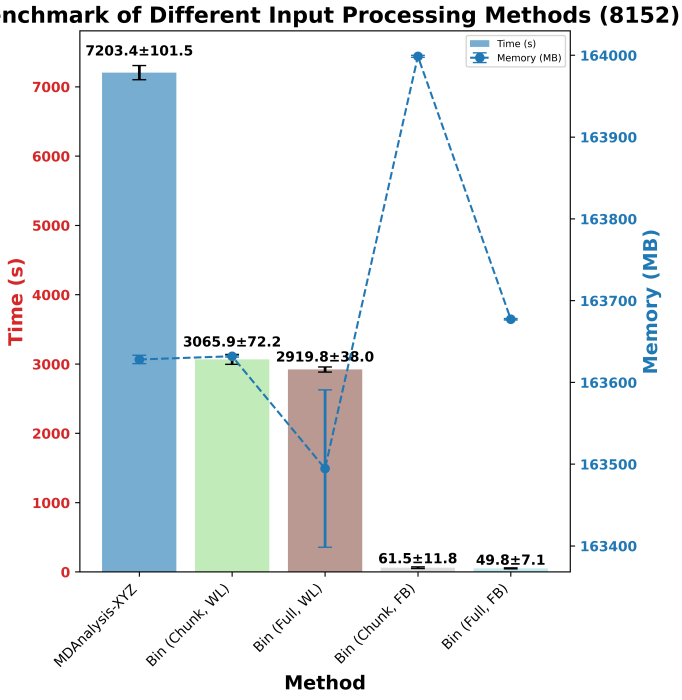


Figure 4.3: Input processing times (bars) and memory usage (dashed line) for MDAnalysis xyz trajectory reader and binary file reading techniques using larger dataset.

Table 4.2: Average Time, Memory Usage, and Relative speedup (RSU) for Large Data Processing Methods

Method	**Time (s) ± SD**	**Memory (MB) ± SD**	**RSU**
MDAnalysis-XYZ	7203.39 ± 101.55	163627.92 ± 5.17	1.0
Bin (Chunk, WL)	3065.86 ± 72.20	163631.91 ± 1.03	2.35
Bin (Full, WL)	2919.83 ± 37.99	163494.54 ± 96.28	2.47
Bin (Chunk, FB)	61.52 ± 11.83	163998.61 ± 1.19	117.10
Bin (Full, FB)	49.76 ± 7.15	163677.13 ± 0.90	144.76

the data in 49.76 seconds on average, compared to 7203.39 seconds for the `MDAnalysis-XYZ` method. This represents an impressive 144.76-fold speedup while maintaining a relatively stable memory usage (163.68 GB mean) for coordinate storage.

Notably, the chunked binary methods also exhibit substantial improvements in both time and memory efficiency. The `Bin (Chunk, FB)` method completed the task in an average of 61.52 seconds, offering an RSU of 117.10 and only slightly higher memory usage compared to the full-block binary method. On the other hand, the looped binary input processing methods; `Bin (Full, WL)` and `Bin (Chunk, WL)`, while offering good speedups compared to `MDAnalysis-XYZ` method, with RSU values of 2.47 and 2.35, respectively, were not as efficient as the FB methods.

Overall, the results strongly suggest that binary data processing, particularly when implemented with the FB method and optimized with full or chunked block reading, provides substantial performance gains for handling large-scale molecular dynamics data, making it a valuable tool for computational studies involving extensive simulation outputs.

4.4 Side Effects of Optimized Input Processing

The input file format from which the atomic coordinates are read has been identified as a one important bottleneck in the angular jump analysis protocol, especially in low-resource (memory-constrained) environments. Converting MD trajectories from the XYZ format to binary format did not only benefit from file size compression but also showed promising potential for reduced memory usage and faster file input read times during post-processing of MD simulation data.

A trade-off worth noting in this case is that, the time taken to arrive at the compressed format, initially incurs overhead but leads to substantial performance gains in subsequent postprocessing stages and eliminates the time and disk space requirements for the MDAnalysis package installation. Thus, the choice of file format and read method should be guided by the available computational resources and the specific requirements of the analysis tasks. For instance, while the measurements above were possible due to the use of computer resources with high memory budgets, to this point, it is still unclear how the fastest method can be adapted suitably to the dipole and HH vector extraction tool to ensure efficient execution in both time and space.

This highlighted the need to thoroughly understand the potential side effects associated with the use of binary input processing methods in post-processing tasks. To address this, the full data blocking configurations, `Bin(Full, WL)` and `Bin(Full, FB)`, were adapted to the dipole and HH vector extraction tool. These configurations once again, represent binary input processing using the `struct.unpack_from` function within a for-loop without chunking, and the FB method, which avoids both chunking and for-loops, respectively. Computations were slightly modified to benefit from Numpy vectorization and Multiprocessing. The effects of these modifications were compared to the original implementation to assess their influence on performance. The original script was named `dphhxyz`, while the scripts using the binary file processing methods were named `dphhbinwl` (`binwl` referring to processing binary data with a for-loop and `struct.unpack_from`) and `dphhbinfb` (with `binfb` referring to processing binary data with the FB method).

4.4.1 Comparison of Serial Executions

Run times and memory usage were measured for the script with triplicate serial executions for both the original and optimized algorithms (i.e., using a binary file, processed with either a for-loop and the `struct.unpack_from` method or Numpy's FB method for coordinates processing). The original script was also measured,

The execution times of the original `.xyz` dipole and HH processing script (`dphhxyz.py`) were 4396.91, 4488.53, and 4335.10 seconds, with an average

of 4406.51 seconds and a standard deviation of 75.81 seconds. The average memory usage for this method was 117073.23 MB with a standard deviation of 4.35 MB.

The execution times for the optimized vector extraction tool with a binary coordinate processing method implemented with the `struct.unpack_from` and a for-loop (`dphhbinwl.py`) were 1630.94, 1638.09, and 1559.41 seconds, with an average of 1609.48 seconds and a standard deviation of 39.72 seconds. The average memory usage for this method was 85.18 MB with a standard deviation of 0.13 MB.

The execution times for the optimized vector extraction tool making use of the fastest coordinate processing method (`dphhbinfb.py`) were 1597.22, 1552.48, and 1568.04 seconds, with an average of 1572.58 seconds and a standard deviation of 7.77 seconds. The average memory usage for this method was 84.97 MB with a standard deviation of 0.04 MB.

A visual comparison of execution times and memory usage for the three implementations is shown in Figure 4.4. It is evident from the

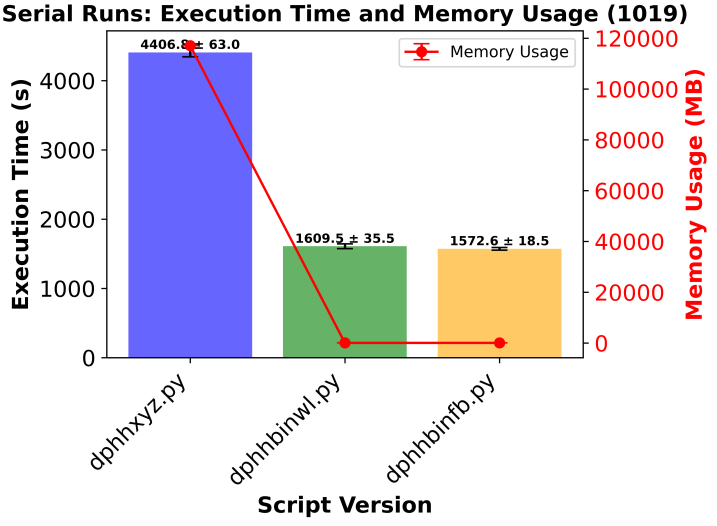


Figure 4.4: Execution times (left y-axis) and memory usage (right y-axis) comparison between the original and optimized DP-HH vector extraction algorithms.

data that both binary processing methods, whether using the for-loop and `struct.unpack_from` or FB, offered significant performance improvements over the original `.xyz` processing method. Specifically, the binary methods reduced both execution time and memory usage drastically.

The vector extraction tool using a binary method with a for-loop (`Bin(Full, WL)`) showed a reduction in execution time by approximately 64.5% compared to the original `.xyz` processing method. Its memory usage was also drastically reduced by about 99.9%.

Similarly, the custom binary trajectory processing method with FB (`Bin(Full, FB)`) demonstrated a reduction in execution time by approximately 64.3% compared to the original method. Memory usage is slightly lower than

the binary with-loop method (by about 0.25 MB), which further emphasizes the efficiency of `np.frombuffer` in handling large datasets.

Both scripts using binary input processing methods outperformed the custom `.xyz` processing method in terms of both time and memory, but the updated version of the vector extraction tool adapted to the FB method for coordinate processing is slightly more efficient in terms of memory usage. This reduction in memory demands may be attributed to a bypass of list storage, improved memory alignment of data, cache locality and efficient execution of tasks with the atomic coordinates yielded into NumPy arrays. To verify this observation, system level performance statistics were taken for the `dphhxyz.py` and `dphhbinfb.py` scripts using `perf`.

The performance counters revealed substantial differences in processing behavior between the original `.xyz` method and the optimized `np.frombuffer` binary method. The following were revealed in contrast:

- **Cycles and Instructions:** The `dphhxyz.py` script recorded approximately 14.5 trillion CPU cycles and executed 37.1 trillion instructions, achieving about 2.57 instructions per cycle (IPC). In contrast, `dphhbinfb.py` utilized only 6.4 trillion cycles and executed 16.2 trillion instructions, resulting in a slightly lower IPC of 2.51. This indicates that the optimized method achieves faster processing by reducing the number of cycles and instructions needed to handle the data, even though the instructions per cycle are similar in both cases.

- **Cache Behavior:** The optimized binary method also demonstrates improved cache efficiency. `dphhxyz.py` incurred 34.4 billion cache references with 8.6 billion cache misses, leading to a relatively high cache miss rate of 24.92%. In contrast, `dphhbinfb.py` had only 7.8 billion cache references and 1.7 billion cache misses, resulting in a lower cache miss rate of 21.59%. Lower cache misses generally contribute to faster data processing, as fewer delays occur from retrieving data from the main memory instead of the cache.

- **Branches and Branch Misses:** Branches represent decision-making points in the code. `dphhxyz.py` had approximately 7.6 trillion branches with a branch miss rate of 0.18%, whereas `dphhbinfb.py` recorded 3.3 trillion branches with a branch miss rate of 0.28%. The optimized binary method shows fewer overall branches, likely due to reduced complexity in loop handling and data retrieval, showing reduction of the need for conditional statements. These can be attributed to exception and error handling conditionals included for debugging.

- **Task Clock and Elapsed Time:** The optimized `np.frombuffer` method demonstrates significant improvements in time efficiency. The task clock (which reflects the time the CPU actively spent processing tasks) for `dphhbinfb.py` was approximately 1.94 seconds, versus 4.36 seconds for `dphhxyz.py`. This nearly doubles the effective CPU usage, as shown by a higher CPU utilization rate (1.245 vs. 1.001). Consequently, the elapsed wall time (i.e., the real-world time taken) was reduced to approximately 1557.70 seconds from 4361.29 seconds—a time savings of approximately 64%.

The rationale for algorithmic efficiency can thus be satisfactorily summarized as follows: the increased performance of vector extraction tool benefiting from the `np.frombuffer` method's efficiency, as well as NumPy vectorization, stems from lower cycle counts, fewer cache misses, and a reduced number of branches, which collectively accelerate the processing time. The reduced cache misses suggest better memory alignment and cache locality, likely due to the streamlined data structure efficiently processed by the `np.frombuffer` approach, avoidance list storage to handle large data more effectively in a single, continuous block. This efficiency makes it especially suitable for memory-intensive molecular dynamics post-processing tasks, where computational resources can be optimized to achieve both speed and memory savings.

4.5 Effects of Optimized Input Processing on Data Parallelism

The impact of efficient file reading methods and the use of multiprocessing for parallel execution, on the performance of the DP and HH vector extraction protocol was evaluated. Two methods were tested: Bin (Full, WL) and Bin (Full, FB), both of which loaded the entire binary file into memory but differed in their coordinate decoding and processing per timestep.

Task processing were subsequently managed by the process pool spawned through the multiprocessing.Pool object. Task here refers to a process executing a call to the binary processing function, dipole and HH vector computation on data and writing of output vectors to file.

For each timestep, all atomic coordinates are processed concurrently by spawned parent-child processes. The output from computation are then mapped in the order in which the input was processed, and pickled to a sub-process which then un-pickles and writes to the respective files. The map and imap methods were also compared for the process pool count delivering the fastest execution, since the imap implementation was perported to be much slower in some certain cases (see author(32) documentation in Appendix C.3).

It was observed that both methods produced identical output content; however, using `pool.map(*func, iterables, chunksize=None)` increased memory requirements significantly. Specifically, when running the parallelized vector extraction tool with 8 processes, the `imap` method required 376.49 seconds and 89.26 MB of memory, while the `map` method required 520.59 seconds and 9036.67 MB.

This result shows that while the `imap` approach can sometimes be slower in certain contexts, it was more efficient in this case, particularly in terms of memory usage. Based on this finding, benchmarking was performed with the original binary dataset for both scripts using the `imap` functionality. These findings highlight the effectiveness of the `imap` approach in managing memory usage while maintaining efficient parallel execution. With these results in mind, we further investigated the scalability of each vector extraction method, which refers to how effectively the task execution time decreases as the number of processes increases. this was done with the `imap` functionality. Ideally, a linear scaling would mean that doubling the number of processes would halve the task execution time, showing a proportional relationship between resources and

performance.

However, real-world parallel algorithms often experience **diminishing returns** in scaling, where increases in process count yield progressively smaller improvements. This phenomenon can arise due to factors such as increased overhead in managing processes or limited parallelizable portions of a task.

4.5.1 Scaling with Increasing Worker Counts

The serial algorithm was adapted to a multiprocessing approach, utilizing a pool of worker processes (automatically regulated CPU cores) to execute tasks on mapped iterables. Each process operates as an independent instance, comprising a copy of the data, code, and assigned computing resources (worker(s)). It was observed that both approaches scaled well with decreasing task execution times with increasing processes but the DP-HH protocol optimized to use the `np.frombuffer` method, produced higher speedup especially with 8 processes after which there was a plateau, indicating less significant gains with increasing number of processes above 8 (Figure 4.5) indicating that additional processes contributed less to speedup.

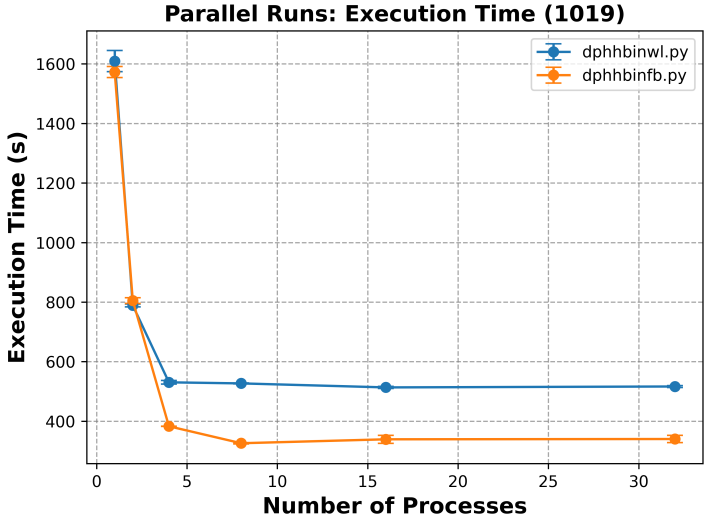


Figure 4.5: Parallel execution times comparison for optimized DP-HH vector extraction algorithms.

4.5.2 Parallel Efficiency

Parallel efficiency for both algorithms was observed to be highest with 2 processes, but with 4 processes efficiency dropped significantly for `dphhbinwl` script while the opposite was observed for the `dphhbinfb` script (Figure 4.6). The significant drop in parallel efficiency for the `dphhbinwl` script with 4 processes could be attributed to inefficiencies in its looping structure. Loops may lead to increased cache misses and memory contention, especially when multiple processes compete for shared resources. On the other hand, vectorized

NumPy operations associated with the `dphhbinfb` script, potentially clarifies improved parallel efficiency with additional processes.

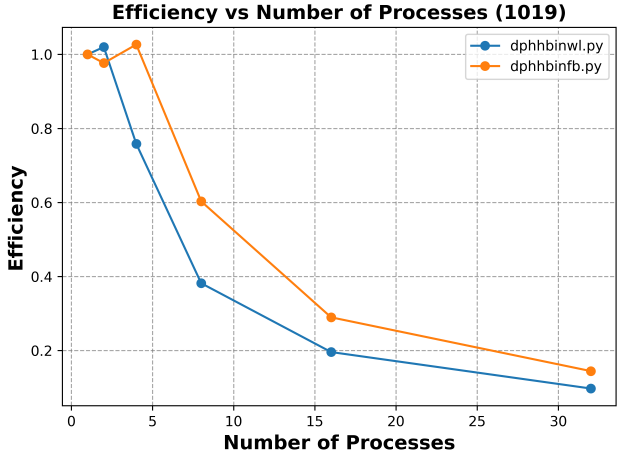


Figure 4.6: Parallel efficiency estimation and comparison between DP-HH vector extraction using a binary file processed with a for-loop and the `struct.unpack_from` method versus Numpy's `np.frombuffer` method.

4.5.3 Memory Usage with Multiprocessing:

Memory usage monitoring across multiple processes revealed a fairly stable pattern for each of the efficient algorithms with negligible growth across multiple processes. From the observations above, it is conceivable that combining xyz file conversion to binary and efficient binary file reading as well as appropriate multiprocessing methods, reduces execution times and memory usage altogether for the original dataset.

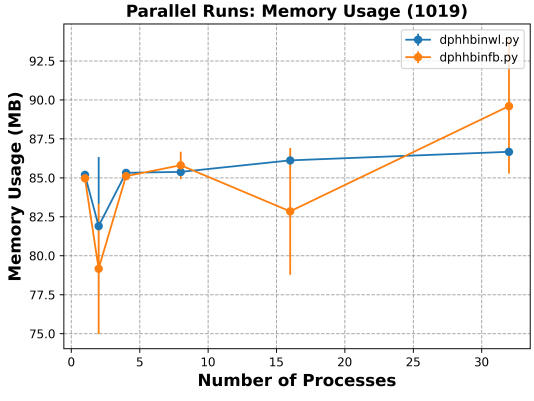


Figure 4.7: Impact of multiprocessing on memory usage in the optimized DP-HH vector extraction algorithms.

4.5.4 Performance Analysis of the Parallel Vector Extraction Tool

Performance statistics were generated for the most efficient parallel approach to inspect hardware related execution efficiency. Single-run parallel executions using multiprocessing pool (imap) with process counts of 4, 8, 16 and 32 processing elements (NPES) were measured initially, followed by using a count of 5. The performance of the dipole-HH vector extraction tool under varying levels of parallelization by analyzing execution time, memory utilization, CPU usage, and various performance counters, assessed whether true parallelism is achieved with the use of the `multiprocessing` package and whether there is an upper limit to the efficiency gained from increasing the process count. The output data was then summarized in the Appendix Table C.1.

Execution Time and Memory Utilization The execution time decreases significantly when moving from 4 to 5 processes (from 415.63 seconds to 358.21 seconds), suggesting that parallel processing effectively reduces the workload. However, beyond 5 processes, the execution time reduction becomes less pronounced. By NPES = 32, execution time only drops to 335.55 seconds. This plateau suggests that additional parallelism does not further optimize performance and instead hits a saturation point. Maximum memory usage decreases substantially when switching from a `pool.map` configuration to `pool.imap` (NPES = 4*) with 4 or more processes, stabilizing around 100–140 MB. This improvement indicates effective memory management through parallel execution.

True Parallelism and CPU Utilization CPU utilization increases with the number of processes, peaking at around 5 CPUs used when NPES = 5 and beyond. The metric "CPUs utilized" reflects the average number of cores actively engaged in computation during the task's duration. As the NPES increases beyond 5, CPUs utilized slightly increases but does not scale linearly with the process count, suggesting limited benefit from additional parallel processes. This limitation implies that true parallelism is achieved up to a certain level, and further increases in NPES do not translate into proportional gains, likely due to core contention or inherent limitations in the tool's parallelization efficiency.

Cache References and Cache Misses The cache reference and cache miss counters reveal additional insights. With NPES = 5, cache references reach approximately 5.69 billion, while cache misses hover around 0.92 billion, with an acceptable cache miss rate (around 16%). However, as NPES increases, cache misses increase disproportionately to cache references, reaching 4.07 billion misses at NPES = 32 with a cache miss rate above 57%. The elevated miss rate suggests that excessive parallelism introduces more contention for cache resources, degrading efficiency by increasing the time spent retrieving data from main memory.

Context Switches and Migrations Context switches (CS) and migrations provide insights into the scheduling overhead. With 5 processes, CS events occur at a manageable rate, around 1,225 per second. However, by NPES = 32, the CS rate drops to approximately 948 per second, indicating the system's ability to distribute processes with fewer interruptions. Migrations, representing the frequency with which processes switch cores, remain low at NPES = 5 (727) but increase significantly at NPES = 32 (37,222). High

migration rates at higher NPES values imply that processes frequently shift between cores, resulting in additional scheduling overhead and potential cache coherence issues, which detract from true parallelism.

Saturation Point and Serialization in Output Processing The NPES = 5 setup appears to be the point of optimal performance for this tool, as increasing NPES beyond this threshold does not proportionally reduce execution time. This saturation is likely due to the tool's structure: while computational execution parallelize effectively on data, output processing is subject to serialization constraints, resulting in an $\mathcal{O}(N \cdot T)$ complexity for data writing. This serial aspect in I/O operations becomes a bottleneck, particularly when more processes attempt concurrent writing to the output. As NPES increases, serialization limits the performance benefit of parallel computation, evidenced by the plateau in execution time reductions.

Overall, the analysis of the `perf` statistics suggest that while the dipole-HH vector extraction tool achieves effective parallelization up to NPES = 5, it encounters diminishing returns beyond this point. Full optimization is achieved at NPES = 5, with additional processes causing increased contention for CPU and cache resources and elevating context-switching and migration rates. The bottleneck is likely due to serialization in output processing, which constrains performance gains from increased parallelism and prevents full utilization of additional CPU resources. For future optimization, addressing the serialization in output handling could further enhance parallel efficiency and reduce execution times at higher NPES values.

4.5.5 Testing Scaling Limits with Larger Dataset

In order to understand the limits of the algorithmic changes enabling faster execution and efficient memory usage, the second dataset, was used for benchmarking to identify potential performance limits that may be presented using big MD outputs. A finer grain of time and memory usage measurement was implemented. Apart from measuring the total program execution time, the output processing loop (output) was timed separately. It was observed that saturation was reached quickly, likely due to output processing time requirements. This is because, the output processing loop adopts serialization from un-pickling of computed vectors communicated through pickling from all processes, before writing them to the respective output files. As such a barrier is created when the file writing is not complete. This results in contention for RAM space to store computed data, until they are written out to file. This led to the drastic increase in memory usage beyond 5 processes (Figure 4.8), suggesting again that the most efficient optimization for the dipole and HH vector extraction tool, is attainable with a multiprocessing pool of 4 or 5 processing elements for small and large MD output. Beyond these limits, execution inefficiencies are inevitable. This observation is further corroborates by `perf` statistics recorded earlier for multiprocessing pool using 4, 5, 8, 16 and 32 processing elements.

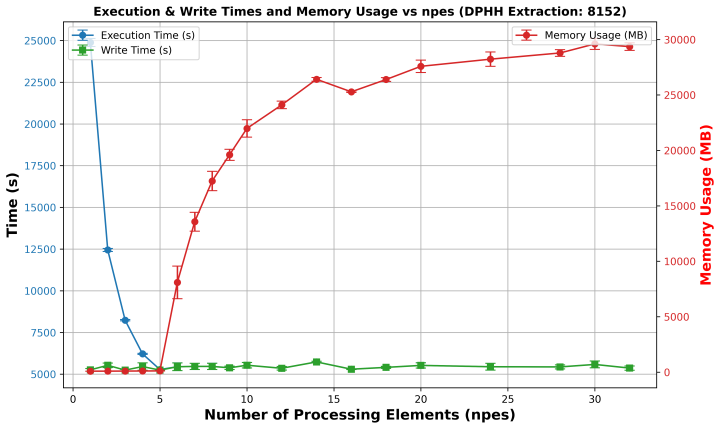


Figure 4.8: Benchmark of dipole and HH vector extraction tool using larger binary input, showing constant time requirement for output processing

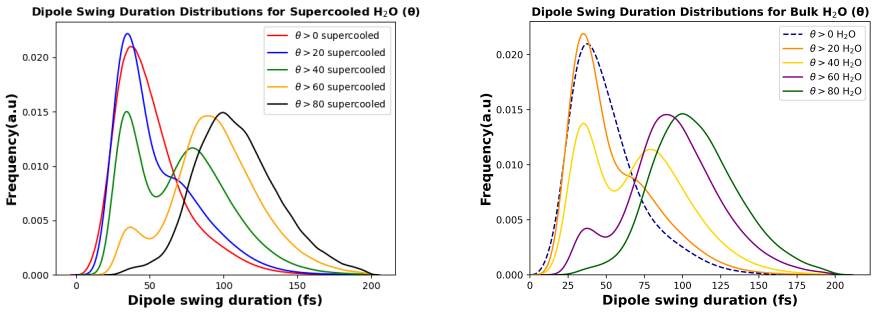
4.6 Computation and Plotting of Angular Jumps

In order to validate the precision of the optimized DP-HH extraction protocol, the generated
`DP_space_LAMMPS_10m.dat` file, which was approximately 2.4 GB in size, was used as input for the swing duration and frequency plot algorithm. The output of this algorithm, the `tbs_DP_n.txt` file, was subsequently used to assess the performance of the new protocol. This comparison was essential to ensure that the optimized process not only improved computational efficiency but also maintained the integrity and accuracy of the results.
Additionally, due to the deprecation of the `distplot` method from the seaborn package, which was used in the original script, the `kdeplot` method was adopted as a suitable replacement. The plot generated using `kdeplot` confirmed that the optimized protocol produced results consistent with those reported in earlier studies, demonstrating that optimizations did not compromise the quality or reproducibility of the analysis.

4.6.1 System Architecture Effects

It is noteworthy that the above analysis outputs were generated using the Leonardo Supercomputer, thus execution times and memory usage could vary significantly, however, maintaining a similar pattern if carried out on the Argo Supercomputer.

This is mainly due to the hardware architecture and performance difference. For instance, memory limits on the Argo supercomputer will present a significant challenge for profiling the original DP-HH extraction tool.



(A) **(B)**

Figure 4.9: Plot of dipole swing duration vs frequency as implemented from (A) original analysis protocol and (B) protocol with optimized algorithms.

4.7 Optimization of Defects Analysis Tool

The defects analysis tool was initially designed to compute defects in the hydrogen-bonding network around water molecules separately from identifying the indices of molecules with detected defects. Since these two outputs involved similar computational complexity, the original code was refactored to generate both outputs in a single run, thereby reducing redundancy and improving efficiency.

The optimized binary input processing method, previously applied to the dipole-HH vector extraction tool, was leveraged here to handle atomic coordinates in the defects and indices analysis tasks. As detailed earlier, the matrix generation step had quadratic computational complexity, while the remaining tasks operated at a linear complexity level. The refactored code was executed in serial, both with and without the application of Numba JIT (Just-In-Time) compilation for the primary functions (tasks), and performance was evaluated over the first 20 frames (see Figure 4.10).

The results showed a significant decrease in execution time when Numba JIT compilation was applied to all major tasks. For the full dataset of 12,500 frames, the execution time to process both defects and indices was approximately 21 hours without Numba JIT decoration. With task decoration, this was reduced to around 5 hours, achieving a speedup of approximately 4.25 times with low memory requirements.

The execution configuration was further modified to support computation and output processing for custom frame ranges, enabling greater flexibility in analyzing subsets of frames (as used by plotting tools). Additionally, a checkpoint functionality was implemented to allow continuation of the defects analysis following errors or crashes when processing very large frame counts (exceeding 125,000). These enhancements improve both reliability and scalability, making the tool more robust for extended analyses during serial execution.

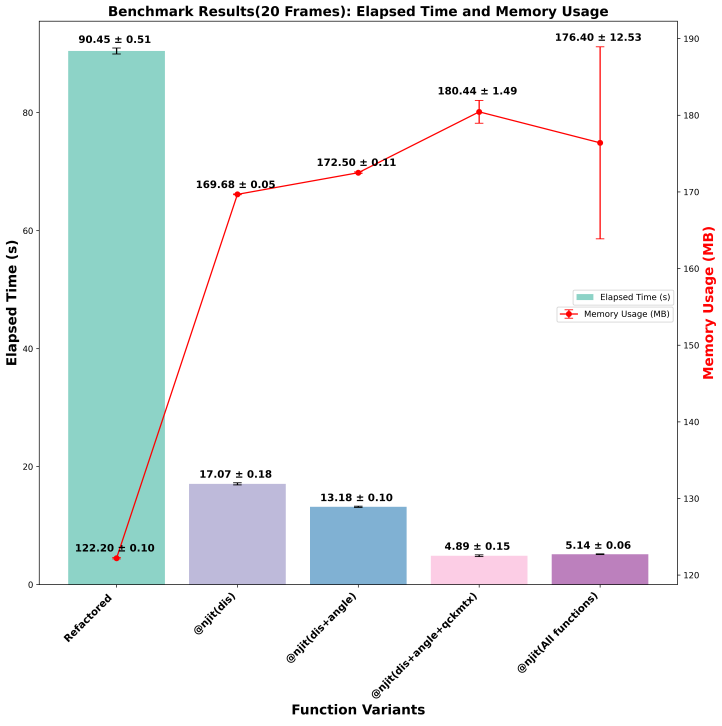


Figure 4.10: Execution time comparison for the defects analysis tool with and without Numba JIT decoration across the first 20 frames.

4.8 Data Parallelism with Joblib

To further reduce execution time, we explored parallelization using Joblib (35). Joblib enhances Python's multiprocessing capabilities by enabling lightweight pipelining, transparent disk-based caching, and optimized memory management for NumPy arrays, which are commonly used in scientific computing. Joblib, built on the Loky library, addresses limitations in Python's multiprocessing by using `cloudpickle`, allowing functions to be pickled even when defined in interactive scopes. For parallel computing (36), Joblib's `Parallel` class and `delayed` decorator are employed. The `Parallel` class creates a process pool, similar to the `multiprocessing` pool used in earlier sections. Here, we use `delayed` to wrap our target function so it can be applied to the `Parallel` object (data) via an iterator. The `Parallel` class accepts parameters such as `n_jobs`, which controls the number of processes, and `verbose`, which provides debugging information. Additional options allow for timeout settings, switching between threads and processes, and memory mapping configurations to enhance performance in specific cases. By batching function calls, `Parallel` executes each `delayed` function with its corresponding arguments, enabling efficient task management and further reducing execution time. A strong scaling was observed for execution time and memory usage while performing defects analysis on 125000 frames. The serial execution ($n\_job = 1$) required 18567.44s ± 43.73 seconds (5.16 hours) and 12.6 GB of memory. In contrast, the most efficient

execution was observed for 32 processes ($n\_job = 32$) which required 697.23 $\pm$ 12.82 seconds and 6.3 GB of memory, representing a speedup of 26.6-fold in time and half the memory requirements relative to the serial job (Figure 4.11).

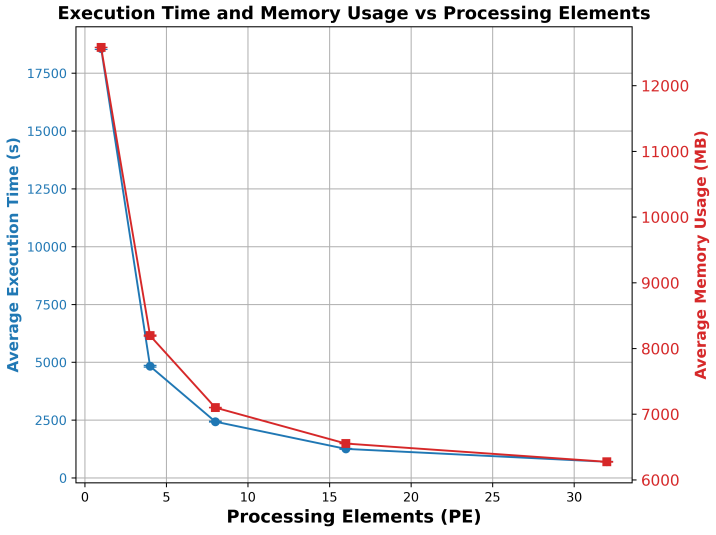


Figure 4.11: Execution time comparison for the defects analysis tool with Numba JIT decoration, and data parallelism across 1 to 32 jobs (processes) using joblib. A strong scaling is observed for execution time (left y-axis) and memory usage (right y-axis).

4.8.1 Performance Gains for Parallel Defects Analysis Tool

Performance gains from parallelization, were examined with `perf` statistics of performance metrics across different numbers of processing elements (PE). The data shown in Appendix Table C.2 demonstrates how critical system-level metrics evolved to show extent of increased use of compute power in parallelism.

Cycles and Instructions: The total number of cycles and instructions remained relatively constant across different PE configurations, with cycles varying between 6.14×10^{13} and 6.37×10^{13}, and instructions hovering around 1.44×10^{14}. This indicates that the overall computational work required does not drastically change with parallelism, as expected from an optimized parallel algorithm.

Cache Performance: Cache references showed slight reduction from 7.50×10^{10} for 32 PE to 7.32×10^{10} for 1 PE, and cache misses decreased marginally with more processing elements, from 2.94×10^{10} at 32 PE to 3.91×10^{9} at 1 PE. This suggests that parallelization improves the effective utilization of the cache and reduces memory bottlenecks, likely due to better data locality and task scheduling in parallel execution through the Joblib interface.

Branch Prediction: Branch instructions and branch misses remained largely stable across configurations (Figure 4.12). This reflects that the computational complexity involving branching does not introduce significant

overhead with increased parallelism, indicating an efficient handling of conditional logic (e.g if-statements) during the analysis.

Task Clock: Comparing the serial execution up to the most efficient in time and space, task clock times were clearly observed to be inversely proportional to the number of processing elements, confirming the estimated speedup with increasing processes (Figure 4.12). That is, for 32 PE, the task clock time is $2.08 \times 10^7 \pm 209,166.32$ seconds, reducing to $1.85 \times 10^4 \pm 167.78$ seconds for a single PE. This shows that the more resources are available, there is a relative proportion of tasks being executed on the data, with less resource being wasted, underscoring the overall gains from parallel execution.

Faults and Minor Faults: Both faults and minor faults decrease proportionally as the number of PE increases, aligning with the expectation that a more distributed workload reduces the incidence of fault conditions that might arise from resource contention or inefficient memory access patterns. This is indicated by a complete overlap between fault and minor fault in Figure 4.12

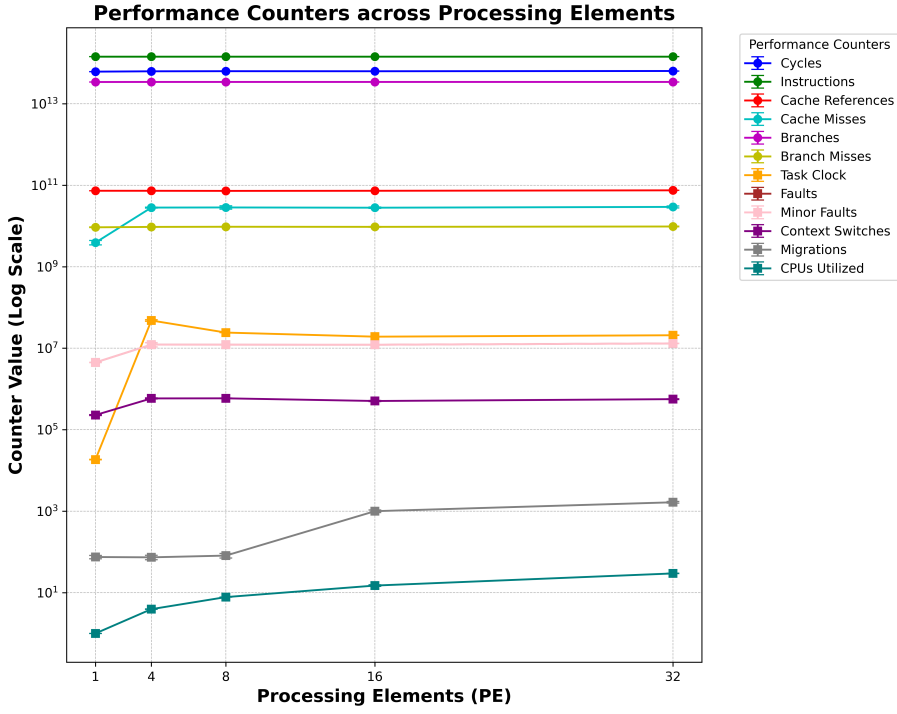


Figure 4.12: Summary of performance metrics data obtained for the optimized hydrogen bonding defects and molecular indices analysis tool. Complete overlap between Faults (brown) and Minor Faults (pink) values reduced line count in plot form 12 to 11

Context Switches and Migrations: Context switches and migrations indicate how often the system moved processes between CPUs. How the program is halted in order to wait for a kernel operation to finish (such as I/O), to let other applications run, or to move execution to another CPU core can be analyzed with this combination. When a context switch happens, the

program's execution is halted and another program is allowed to run instead. This is a very time-intensive task. As expected, 32 PE shows the highest number of context switches (562,154 ± 13,040.76), with a fairly noticeable drop as the number of PE decreases (Figure 4.12). This is consistent with the system's attempt to allocate more resources to each task when fewer CPUs are available.

CPU Utilization: The number of CPUs utilized aligns with the expected scaling behavior, from 29.67 ± 0.23 at 32 PE to 1.00 ± 0.00 at 1 PE. This shows that with more processing elements, CPU utilization is maximized, leading to more efficient use of available hardware resources (Figure 4.12.

In summary, the performance counters support the conclusion that Joblib's parallelization provides significant improvements in execution time and memory efficiency for defects analysis. As the number of processing elements increased, the system exhibited better utilization of cache and CPU resources, along with a more effective distribution of tasks, resulting in the observed performance gains.

Chapter 5

Conclusions and Recommendations

5.1 Summary

Molecular reorientation events in water previously studied by Offei-Danso and coworkers (1) using an automatized protocol implemented in parts with FORTRAN and MATLAB, provided insights into the collective burst mechanism of angular jumps in MD simulated liquid water. Translation of the automated protocol into Python, was to ensure a self-contained framework for interactive post-processing of larger MD trajectories of water molecules, leveraging efficient Python library functions.

Performance bottlenecks have been confirmed from codelet profiling to have emerged from resource constraints, especially in memory, which was fundamental for high latency, slow execution, and frequent crashes due to program instability. Probing this further provided opportunities for the optimization of the post-processing protocol. Optimization goals were to iteratively profile and identify computational bottlenecks, implement efficient versions with an HPC approach, such that optimization choices leads to efficient program execution with efficient memory usage while lowering execution time.

To this point, we demonstrate an alternative approach for efficient MD input processing by converting custom xyz trajectory into a continuous stream of bytes (binary format) and leveraging NumPy's capabilities. Adapting this to both dipole (DP) and hydrogen-hydrogen (HH) vector extraction and the hydrogen bonding defect analysis tools, led to optimization choices that ensured performance gains involving faster execution and high memory efficiency.

Combining efficient input processing, to data parallelized multiprocessing executions, it was observed that for the DP and HH vector extraction protocol, 4 or 5 processing elements are required to deliver the best performance for both small and large MD datasets, with respect to the system architecture cited (19). Of note, it is not clear from the scope of this work, how other computational system architectures would affect execution times and memory usage, however, a similar profile is expected.

Refactoring the defect analysis protocol to have it adapted to efficient input processing, together with NumbaJIT decorated functions, also lead to a 4-fold

speedup in serial execution for 125000 frames. Data parallelism implementation using Joblib's Parallel and delayed packages, further led to a 25-fold speed-up using 32 processing elements. Again on the cited system architecture, combining both analysis tools (see Appendix D.1) required 1055.57 seconds (17.59 minutes) to process DP and HH vectors (2.4 GB output file each) as well as hydrogen bonding defects (486 MB file) and the respective indices (3.2 GB file), from the 1 ns (125000 frames) trajectory.

Performance statistics and the measurement of time and memory usage during execution provided insights into resource usage and parallel efficiency. Quality checks were essentially carried out on only angular jumps which had a complete plotting utility code updated to avoid deprecation warnings found during the execution of the original implementations.

5.2 Recommendations

Having identified and eliminated computational bottlenecks of the two major tools of the protocol, it is recommended that output processing tools, such as, time series analysis and plotting utilities which rely on the outputs of the DP and HH vectors, swings due to molecular dipoles, swings due to hydrogen-hydrogen vectors, and hydrogen bonding defects and indices tools are profiled to identify other optimization opportunities.

Additionally, it is recommended that codelets constituting a completely optimized protocol are continuously integrated and tested on smaller and larger datasets to further identify unforeseen side effects.

The use of other Python libraries, and heterogeneous computational approaches (e.g. GPU acceleration), as well as multi-node parallel configurations (such as using Dask and Ray, as the parallel back-end of Joblib) can be implemented to further enhance computational efficiencies here-in reported, are highly recommended.

Appendices

Appendix A

Appendix for Chapter 1

A.1 Profiling Original Algorithm:

The original Python script provided below was designed to process molecular dynamics data from an XYZ file. This script performs tasks including reading atomic coordinates, calculating dipole and hydrogen-hydrogen (HH) vectors, and writing the results to output files. To gain insights into potential performance bottlenecks, timing and memory usage monitoring were included in the script. The script was subsequently run on the Leonardo Supercomputer trice and results appended below the script as follows:

```
from memory_profiler import memory_usage
import numpy as np
import time

def main_script():
    # Analyze data every 4fs from the datafile (1fs interval)
    t_step = 4
    t_start = 500

    # Correct coordinates
    def pbc(x, box_size):
        x = x - box_size * np.round(x / box_size)
        return x

    # Read .xyz file
    file_path = '../input/water_lmp_3057_10-001_first.xyz'
    xyz_file = open(file_path)
    start_time = time.time()
    N = xyz_file.readline()
    header = xyz_file.readline()
    atom_symbol, coords = ([] for i in range(2))

    for line in xyz_file:
        try:
            atom, x, y, z = line.replace("  ", " ").split(" ")
            coords.append([float(x), float(y), float(z)])
        except:
            n = line.split("  ")
    xyz_file.close()
    dp_list = []
    hh_list = []
    coords_np = np.array(coords)

    num_molecul = 1019  # number of Oxygen molecules
    box_size = 31.197

    # Writing our dipole angles
    for j in range(0, round(coords_np.shape[0] / (3 * num_molecul))):
        for i in range(num_molecul):
            oh1 = coords_np[num_molecul*j*3+i]
                  - coords_np[num_molecul*(3*j+1)+2*i]
            oh2 = coords_np[num_molecul*j*3+i]
                  - coords_np[num_molecul*(3*j+1)+2*i+1]
```

```
            oh1 = pbc(oh1, box_size)
            oh2 = pbc(oh2, box_size)

            dp = oh1 + oh2
            hh = oh1 - oh2
            dp_list.append(round(dp[0], 3))
            dp_list.append(round(dp[1], 3))
            dp_list.append(round(dp[2], 3))

            hh_list.append(round(hh[0], 3))
            hh_list.append(round(hh[1], 3))
            hh_list.append(round(hh[2], 3))
    dp_np = np.array(dp_list)
    hh_np = np.array(hh_list)

    # Open file in write mode
    with open('../output/DP_space_LAMMPS_10_first.dat', 'w') as fp:
        for item in dp_list:
            # Write each item on a new line
            fp.write("%s  " % item)
        print('Done DP_space_LAMMPS_first.dat')

    # Open file in write mode
    with open('../output/HH_space_LAMMPS_10_first.dat', 'w') as fp:
        for item in hh_list:
            # Write each item on a new line
            fp.write("%s  " % item)
        print('Done HH_space_LAMMPS_first.dat')

    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f'Elapsed time is: {elapsed_time:.2f} seconds')

if __name__ == "__main__":
    # Monitor memory usage and run the main script
    mem_usage = memory_usage((main_script,), interval=0.1, retval=False)
    # Print memory usage statistics
    print(f"Peak memory usage: {max(mem_usage):.2f} MB")

RESULTS
Reading Original-XYZ file and processing the DP_HH vetor in serial
Run 1
Elapsed time is: 4396.91 seconds
Peak memory usage: 117073.31 MB

Run 2
Elapsed time is: 4488.53 seconds
Peak memory usage: 117065.26 MB

Run 3
Elapsed time is: 4335.10 seconds
Peak memory usage: 117074.12 MB
```

Appendix B

Appendix for Chapter 3

B.1 Detailed Descriptions

B.1.1 XYZ File Reading

Two distinct methods were explored for reading XYZ files:

B.1.2 Custom xyz File Reading

This method involves reading the XYZ file line-by-line using a custom function. The approach is relatively straightforward, where each line is processed to extract atomic coordinates and store them in a NumPy array. This method is based on a serial approach, which processes one line at a time and converts the data into a list of coordinates. The primary advantage is its simplicity and ease of implementation. However, its efficiency, especially for larger files, needs to be evaluated to determine if it meets performance requirements.

```
def read_file_serial(file_path):
    with open(file_path) as xyz_file:
    # Read the number of atoms
    # (not used in this function)
        N = xyz_file.readline()
        # Read the header line
        # (not used in this function)
        header = xyz_file.readline()
        atom_symbol, coords = ([] for i in range(2))

        for line in xyz_file:
            try:
                atom, x, y, z = line.replace("  ", " ").split(" ")
                coords.append([float(x), float(y), float(z)])
            except:
                # Handle lines that don't
                # contain the expected data format
                n = line.split("  ")
    return np.array(coords)
```

This method is simple and effective for smaller files or quick checks. However, its performance might degrade with very large files due to the overhead of repeated file operations and Python list manipulations. It also lacks built-in error handling for corrupted or irregular file formats, which could lead to issues if the file structure is not strictly followed.

B.1.3 Reading with MDAnalysis

This method leverages the MDAnalysis package, which is specifically designed for efficient handling of molecular dynamics trajectories and related file formats. By using MDAnalysis with the option "in_memory=True", the entire trajectory can be loaded into memory at once, and the atomic coordinates can be retrieved directly. This method is expected to offer significant improvements in read times and memory efficiency, particularly for large files or datasets where performance is critical. The use of specialized libraries like MDAnalysis can optimize both speed and reliability.

```
def read_file_mdanalysis(file_path):
# Load the entire trajectory into memory
    u = mda.Universe(file_path, in_memory=True)
    coords = u.atoms.positions  # Retrieve the atomic coordinates
    return coords
```

This approach is well-suited for large-scale data handling and provides a streamlined way to access atomic coordinates. MDAnalysis is optimized for performance and includes additional features for analyzing molecular data, which can be advantageous for more complex workflows. However, it requires the installation of the MDAnalysis package and assumes that the file format is compatible with its parsing capabilities.

B.1.4 XYZ to Binary File Conversion

The following Python function converts an XYZ format file to a binary format file:

```
def xyz_to_binary(xyz_filename, bin_filename):
    """
    Converts an XYZ format file to a binary format file.

    Parameters:
    - xyz_filename (str): Path to the input XYZ file.
    - bin_filename (str): Path to the output binary file.

    Returns:
    - None
    """
    # Open the XYZ file for reading and binary file for writing
    with open(xyz_filename, 'r') as -
        xyz_file, open(bin_filename, 'wb') as bin_file:
        while True:
            # Read number of atoms from the first line
            line = xyz_file.readline()
            if not line:
                break  # End of file
            num_atoms = int(line.strip())
            # Write number of atoms in binary
            # format (4 bytes, unsigned int)
            bin_file.write(struct.pack('I', num_atoms))
            # Read and write header information
            header = xyz_file.readline().strip()
            # Write header length in binary
            # format (4 bytes, unsigned int)
            bin_file.write(struct.pack('I', len(header)))
            bin_file.write(header.encode('utf-8'))

             # Iterate through each atomic coordinates of each timestep
            for _ in range(num_atoms):
                parts = xyz_file.readline().split()
                atom_type = int(parts[0])
                x = float(parts[1])
                y = float parts[2])
                z = float(parts[3])
                # atom data in bin fmt
                #(int = 4 bytes for atom type + 12 bytes for x, y, z)
                bin_file.write(struct.pack('Ifff', atom_type, x, y, z))
```

B.1.5 Binary File Reading

To determine the most effective method for reading and processing atomic coordinates from binary files, several techniques were tested. These methods differ primarily in how they manage memory and the efficiency of processing the binary data.

- **Chunked Reading with Loop:** This method reads the binary file in small chunks and processes each chunk sequentially using a loop. The primary goal of this approach is to manage memory usage effectively, particularly for very large files. By processing the file in chunks, the program only keeps a small portion of the data in memory at any given time, which helps prevent memory exhaustion. However, this method might be slower due to the overhead introduced by repeated input/output (I/O) operations (reading from the file multiple times) and frequent memory allocations (creating space in memory to store each chunk of data).

```
def read_binary_tschnk_withloop(file_path, chunk_size=3057):
    with open(file_path, 'rb') as file:
        while True:
            num_atoms_data = file.read(4)
            if not num_atoms_data:
                break

            num_atoms = struct.unpack('I', num_atoms_data)[0]
            header_len_data = file.read(4)
            if not header_len_data:
                break

            header_len = struct.unpack('I', header_len_data)[0]
            header = file.read(header_len)

            coords = np.zeros((num_atoms, 3), dtype=np.float32)
            for start in range(0, num_atoms, chunk_size):
                end = min(start + chunk_size, num_atoms)
                data = file.read((end - start) * 16)
                if len(data) < (end - start) * 16:
                    break
                for i in range(end - start):
                    atom, x, y, z =
                    struct.unpack_from('Ifff', data, i * 16)
                    coords[start + i] = [x, y, z]

            if coords.size > 0:
                yield coords
```

- **Full Reading with Loop:** In this method, the entire binary file is read into memory at once, and the coordinates are processed for each timestep using a loop. This approach is intended to test whether reading the entire file into memory at once can improve performance by reducing the need for repeated I/O operations. The benefit here is that all the data is available in memory, which can make the processing faster since the program doesn't need to keep accessing the file. However, this method requires a significant amount of memory, as the entire file must fit in memory. If the file is too large and exceeds the available memory, this method may not be practical and could lead to system instability or crashes.

```
def read_binaryfull_withloop(file_path):
    with open(file_path, 'rb') as file:
        all_coords = []
        while True:
            num_atoms_data = file.read(4)
            if not num_atoms_data:
                break

            num_atoms = struct.unpack('I', num_atoms_data)[0]
            header_len_data = file.read(4)
            if not header_len_data:
                break

            header_len = struct.unpack('I', header_len_data)[0]
            header = file.read(header_len)

            coords = np.zeros((num_atoms, 3), dtype=np.float32)
            data = file.read(num_atoms * 16)
            if len(data) < num_atoms * 16:
                break
            for i in range(num_atoms):
                atom, x, y, z =
                struct.unpack_from('Ifff', data, i * 16)
                coords[i] = [x, y, z]

            all_coords.append(coords)
    return all_coords
```

- **Chunked Reading with `np.frombuffer`:** This method also reads the binary file in chunks but uses NumPy's `frombuffer` function to convert the binary data directly into a NumPy array. NumPy is a powerful library for numerical computing in Python, and `frombuffer` allows the program to efficiently convert binary data into an array format that can be easily manipulated. This method is expected to be faster than using a loop to process each chunk because `frombuffer` handles the conversion more efficiently, leveraging NumPy's optimized internal routines. The chunking approach still helps manage memory usage, making it suitable for large files, while the use of NumPy improves processing speed.

```
def read_binary_tsfbchnk(file_path, chunk_size=3057):
    with open(file_path, 'rb') as file:
        while True:
            num_atoms_data = file.read(4)
            if not num_atoms_data:
                break

            num_atoms = struct.unpack('I', num_atoms_data)[0]
            header_len_data = file.read(4)
            if not header_len_data:
                break

            header_len = struct.unpack('I', header_len_data)[0]
            header = file.read(header_len)

            coords = np.zeros((num_atoms, 3), dtype=np.float32)
            for start in range(0, num_atoms, chunk_size):
                end = min(start + chunk_size, num_atoms)
                data = file.read((end - start) * 16)
                if len(data) < (end - start) * 16):
                    break

                coords =
                np.frombuffer(data,
                dtype=np.float32).reshape(-1, 4)[:, 1:]

                if coords.size > 0:
                    yield coords
```

- **Full Reading with `np.frombuffer`:** In this method, the entire binary file is read into memory at once, and `np.frombuffer` is used to decode the atomic coordinates directly. This method is particularly efficient in terms of speed because it minimizes Python's overhead by avoiding explicit loops and taking full advantage of NumPy's capabilities. However, like the full reading method with a loop, this approach requires sufficient memory to hold the entire file. If memory is not a limitation, this method can offer the best performance because it reduces both the number of I/O operations and the processing time needed to convert the data.

```
def read_binary_fbfull(file_path):
    with open(file_path, 'rb') as file:
        all_coords = []
        while True:
            num_atoms_data = file.read(4)
            if not num_atoms_data:
                break

            num_atoms = struct.unpack('I', num_atoms_data)[0]
            header_len_data = file.read(4)
            if not header_len_data):
                break

            header_len = struct.unpack('I', header_len_data)[0]
            header = file.read(header_len)

            data = file.read(num_atoms * 16)
            if len(data) < num_atoms * 16):
                break

            coords =
            np.frombuffer(data,
            dtype=np.float32).reshape(-1, 4)[:, 1:]

            all_coords.append(coords)
    return all_coords
```

Appendix C

Appendix for Chapter 4

C.1 For-Loops and `np.frombuffer` Method

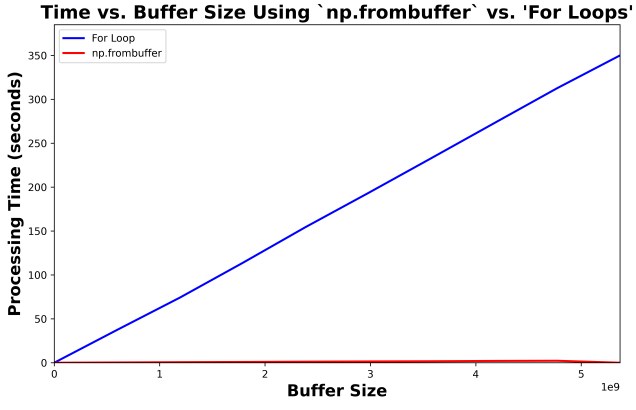


Figure C.1: The execution times it takes for a for-loop approach and `np.frombuffer` method to processes increasing unsigned integer data up to a 5 GB maximum

C.2 GROMOS-87 Trajectory Conversion Script

```
from tqdm import tqdm

def read_gro_frames_custom(file_path):
    with open(file_path, 'r') as file:
        while 1:
            # Read the title line
            title = file.readline().strip()
            if not title:  # End of file
                break

            # Read the atom count line
            atom_count_line = file.readline().strip()
            atom_count = int(atom_count_line)

            # Initialize lists for grouped atom data by type
            oxygen_data, hydrogen1_data, hydrogen2_data = [], [], []
```

```
            # Read the atom data lines and group by atom type
            for _ in range(atom_count):
                atom_line = file.readline().strip()
                parts = atom_line.split()

                # Extract only atom type and coordinates
                atom_type = parts[1]
                # Convert to float for calculation
                coordinates = list(map(float, parts[3:6]))


                # Multiply coordinates by 10 to convert from nm to angstroms
                coordinates = [coord * 10 for coord in coordinates]

                # Rename and group by atom type
                if atom_type == 'OW':
                    oxygen_data.append(f"1 {' '.join(f'{coord:.3f}'
                                    for coord in coordinates)}")
                elif atom_type == 'HW1':
                    hydrogen1_data.append(f"2 {' '.join(f'{coord:.3f}'
                                        for coord in coordinates)}")
                elif atom_type == 'HW2':
                    hydrogen2_data.append(f"3 {' '.join(f'{coord:.3f}'
                                        for coord in coordinates)}")

            # Combine the data in the desired order and yield it
            frame_data = oxygen_data + hydrogen1_data + hydrogen2_data
            yield {
                # Each entry formatted as 'atom_type x y z'
                "atom_count": len(frame_data),
                "title": title,
                "atoms": frame_data

            }

            # Skip the box vector line
            file.readline()

def gro_to_xyz_single_file(gro_file, xyz_file):
    '''
    Convert a single trajectory from .gro to .xyz format.
    Efficient memory handling by using a generator.
    '''
    try:
        with open(xyz_file, 'w') as xyz:
            # Show progress bar using tqdm
            for frame_data in tqdm(read_gro_frames_custom(gro_file),
                                   desc="Converting GRO to XYZ"):
                header = frame_data["title"].strip()
                atom_count = frame_data["atom_count"]

                # Write atom data to the .xyz file
                xyz.write(f"{atom_count}\n{header}\n")
                for atom in frame_data["atoms"]:
                # formatted as 'atom_type x y z'
                    xyz.write(f"{atom}\n")

    except FileNotFoundError:
        print(f"Error: File not found - {gro_file}")
    except Exception as e:
        print(f"An error occurred: {e}")

# main
#  .gro file path
gro_file_path = 'water1019sp.gro'
# output .xyz file path
xyz_file_path = 'water1019spcustom.xyz'
gro_to_xyz_single_file(gro_file_path, xyz_file_path)
```

C.3 The Pool Class

```
class Pool(object):
'''
Class which supports an async version of applying functions to arguments.
'''
_wrap_exception = True

@staticmethod
def Process(ctx, *args, **kwds):
    return ctx.Process(*args, **kwds)

.........(content removed for clarity)

def _setup_queues(self):
    self._inqueue = self._ctx.SimpleQueue()
    self._outqueue = self._ctx.SimpleQueue()
    self._quick_put = self._inqueue._writer.send
    self._quick_get = self._outqueue._reader.recv

def _check_running(self):
    if self._state != RUN:
        raise ValueError("Pool not running")

def apply(self, func, args=(), kwds={}):
    '''
    Equivalent of `func(*args, **kwds)`.
    Pool must be running.
    '''
    return self.apply_async(func, args, kwds).get()

def map(self, func, iterable, chunksize=None):
    '''
    Apply `func` to each element in `iterable`, collecting the results
    in a list that is returned.
    '''
    return self._map_async(func, iterable, mapstar, chunksize).get()

.......(content removed for clarity)
def imap(self, func, iterable, chunksize=1):
    '''
    Equivalent of `map()` -- can be MUCH slower than `Pool.map()`.
    '''
    self._check_running()
    if chunksize == 1:
        result = IMapIterator(self)
        self._taskqueue.put(
            (
                self._guarded_task_generation(result._job, func, iterable),
                result._set_length
            ))
        return result
    else:
        if chunksize < 1:
            raise ValueError(
                "Chunksize must be 1+, not {0:n}".format(
                    chunksize))
        task_batches = Pool._get_tasks(func, iterable, chunksize)
        result = IMapIterator(self)
        self._taskqueue.put(
            (
                self._guarded_task_generation(result._job,
                                              mapstar,
                                              task_batches),
                result._set_length
            ))
        return (item for chunk in result for item in chunk)
```

C.4 Performance of Optimized DPHH Vector Extraction Tool

C.5 Performance Counters for Optimized Defects Analysis Tool

NPES	Ex. T (s)	Mem (GB)	Cyc (T)	Instr. (T)	C Ref (G)	C M (G)
4*	730.95	9.299	6.20	16.15	3.182	1.077
4	415.63	0.103	6.43	16.56	6.368	0.809
5	358.21	0.089	6.32	16.18	5.686	0.919
8	377.89	0.101	6.42	16.17	6.682	4.052
16	361.94	0.102	6.39	16.18	7.044.2	4.322
32	335.55	0.141	6.29	16.19	7.113	4.075

Brs (T)	Br. M (G)	T-C (s)	CPUs	Faults (K)	Minor Faults (K)	Context Switches	Migr.
3.345	8.147	1868.94	2.547	9.31	9.31	273235	15
3.429	8.281	1943.29	4.669	50.16	50.16	8147929	386
3.346	8.618	1908.27	5.295	45.43	45.43	2255023	727
3.343	8.956	1941.78	5.109	57.25	57.25	2130787	1399
3.347	8.540	1934.18	5.336	71.50	71.50	2064884	5784
3.3501	8.683	1909.21	5.680	123.91	123.91	1809951	37222

Table C.1: Performance metrics for different NPES using `map`* and `imap` execution modes.

*`Pool.map()`
Abbreviations: NPES = Number of Processes, Ex. T = Execution Time, Mem = Memory, Cyc = Cycles (Trillion), Instr. = Instructions (Trillion), C Ref = Cache References (Billion), Brs = Branches (Trillion), Br. M = Branch Misses (Billion), T-C = Task-Clock, CPUs = CPUs Utilized, Faults = Faults, Min. Faults = Minor Faults, CS = Context Switches, Migr. = Migrations.

Performance Counter	32 PE	16 PE	8 PE	4 PE	1 PE
Cycles	$6.37 \times 10^{13} \pm 7.31 \times 10^{9}$	$6.28 \times 10^{13} \pm 6.80 \times 10^{9}$	$6.30 \times 10^{13} \pm 1.09 \times 10^{10}$	$6.26 \times 10^{13} \pm 6.25 \times 10^{9}$	$6.14 \times 10^{13} \pm 7.14 \times 10^{9}$
Instructions	$1.44 \times 10^{14} \pm 1.06 \times 10^{9}$	$1.44 \times 10^{14} \pm 1.08 \times 10^{9}$	$1.44 \times 10^{14} \pm 1.08 \times 10^{9}$	$1.44 \times 10^{14} \pm 1.08 \times 10^{9}$	$1.44 \times 10^{14} \pm 1.09 \times 10^{9}$
Cache References	$7.50 \times 10^{10} \pm 2.63 \times 10^{7}$	$7.31 \times 10^{10} \pm 2.62 \times 10^{7}$	$7.27 \times 10^{10} \pm 2.64 \times 10^{7}$	$7.31 \times 10^{10} \pm 1.73 \times 10^{7}$	$7.32 \times 10^{10} \pm 2.62 \times 10^{7}$
Cache Misses	$2.94 \times 10^{10} \pm 1.57 \times 10^{9}$	$2.81 \times 10^{10} \pm 1.17 \times 10^{9}$	$2.85 \times 10^{10} \pm 2.66 \times 10^{9}$	$2.83 \times 10^{10} \pm 7.58 \times 10^{8}$	$3.91 \times 10^{9} \pm 4.83 \times 10^{8}$
Branches	$3.43 \times 10^{13} \pm 1.11 \times 10^{9}$	$3.43 \times 10^{13} \pm 1.11 \times 10^{9}$	$3.43 \times 10^{13} \pm 1.11 \times 10^{9}$	$3.43 \times 10^{13} \pm 1.12 \times 10^{9}$	$3.42 \times 10^{13} \pm 1.09 \times 10^{9}$
Branch Misses	$9.71 \times 10^{9} \pm 1.56 \times 10^{8}$	$9.52 \times 10^{9} \pm 1.77 \times 10^{8}$	$9.57 \times 10^{9} \pm 1.43 \times 10^{8}$	$9.49 \times 10^{9} \pm 1.58 \times 10^{8}$	$9.28 \times 10^{9} \pm 1.57 \times 10^{8}$
Task Clock (sec)	$2.08 \times 10^{7} \pm 209,166.32$	$1.92 \times 10^{7} \pm 155,339.68$	$2.42 \times 10^{7} \pm 18,348.35$	$4.79 \times 10^{7} \pm 2.13 \times 10^{6}$	$1.85 \times 10^{4} \pm 167.78$
Faults (per sec)	$13.10 \times 10^{6} \pm 169.90$	$12.18 \times 10^{6} \pm 137.67$	$12.23 \times 10^{6} \pm 137.33$	$12.25 \times 10^{6} \pm 1.46 \times 10^{6}$	$4.47 \times 10^{6} \pm 62.51$
Minor Faults (per sec)	$13.10 \times 10^{6} \pm 169.90$	$12.18 \times 10^{6} \pm 137.67$	$12.23 \times 10^{6} \pm 137.33$	$12.25 \times 10^{6} \pm 1.46 \times 10^{6}$	$4.47 \times 10^{6} \pm 62.51$
Context Switches	$562,154 \pm 13,040.76$	$506,324 \pm 13,435.87$	$589,000 \pm 11,678.25$	$585,933 \pm 18,243.83$	$228,972 \pm 6,529.45$
Migrations	$1,654.33 \pm 72.44$	$1,000.33 \pm 71.73$	81.33 ± 10.56	73.67 ± 9.19	75.00 ± 7.07
CPUs Utilized	29.67 ± 0.23	14.94 ± 0.66	7.79 ± 0.01	3.94 ± 0.04	1.00 ± 0.00

Table C.2: Performance counter statistics across different numbers of processing elements (PE) with standard deviations represented using $\pm$.

Appendix D

Appendix for Chapter 5

D.1 Optimized DP-HH and Defects Analysis Tool

```
import struct
import numpy as np
import os
from multiprocessing import Pool
from numba import njit
from joblib import Parallel, delayed
import tempfile
import math

# Directory to save the output files
output_dir = '../output'
# Create the output directory if it doesn't exist
os.makedirs(output_dir, exist_ok=True)

################################################
#### GENERATOR FOR BINARY TRAJECTORY READING ####
################################################

# Coordinates generator using FB method
def read_binary(file_path):
    """
    Generator to read binary file containing
    atomic coordinates.

    This function reads a binary file, extracts
    the number of atoms, header, atom types and their coordinates,
    and yields the coordinates for each timestep.

    Parameters:
    file_path (str): Path to the binary file.

    Yields:
    np.ndarray: Array of coordinates for each timestep.
    """
    with open(file_path, 'rb') as file:
        while 1:
            # Read the number of atoms in the current timestep
            num_atoms_data = file.read(4)
            if not num_atoms_data:
                break  # End of file
            num_atoms = struct.unpack('I', num_atoms_data)[0]

            # Read the length of the header
            header_len_data = file.read(4)
            if not header_len_data:
                break  # End of file
            header_len = struct.unpack('I', header_len_data)[0]
```

```
            file.read(header_len)  # Skip the header

            # Read the coordinates data
            data = file.read(num_atoms * 16)
            if len(data) < num_atoms * 16:
                break  # Incomplete data

            # Extract and reshape the coordinates
            # (buffer = data = 1D array of byte-type object,
            # reshape atomic coordinates into 2D ignoring the atomtypes)
            # If the buffer has data that is not in machine byte-order,
            # this should be specified
            # as part of the data-type for correct interpretation
            coords = np.frombuffer(data, dtype=np.float32).reshape(-1, 4)[:, 1:]
            yield coords

################################################################
########## DIPOLE AND HYDROGEN-HYDROGEN VECTOR EXTRACTION #######
################################################################
def pbc(x, box_size):
    """
    Apply periodic boundary conditions (PBC)
    to coordinates.

    This function ensures that coordinates remain
    within the simulation box by wrapping them around
    if they exceed the boundaries.

    Parameters:
    x (np.ndarray): Array of coordinates.
    box_size (float): Size of the simulation box.

    Returns:
    np.ndarray: Corrected coordinates after applying PBC.
    """
    return x - box_size * np.round(x / box_size)

def process_timestep(coords_np):
    """
    Process coordinates for a single
    timestep to compute DP and HH vectors.

    This function calculates the dipole (DP)
    and hydrogen-hydrogen (HH) vectors
    for each water molecule in the simulation box.

    Parameters:
    coords_np (np.ndarray): Array of atomic coordinates.

    Returns:
    tuple: Two np.ndarrays containing DP and HH
    vectors respectively.
    """
    num_molecul = 1019  # Number of water molecules
    box_size = 31.197   # Size of the simulation box

    dp_list = []  # List to store DP vectors
    hh_list = []  # List to store HH vectors

    try:
        # Iterate over each molecule in the timestep
        for j in range(coords_np.shape[0] // (3 * num_molecul)):
            for i in range(num_molecul):
                # Calculate the OH vectors
                oh1 = coords_np[3 * num_molecul * j + i]
                - coords_np[3 * num_molecul * j + num_molecul + 2 * i]
                oh2 = coords_np[3 * num_molecul * j + i]
                - coords_np[3 * num_molecul * j + num_molecul + 2 * i + 1]

                # Apply periodic boundary conditions
                oh1 = pbc(oh1, box_size)
                oh2 = pbc(oh2, box_size)

                # Calculate the DP and HH vectors
```

```
                dp = oh1 + oh2
                hh = oh1 - oh2

                # Append the vectors to the respective lists
                dp_list.append(dp)
                hh_list.append(hh)

        # Convert lists to numpy arrays
        dp_np = np.array(dp_list)
        hh_np = np.array(hh_list)
    except Exception as e:
        print(f"Error processing timestep: {e}")
        return np.array([]), np.array([])

    return dp_np, hh_np


def dphh_extraction():
    """
    Main function to read
    binary data and process it.

    This function reads atomic
    coordinates from a binary file,
    computes DP and HH vectors
    for each timestep, and writes
    the results to output files.
    """
    # Path to the binary input file
    binary_file_path = '../output/water_lmp_3057_10-001.bin'

    # Paths to the output files
    dp_path = os.path.join(output_dir, 'DP_space_LAMMPS.dat')
    hh_path = os.path.join(output_dir, 'HH_space_LAMMPS.dat')

    # Open output files for writing
    with open(dp_path, 'w') as f_dp, open(hh_path, 'w') as f_hh:
        with Pool(5) as pool:
            try:
                # Process each timestep in parallel
                for result in pool.imap(process_timestep,
                                        read_binary(binary_file_path)):
                    dp_np, hh_np = result
                    if dp_np.size and hh_np.size:
                        # Write DP vectors to the DP file
                        for dp in dp_np:
                            f_dp.write(f"{dp[0]:.3f}\n{dp[1]:.3f}\n{dp[2]:.3f}\n")

                        # Write HH vectors to the HH file
                        for hh in hh_np:
                            f_hh.write(f"{hh[0]:.3f}\n{hh[1]:.3f}\n{hh[2]:.3f}\n")
            except Exception as e:
                print(f"Error in multiprocessing pool: {e}")

    print(f'Done writing DP_space_LAMMPS.dat to {dp_path}')
    print(f'Done writing HH_space_LAMMPS.dat to {hh_path}')


######################################################
#######  H-BOND DEFECTS AND MOLECULAR INDICES  #######
######################################################
# Remember to Change this
# when working with a different box size
# inverse L = 1.0/box_size
L_INV = 0.03205436
# Optimized: vectorized
@njit
def dis(vec1, vec2, L):
    """
    Calculate the minimum distance
    between two vectors in a periodic
    boundary condition (PBC) system.

    The function uses vectorized
```

```
    operations and applies minimum image convention
    to account for periodic boundaries.
    It calculates the shortest distance between
    two points, considering the system's box length L.

    Parameters:
    vec1 (np.array): The first vector (position).
    vec2 (np.array): The second vector (position).
    L (float): The length of the periodic box.

    Returns:
    float: The shortest distance between vec1 and vec2
    within the PBC.
    """
    vesc = vec1 - vec2
    vesc -= L * np.round(vesc * L_INV)
    dist_sq = np.dot(vesc, vesc)
    return np.sqrt(dist_sq)

# Optimized: vectorized
@njit
def angle(vec1, vec2, L):
    """
    Calculate the angle between two
    vectors in a periodic boundary
    condition (PBC) system.

    This function normalizes
    the vectors and computes
    the angle between them using
    the dot product. It applies minimum
    image convention for PBC and handles
    floating-point errors when calculating
    the angle in degrees.

    Parameters:
    vec1 (np.array): The first vector.
    vec2 (np.array): The second vector.
    L (float): The length of the periodic box.

    Returns:
    float: The angle between vec1 and vec2
    in degrees. Returns 0 if either vector
    is a zero vector.
    """
    vec1 -= L * np.round(vec1 * L_INV)
    vec2 -= L * np.round(vec2 * L_INV)

    norm_vec1 = np.sqrt(np.dot(vec1, vec1))
    norm_vec2 = np.sqrt(np.dot(vec2, vec2))

    if norm_vec1 == 0 or norm_vec2 == 0:
        return 0  # Return 0 for zero vectors

    xa = np.dot(vec1, vec2) / (norm_vec1 * norm_vec2)
    # xa = np.clip(xa, -1.0, 1.0)  # To avoid floating-point errors
    if xa >= 1.0 or xa <= -1.0:
        xa = int(xa)
    return math.acos(xa) * 57.2957795129  # (180.0 / 3.1415926536)

@njit
# Refactored to avoid duplication in calculate_defects and calculate_indices functions
def compute_quickmatrix(coords_np, kg, frame_idx, L, ang_ct, dist_ct):
    """
    Generate a matrix representing the
    connectivity between particles based on
    distance and angle criteria.

    The function creates a matrix where entries are
    set to 1 if two particles are within a given
    distance (dist_ct) and the angle between them
    is less than a given angle (ang_ct), otherwise 0.

    Parameters:
```

```
    coords_np (np.array): The coordinate array
    containing particle positions.
    kg (int): The number of particles per group.
    frame_idx (int): Index of the current frame or group.
    L (float): The length of the periodic box.
    ang_ct (float): The angle threshold for connectivity (in degrees).
    dist_ct (float): The distance threshold for connectivity.

    Returns:
    np.array: A symmetric matrix where 1 indicates that
    two particles are within distance and angle criteria.
    """
    quickmatrix = np.zeros((kg, kg))

    for ig in range(kg - 1):
        for jg in range(ig + 1, kg):
            if dis(coords_np[ig + frame_idx * kg * 3],
                   coords_np[jg + frame_idx * kg * 3], L) <= dist_ct:
                quickmatrix[ig, jg] = 1.0
                quickmatrix[jg, ig] = 1.0

    for ig in range(kg):
        for jg in range(kg):
            if quickmatrix[ig, jg] == 0:  # Skip angle calculation if distance is greater than dist_ct
                continue
            ang1 = angle(coords_np[ig + frame_idx * kg * 3] -
                         coords_np[jg + frame_idx * kg * 3],
                         coords_np[ig + frame_idx * kg * 3] -
                         coords_np[(2 * ig) + frame_idx * kg * 3 + kg], L)
            ang2 = angle(coords_np[ig + frame_idx * kg * 3] -
                         coords_np[jg + frame_idx * kg * 3],
                         coords_np[ig + frame_idx * kg * 3] -
                         coords_np[(2 * ig) + 1 + frame_idx * kg * 3 + kg], L)
            ang1 = min(ang1, ang2)
            angw = 0 if ang1 >= ang_ct else 1.0

            quickmatrix[ig, jg] *= angw

    return quickmatrix

# Optimized version using the refactored compute_quickmatrix function
def calculate_defects_and_indices(coords_np, kg, frame_idx, L, ang_ct, dist_ct):
    """
    Calculate the defect scores and index
    matrices for each particle in a system.

    The function determines defects
    based on connectivity of particles using distance
    and angle thresholds, storing defect scores
    and indices for each particle.

    Parameters:
    coords_np (np.array): The coordinate array containing
    particle positions.
    kg (int): The number of particles per group.
    frame_idx (int): Index of the current frame or group.
    L (float): The length of the periodic box.
    ang_ct (float): The angle threshold for connectivity (in degrees).
    dist_ct (float): The distance threshold for connectivity.

    Returns:
    np.array: Array of defect scores.
    np.array: Matrix of indices representing
              connected molecules.
    """
    defects = np.zeros(2 * kg)
    indices = np.zeros((2 * kg, 5))
    quickmatrix = compute_quickmatrix(coords_np, kg, frame_idx, L, ang_ct, dist_ct)

    for i in range(kg):
        defects[i * 2] = np.sum(quickmatrix[i, :])
        defects[1 + i * 2] = np.sum(quickmatrix[:, i])

        non_zero_rows = np.nonzero(quickmatrix[i, :])[0]
```

```
        non_zero_cols = np.nonzero(quickmatrix[:, i])[0]

        indices[i * 2, :len(non_zero_rows)] = non_zero_rows
        indices[1 + i * 2, :len(non_zero_cols)] = non_zero_cols

    return defects, indices

def process_single_frame(coords_np, kg, frame_idx, L, ang_ct, dist_ct):
    """Calculate defects and indices for a single frame."""
    return calculate_defects_and_indices(coords_np, kg, frame_idx, L, ang_ct, dist_ct)


def process_frame_chunk(chunk_frames, kg, L, ang_ct, dist_ct):
    """
    Process a chunk of frames and store the
    results in temporary files.

    Parameters:
    - chunk_frames: A list of frames to process.
    - kg: Number of water molecules.
    - L: Size of the simulation box.
    - ang_ct: Angle cutoff for defect calculation.
    - dist_ct: Distance cutoff for defect calculation.

    Returns:
    - Paths to the temporary files for defects and indices.
    """
    temp_def_file = tempfile.NamedTemporaryFile(delete=False, mode='w')
    temp_ndx_file = tempfile.NamedTemporaryFile(delete=False, mode='w')

    with open(temp_def_file.name, 'w') as f_def, open(temp_ndx_file.name, 'w') as f_ndx:
        for frame_idx, coords_np in chunk_frames:
            # Process the frame
            numframes = int(len(coords_np) / (3 * kg))
            results = [process_single_frame(coords_np, kg, j, L, ang_ct, dist_ct)
                       for j in range(numframes)]

            # Write results to temporary files
            for defects, indices in results:
                np.savetxt(f_def, np.array(defects).reshape(1, -1), fmt='%d')
                np.savetxt(f_ndx, np.transpose(indices), fmt='%d', delimiter=' ')

    return temp_def_file.name, temp_ndx_file.name

def process_defects_and_indices_parallel(kg=1019, L=31.197, ang_ct=30, dist_ct=3.5,
                                         start_frame=0, end_frame=None, n_jobs=32):
    """
    Parallel processing of frames using joblib to calculate defects and indices.

    Parameters:
    - kg: Number of water molecules.
    - L: Size of the simulation box.
    - ang_ct: Angle cutoff for defect calculation.
    - dist_ct: Distance cutoff for defect calculation.
    - start_frame: Starting frame for processing.
    - end_frame: Ending frame for processing.
    - n_jobs: Number of parallel jobs to run.
    """
    defects_file_path = f'{output_dir}/defects_space_combined.txt'
    indices_file_path = f'{output_dir}/indices_space_combined.txt'
    binary_file_path = '../output/water_lmp_3057_10-001.bin'

    coords_generator = read_binary(binary_file_path)
    frames = [(i, coords) for i, coords in enumerate(coords_generator)
              if i >= start_frame and (end_frame is None or i < end_frame)]

    # Calculate chunk sizes that account for an uneven number of frames
    chunk_size = len(frames) // n_jobs
    remainder = len(frames) % n_jobs

    # Distribute frames with remainder accounted for
    frame_chunks = []
    start_idx = 0
    for i in range(n_jobs):
```

```
        # add an extra frame to the first ‘remainder‘ chunks
        end_idx = start_idx + chunk_size + (1 if i < remainder else 0)
        frame_chunks.append(frames[start_idx:end_idx])
        start_idx = end_idx

    # Run parallel processing with joblib
    temp_files = Parallel(n_jobs=n_jobs)(delayed(process_frame_chunk)(
        chunk_frames=chunk, kg=kg, L=L, ang_ct=ang_ct, dist_ct=dist_ct
    ) for chunk in frame_chunks)

    # Merge temporary files into final output files
    with open(defects_file_path, 'w') as f_def, open(indices_file_path, 'w') as f_ndx:
        for temp_def, temp_ndx in temp_files:
            with open(temp_def, 'r') as td, open(temp_ndx, 'r') as ti:
                f_def.write(td.read())
                f_ndx.write(ti.read())
            # Remove temporary files after merging
            os.remove(temp_def)
            os.remove(temp_ndx)

    print(f'Done writing defects to {defects_file_path}')
    print(f'Done writing indices to {indices_file_path}')


#######################################
##### MAIN TASK EXECUTION BLOCK ######
#######################################

if __name__ == "__main__":

    # Process coordinates to extract DP HH vectors
    # into their respective output files using
    # multiprocessing (8 CPUs; Kindly change to suit available CPUs)
    dphh_extraction()

    # Process coordinates to analyze the defects in
    # Hydrogen bonding and the indices of affected molecules in parallel
    # Change the number of processes according to available CPUs
    process_defects_and_indices_parallel(kg=1019, L=31.197, ang_ct=30, dist_ct=3.5,
                                         start_frame=0, end_frame=None, n_jobs=32)
    # Estimated total execution = 18 minutes

#######################################################
###### RESULTS FROM TEST RUN ON 125000 FRAMES ######
#######################################################

'''
(base) [egasu000@lrdn0186 src]$ perf stat -e cycles,instructions,
cache-references,cache-misses,branches,branch-misses,task-clock,faults
,minor-faults,cs,migrations python dphhdefndx.py

Done writing DP_space_LAMMPS.dat to ../output/DP_space_LAMMPS.dat
Done writing HH_space_LAMMPS.dat to ../output/HH_space_LAMMPS.dat
Done writing defects to ../output/defects_space_combined.txt
Done writing indices to ../output/indices_space_combined.txt

 Performance counter stats for 'python dphhdefndx.py':

    70071429975056      cycles                    #    3.017 GHz
   160134037249452      instructions              #    2.29  insn per cycle
       80996482224      cache-references          #    3.487 M/sec
       32074708873      cache-misses              #   39.600 % of all cache refs
    37656523831128      branches                  #    1.621 G/sec
       19111717543      branch-misses             #    0.05% of all branches
       23228022.05 msec task-clock               #   22.005 CPUs utilized
          13117161      faults                    #  564.713 /sec
          13117161      minor-faults              #  564.713 /sec
           2633066      cs                        #  113.357 /sec
              3821      migrations                #    0.164 /sec

    1055.566068567 seconds time elapsed

   23116.817235000 seconds user
      67.954982000 seconds sys
```

```
(base) [egasu000@login07 src]$ ls -ltr -h ../output
total 8.3G
-rw-r--r--. 1 egasu000 interactive 2.4G Nov 16 23:25 HH_space_LAMMPS.dat
-rw-r--r--. 1 egasu000 interactive 2.4G Nov 16 23:25 DP_space_LAMMPS.dat
-rw-r--r--. 1 egasu000 interactive 3.2G Nov 16 23:37 indices_space_combined.txt
-rw-r--r--. 1 egasu000 interactive 486M Nov 16 23:37 defects_space_combined.txt
'''
```

Bibliography

[1] A. Offei-Danso, U. N. Morzan, A. Rodriguez, A. Hassanali, and A. Jelic, "The collective burst mechanism of angular jumps in liquid water," *Nature Communications*, vol. 14, no. 1, p. 1345, 2023.

[2] R. Goni, R. Apostolov, M. Lundborg, C. Bernau, F. Jamitzky, E. Laure, E. Lindhal, P. Andrio, Y. Becerra, M. Orozco, and J. L. Gelpi, "Standards for data handling," tech. rep., ScalaLife, 2013. https://www.bsc.es/sites/default/files/public/life_science/molecular_modeling/d7.3_-_white_paper_on_standards_for_data_handling.pdf.

[3] E. Bressert, *SciPy and NumPy: An Overview for Developers*. Sebastopol, CA: O'Reilly Media, 2012. pp. 9-14.

[4] P. Sarker, T. Lu, D. Liu, G. Wu, H. Chen, M. S. J. Sajib, S. Jiang, Z. Chen, and T. Wei, "Hydration behaviors of nonfouling zwitterionic materials," *Chemical Science*, vol. 14, no. 27, pp. 7500–7511, 2023.

[5] C. Gabriel, "Dielectric properties of biological materials," *Bioengineering and Biophysical Aspects of Electromagnetic Fields*, pp. 87–136, 2018.

[6] P. E. Lopes, B. Roux, and A. D. MacKerell, "Molecular modeling and dynamics studies with explicit inclusion of electronic polarizability: theory and applications," *Theoretical Chemistry Accounts*, vol. 124, pp. 11–28, 2009.

[7] S. Rai and D. Rai, "Probing the electric field response of a water molecule confined in small carbon nanocages: A density functional theory investigation," *ChemPhysChem*, p. e202400718, 2024.

[8] S.-Y. Sheu, Y.-C. Liu, J.-K. Zhou, E. W. Schlag, and D.-Y. Yang, "Surface topography effects of globular biomolecules on hydration water," *The Journal of Physical Chemistry B*, vol. 123, no. 32, pp. 6917–6932, 2019.

[9] D. Zong, H. Hu, Y. Duan, and Y. Sun, "Viscosity of water under electric field: Anisotropy induced by redistribution of hydrogen bonds," *The Journal of Physical Chemistry B*, vol. 120, no. 21, pp. 4818–4827, 2016.

[10] M. D. Fayer, "Water in a crowd," *Physiology*, vol. 26, no. 6, pp. 381–392, 2011.

[11] M. Dinpajooh and D. V. Matyushov, "Interface dielectric constant of water at the surface of a spherical solute," *Journal of Molecular Liquids*, vol. 376, p. 121400, 2023.

[12] R. Hou, C. Li, and D. Pan, "Raman and ir spectra of water under graphene nanoconfinement at ambient and extreme pressure–temperature conditions: a first-principles study," *Faraday Discussions*, vol. 249, pp. 181–194, 2024.

[13] A. Luzar and D. Chandler, "Hydrogen-bond kinetics in liquid water," *Nature*, vol. 379, no. 6560, pp. 55–57, 1996.

[14] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011.

[15] J. P. Mesirov, "Accessible reproducible research," *Science*, vol. 327, no. 5964, pp. 415–416, 2010.

[16] P. Virtanen, R. Gommers, E. Burovski, T. E. Oliphant, W. Weckesser, D. Cournapeau, P. Peterson, T. Reddy, M. Haberland, J. Wilson, *et al.*, "scipy/scipy: Scipy 1.6. 0," *Zenodo*, 2021.

[17] J. Ranjani, A. Sheela, and K. P. Meena, "Combination of numpy, scipy and matplotlib/pylab-a good alternative methodology to matlab-a comparative analysis," in *2019 1st international conference on innovations in information and communication technology (ICIICT)*, pp. 1–5, IEEE, 2019.

[18] M. Rocklin *et al.*, "Dask: Parallel computation with blocked algorithms and task scheduling.," in *SciPy*, pp. 126–132, 2015.

[19] CINECA, *Leonardo Supercomputer Service Description*, 2024. https://leonardo-supercomputer.cineca.eu/leonardo-service-description/ Accessed: 2024-10-07.

[20] B. R. Brooks, C. L. Brooks III, A. D. Mackerell Jr, L. Nilsson, R. J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, *et al.*, "Charmm: the biomolecular simulation program," *Journal of computational chemistry*, vol. 30, no. 10, pp. 1545–1614, 2009.

[21] W. Humphrey, A. Dalke, and K. Schulten, "Vmd: visual molecular dynamics," *Journal of molecular graphics*, vol. 14, no. 1, pp. 33–38, 1996.

[22] B. Hess, C. Kutzner, D. Van Der Spoel, and E. Lindahl, "Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of chemical theory and computation*, vol. 4, no. 3, pp. 435–447, 2008.

[23] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein, "Mdanalysis: a toolkit for the analysis of molecular dynamics simulations," *Journal of computational chemistry*, vol. 32, no. 10, pp. 2319–2327, 2011.

[24] L. Dalcín, R. Paz, and M. Storti, "Mpi for python," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 1108–1115, 2005.

[25] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf Accessed: 2024-06-16.

[26] J. Enkovaara, C. Rostgaard, J. J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. Hansen, *et al.*, "Electronic structure calculations with gpaw: a real-space implementation of the projector augmented-wave method," *Journal of physics: Condensed matter*, vol. 22, no. 25, p. 253202, 2010.

[27] J. Enkovaara, N. A. Romero, S. Shende, and J. J. Mortensen, "Gpaw-massively parallel electronic structure calculations with python-based software," *Procedia Computer Science*, vol. 4, pp. 17–25, 2011.

[28] GPAW Project, *GPAW Manual*, 2017. https://wiki.fysik.dtu.dk/gpaw/documentation/manual.html Accessed 2024-11-05.

[29] M. Wagner, G. Llort, E. Mercadal, J. Giménez, and J. Labarta, "Performance analysis of parallel python applications," *Procedia Computer Science*, vol. 108, pp. 2171–2179, 2017.

[30] *MareNostrum III User's Guide*, 2016. https://www.bsc.es/support/MareNostrum3-ug.pdf Accessed: 2024-11-01.

[31] Python Software Foundation, *The Python Profilers*, 2017. https://docs.python.org/2/library/profile.html Accessed 2024-10-18.

[32] J. Noller and R. Oudkerk, *Addition of the multiprocessing package to the standard library*, 2008.

[33] Python Software Foundation, *Binary Data Services — Python 3.12.0 Documentation*, 2024. https://docs.python.org/3/library/struct.html#struct.Struct.unpack_from Accessed: 2024-07-19.

[34] NumPy Developers, *numpy.frombuffer*, 2023. https://numpy.org/doc/stable/reference/generated/numpy.frombuffer.html Accessed: 2024-07-13.

[35] "Joblib: running python functions as pipeline jobs," ., 2020.

[36] Joblib Developers, *Thread-based parallelism vs process-based parallelism — Joblib 1.4.0 documentation*, 2023. https://joblib.readthedocs.io/en/stable/parallel.html#thread-based-parallelism-vs-process-based-parallelism.