Cloud Computing and Data Science
https://ojs.wiserpub.com/index.php/CCDS/

UNIVERSAL WISER PUBLISHER

Research Article

Performance and Reproducibility Assessment of Quantum Dissipative Dynamics Framework: A Comparative Study of Fortran Compilers, MKL, and FFTW

Benjamin Antunes, Mazel Claude, David R. C. Hill*

Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines St-Etienne, LIMOS UMR CNRS 6158, 1 rue de la Chebarde, Aubière, 63178, France
E-mail: david.hill@uca.fr

Received: 20 December 2024; **Revised:** 28 May 2025; **Accepted:** 3 June 2025

Abstract: This paper seeks to assess the performance, energy consumption, and repeatability of the framework Quantum Dissipative Dynamics (QDD). We have observed some trouble with repeatability and reproducibility in such programs. QDD uses the Math Kernel Library (MKL) and the Fastest Fourier Transform in the West (FFTW) library to compute the Discrete Fourier Transform (DFT), in addition to the new Intel Fortran compiler compared to well-established ones, such as gfortran and the former ifort. Our findings indicate that gfortran, despite being open source, exhibits commendable performance when compared to the Intel compilers for this application. In our case, the new ifx compiler does not appear to offer significant benefits in performance over its predecessors. Additionally, our results suggest that MKL outperforms FFTW in terms of computational speed. Regarding energy consumption, there is minimal difference among the options, supporting the notion that faster execution is more energy-efficient. This paper provides performance comparisons and recommendations aimed at enhancing the repeatability and reproducibility of scientific computing experiments. In addition, we found that, by default, FFTW sometimes lacks determinism, which compromises the repeatability essential for debugging. We found a compromise between speed and reliable results for our configuration with QDD. The combination of gfortran and MKL is the one performing the best, contrary to what was expected.

***Keywords*:** high performance computing, reproducibility, MKL, FFTW, Fortran

1. Introduction

As modern physics increasingly relies on computer science and High-Performance Computing (HPC), researchers engage in intensive scientific computing. In the realm of scientific computing, particularly within the field of physics, Fortran remains a language of choice. Intel has recently made its proprietary compiler, ifort, available for free and introduced a new compiler, ifx, designed to succeed ifort. In this study, we focus on a physics application known as Quantum Dissipative Dynamics (QDD) [1]. Our goals are twofold: firstly, to illustrate a real-world example of the challenges with non-repeatability and non-reproducibility that many researchers in high-performance computing encounter; and secondly, to assess the performance in terms of time and energy consumption of three compilers: gfortran, ifort, and ifx. We also compare these variables using the widely recognized implementations of the Discrete

DOI: https://doi.org/10.37256/ccds.6220256280

Fourier Transform (DFT) with the Fastest Fourier Transform in the West (FFTW) library and the Intel Math Kernel Library. The current versions of both libraries are particularly successful, and they still show interesting performances [2]. Even if the Intel MKL is known to be efficient, scientists are competing to always provide better performance. The Basic Linear Algebra Subprograms (BLAS) are widely used routines that provide efficient building blocks for vector and matrix operations. A recent thesis has presented a complex BLAS library optimized for both memory-bound (Level-1 BLAS) and compute-bound (Level-3 BLAS) operations, with a detailed optimization guide [3]. This has led to better performances against Intel MKL (gains of up to 2.75%) and OpenBLAS (gains of up to 3.48%). This highly technical research is also important in Machine Learning, where we find extensions of MKL. For instance, the OneAPI Deep Neural Network (OneDNN) library, formerly known as Intel MKL-DNN, is an open-source performance library crafted explicitly for deep learning tasks [4]. When Graphics Processing Units (GPUs) are also used, specialized implementations for the Compute Unified Device Architecture (CUDA) can be more efficient [5], particularly for matrix multiplications. The purpose of this paper is to provide valuable technical insights for physics simulationists, facilitating informed decisions regarding the implementation of their simulations with these two main libraries, and mainly for fast implementations of the Discrete Fourier Transform.

QDD is a computational framework designed for simulating the behavior of electrons and ions in finite systems such as atoms, molecules, and clusters under external electromagnetic fields. A key aspect of QDD is its focus on dissipative dynamics, which are the energy loss processes occurring due to interactions between electrons—specifically, electron-electron collisions. These interactions lead to dynamic correlations, which can now be modeled from the very beginning of the excitation process using quantum mechanics, specifically through the Relaxation-Time Approximation (RTA). The physical concept underpinning QDD is based on the principles of Time-Dependent Density Functional Theory (TDDFT), particularly using the Time-Dependent Local-Density Approximation (TDLDA). This approach is augmented by a Self-Interaction Correction (SIC), which is mandatory for accurately predicting electron emission—a phenomenon where electrons are ejected from the system due to the energy imparted by external electromagnetic fields such as lasers. The computational method integrates quantum mechanical treatments of electron dynamics with classical molecular dynamics for ions. This involves using a 3D grid for mapping electronic states and employing Fast Fourier Transforms (FFT) to switch between real space and momentum space. The inclusion of absorbing boundary conditions helps model the electron emission accurately.

Fast Fourier Transform is an algorithm that computes the Discrete Fourier Transform (DFT) of a sequence. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. This is why, in practice, we use known implementations of FFT. FFTW [6] is one of the most used open-source libraries for computing DFT [7, 8]. On the other hand, Intel offers a proprietary solution with the Math Kernel Library (MKL), which is free to use and can also perform fast Fourier transforms, with its Application Programming Interface (API) adapted for codes that use FFTW (making it easy to switch from FFTW to MKL).

To speed up the computation, we can also parallelize it. QDD uses OpenMP. OpenMP, short for Open Multi-Processing, is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran on many types of processor architectures [9]. It is designed to enable programmers to develop parallel applications more easily by providing an interface for defining parallel regions, loops, and sections of code. OpenMP is particularly favored in the scientific computing community for its ease of use and ability to incrementally parallelize existing codebases.

The utilization of Fortran in conjunction with OpenMP and a Fast Fourier Transform algorithm is commonplace in high-performance computing for physics. Moreover, there is increasing pressure to enhance computation speed while reducing energy consumption. In this paper, we seek to determine whether FFTW or MKL is superior in terms of numerical reproducibility, time efficiency, and energy usage. Similar inquiries apply to the compilers ifort, ifx, and gfortran. We also explore how to achieve numerical reproducibility.

To address these questions, we will initially discuss related previous studies. Subsequently, we will outline the significance of reproducibility and the challenges encountered in ensuring it within the realm of high-performance computing for physics simulations. We will describe our materials and methods, and present our results, which include statistical data on time and energy consumption, as well as assessments of numerical reproducibility, which revealed the non-deterministic behavior of the FFTW library.

2. Related work

Several studies have addressed the performance, energy consumption, and reproducibility of computational tools in High-Performance Computing (HPC). Memeti et al. [10] highlighted that programming with OpenMP requires less effort compared to OpenCL and CUDA, with similar performance and energy efficiency observed across these platforms. They also established an easy-to-understand correlation between computation time and energy consumption: programs that take longer to compute typically use more energy.

Pereira et al. [11] examined computing time and energy consumption across several benchmarks, concluding that languages that perform faster also tend to be more energy-efficient. Their findings show strong similarities in rankings for both time and energy consumption. The study also positions Fortran as moderate in terms of energy and time efficiency, ranking 10th and 14th, respectively, out of 27. Despite this, Fortran remains competitive in HPC due to the optimization and availability of scientific libraries like OpenMP and FFTW, more so than many other languages, with the exception of C and C++.

Investigations into DFT algorithms reveal that the open-source Fast Fourier Transform implementation, FFTW, frequently used in engineering and scientific applications for its high performance, compares well against vendor-supplied libraries like MKL, which has recently become freely available [6]. In [8], the study suggests that while FFTW and MKL show comparable results in one-dimensional serial executions, MKL often outperforms in multi-dimensional and parallel scenarios. In [8], the authors state regarding FFTW: "One of the most frequently used algorithms in engineering and scientific applications is Fast Fourier Transform (FFT). Its open-source implementation (Fastest Fourier Transform in the West (FFTW)) is widely used, mainly due to its excellent performance, comparable to vendor-supplied libraries." In [12-14], the authors have explored the computation time performance of FFT libraries on different C/C++ compilers and supercomputers, varying numbers of cores, and interconnect technologies. They also explore the reproducibility of time performance for FFT libraries on different machines. However, they do not study the ability to reproduce numerical values, as we do in our experiment. In addition to measuring the performance of FFT libraries, they use a different FFT library for each supercomputer. They do not compare MKL and FFTW on the same machine, and their objective was not to compare the performances of MKL and FFTW.

Considering the different Fortran compilers, the Polyhedron benchmarks [15] indicate that the Intel compiler generally performs better than gfortran, although these results should be considered with caution due to potential bias as they originate from a vendor. In [16], the authors compare ifort and gfortran, finding that ifort is more performant overall, both on a single CPU core and on multiple cores.

Concerning reproducibility when trying to achieve high performance, [17] describes the difficulties in achieving reproducibility with Fortran programs, pointing out issues like the non-determinism introduced by certain compiler optimizations and the variability in computational results due to different system architectures and compiler behaviors. Further research by [18] presents compiler options to enhance repeatability across GNU and Intel compilers, which we have applied in our work to test reproducibility among different compilers.

In summary, while Fortran is not always the most efficient in terms of energy or time, its integration with well-optimized libraries and the ability to control compiler settings make it a viable option that is still widely used in HPC. Discussions at conferences and in the literature, such as the Supercomputing Conference in 2013 [19], underscore the importance of reproducibility in Fortran applications, advocating specific compilation strategies to enhance repeatability.

3. The importance of reproducibility for HPC applications

Complex software can act as black boxes [20], making it hard to reproduce results without the same computer and software setup. As computers are vital research tools, they require the same rigorous quality checks as instruments in other sciences. Popper emphasized that reproducibility is crucial for scientific progress, requiring clear documentation so others can replicate findings and increase trust in the results. A re-edition of his thoughts about scientific discovery can be found here [21]. Moreover, having diverse researchers redo experiments helps control for biases and variations in setups. Although exact reproduction is not always possible or necessary—especially in high-performance computing, where large computations can take weeks or months—the reproduction of such work can be very costly.

In high-performance computing research, we have sometimes seen it written that Monte Carlo simulations are nondeterministic due to their reliance on randomness. However, this claim is misleading, as these simulations employ Pseudo-Random Number Generators (PRNGs), which are deterministic methods that mimic randomness but ensure that any random sequence can be exactly replayed. While PRNGs allow for precise control of randomness, their effectiveness hinges on correct usage and initialization. For instance, initializing PRNGs with the system time is ill-advised for scientific applications due to its impact on reproducibility. Proper management of PRNG states and careful selection of parallelization techniques are essential for reliable Monte Carlo simulations, especially when running parallel computations [22]. Furthermore, researchers must distinguish between the fake seed given (often an integer) and the real seed (which is the full state of the PRNG), as the transformation from an Application Programming Interface (API) seed function to the real PRNG state can vary significantly across different platforms, potentially leading to varied outcomes with the same "fake" seed.

In addition, when performing physics simulations, you will probably use parallel computing to speed up the computation. Parallel computing simultaneously executes multiple tasks for more efficient processing but faces challenges such as out-of-order floating-point arithmetic, which can cause non-reproducible results. Variations in task execution order due to factors like scheduling and multithreading introduce nondeterminism, which can alter outcomes unpredictably and complicate debugging and validation. The non-associativity of floating-point operations like addition and multiplication further complicates this, as changing the order of operations can lead to different outcomes, making exact reproducibility difficult in large-scale systems such as exascale supercomputers.

Finally, to be able to detect silent errors, you need to have repeatable code. Otherwise, you will not be able to detect when any silent error occurs and that your results are corrupted. Silent errors significantly impact the reliability of HPC. Disturbances such as electrical or magnetic interference can cause bit flips in Dynamic Random Access Memory (DRAM), leading to unintended state changes. Error-Correcting Code (ECC) memory is able to correct one bit at a time and prevents many issues. As chip components become smaller and denser, the likelihood of such errors increases, impacting even advanced supercomputers like Frontier, with its vast number of cores, where the mean time before failure drops to just a few hours. For clarity purposes, here is a quick reminder of the current ACM definitions of reproducibility, replicability, and repeatability: Reproducibility (different team, same experimental setup); Replicability (different team, different experimental setup); Repeatability (same team, same experimental setup). A detailed survey published in Computer Science Review dealing with reproducible research in HPC explores all these points in detail [23].

4. Materials and methods

Our research was conducted on an Intel server running the Linux distribution Debian 6.1.76-1 with kernel version 6.1.0-18-amd64. The server is equipped with an Intel Xeon Platinum 8160 CPU clocked at 2.10 GHz, featuring 192 cores across 4 sockets—each socket hosting 24 cores with 2 threads per core. The cache includes 3 MiB L1d and L1i (96 instances each), 96 MiB L2 (96 instances), and 132 MiB L3 cache (4 instances).

The study utilized several compilers and libraries to test the performance, energy consumption, and reproducibility of computational methods. The compilers used included GNU Fortran (version 12.2.0), Intel Fortran Compiler Classic (IFORT, version 2021.11.1), and the newer Low Level Virtual Machine (LLVM)-based Intel Fortran Compiler (IFX, version 2024.0.2). LLVM, originally an acronym for Low Level Virtual Machine, has evolved beyond its initial scope and now serves as a compiler infrastructure supporting multiple programming languages and platforms. The mathematical computations leveraged libraries such as FFTW version 3.3.10 and Intel's Math Kernel Library (MKL) version 2024.1.0, included in the oneAPI Base Toolkit. OpenMP version 4.5 was employed for threading support.

The QDD code was compiled using various combinations of compilers, libraries, and settings to investigate different performance metrics. The makefile associated with QDD allowed us to switch between compilers (gfortran, ifort, ifx), FFT libraries (FFTW, MKL), threading options (OpenMP enabled/disabled, dynamic OpenMP), and debugging options (enabled/disabled). The "DYN" option (DYNOMP in the makefile) determines whether OpenMP parallelization is applied to single-particle wave functions (DYNOMP = YES) or to FFT operations using the FFTW3 library (DYNOMP = NO).

Provided compilation options from QDD have been specified to optimize performance for different environments.

For the GNU Fortran compiler (gfortran), the standard compilation settings include optimization flags such as -Ofast for maximum optimization, -mfpmath=sse and -msse4.2 to specify the use of SSE4.2 instructions, and -fdefault-real-8 and -fdefault-double-8 to enforce double precision. Parallel processing is enabled with the -fopenmp flag. These settings are supplemented with either the FFTW or MKL libraries. For debugging purposes, the flags -pg for profiling, -g for generating debug information, and -fbacktrace for stack trace logging are used alongside -w to suppress warnings.

Similarly, for Intel compilers, the configuration includes -fpp for preprocessing, -w to suppress warnings, and -xsse4.2 and -Ofast for optimized Single Instruction/Multiple Data (SIMD) instructions and maximum speed. Additional flags such as -ip for interprocedural optimization, -no-prec-div to disable precise division, and -align all for data alignment are specified. The Intel-specific -qopenmp is used for OpenMP directives, and options for FFTW or MKL are also available. Debug configurations incorporate -pg for profiling, -g for debug information, -CB for array bounds checking, -traceback for enhanced trace information, and -align all and -autodouble for alignment and double precision enforcement, respectively. We compiled QDD with all available combinations of these configuration options. This resulted in thirty-six distinct experimental conditions, each pertaining to a unique combination of compiler settings, debug modes, OpenMP usage, and Fourier transform libraries. For each configuration, a corresponding directory was established, named systematically to reflect the specific compilation settings (e.g., gfortran-FFTW-noOMP-noDebug or ifort-MKL-OMP-noDebug).

Each directory contained the compiled executable and was further subdivided into fifty replication subdirectories. Thanks to these fifty replications, we were able to assess the repeatability of results across multiple runs (within the exact same experimental setup). Similarly, thanks to the different compilation options folders, we could assess the reproducibility across different configurations (with differences in the experimental setup, e.g., the compilation options). The primary metrics evaluated were time performance, energy consumption, and the consistency of output data.

To streamline the compilation and execution process, Bash scripts were developed. These scripts dynamically adjusted the Makefile parameters according to predefined arrays of compiler types, debug settings, OpenMP configurations, and FFT libraries. Specifically, the sed command was utilized to modify the Makefile to reflect the desired compilation options, ensuring that each executable was appropriately configured and placed in its designated directory.

Furthermore, another Bash script facilitated the execution of these experiments. It utilized environment variables to configure system settings, navigated through the directory structure, and executed the compiled QDD software within each replication subdirectory. Execution times for each replication were recorded.

An additional script was dedicated to measuring energy consumption. This involved iterating over directories corresponding to each replication, where the executable was run alongside the PowerJoular tool to monitor energy usage over specified intervals. The script managed temporary directories to isolate each test run, ensuring that energy measurements did not modify numerical results from the time experiment. Once all the results were collected in each folder, we used a Jupyter Notebook to compile and analyze all the results.

To measure the energy consumption of our programs, we used the tool PowerJoular [24]. However, we needed to modify PowerJoular to accurately account for energy consumption in multithreaded applications. Originally, PowerJoular monitored resource usage based solely on the primary Process Identifier (PID) by extracting data from /proc/PID/stat. This method did not capture the resource usage of multiple threads within a process, which are common in modern applications—particularly in our case, where OpenMP parallelization is employed for quantum physics simulations.

To address this limitation, we enhanced PowerJoular to monitor all threads associated with a given process. We achieved this by adding functionality to read from /proc/PID/task, a directory containing subdirectories for each Thread ID (TID) related to the PID. PowerJoular now iterates through these TID subdirectories, collecting data from each stat file and aggregating this information to more accurately capture total resource consumption.

Our modifications have been merged into the main project and can be found in the GitHub merge request: https://github.com/joular/powerjoular/pull/36.

The code related to this article is available at: https://gitlab.isima.fr/beantunes/qdd-reproducibility.

5. Results

All the time and energy data hereafter are considered with the original compilation option for performance given with QDD that we presented above (-Ofast, -sse4.2, …). We will discuss in the numerical reproducibility section some modifications we made about these options and how they influenced numerical reproducibility and performance results. As previously noted, each experiment was replicated fifty times. We used statistical tests, such as Levene's test and the Student's *T*-test, to establish the equivalence of variances and means, respectively, based on a *p*-value of 0.005 (99.5% confidence level).

Levene's test is employed to assess the equality of variances for a variable calculated for two or more groups. The null hypothesis for Levene's test is that the variances are equal across the groups. If the *p*-value is less than 0.005, we reject the null hypothesis, indicating that the variances are not equal. When variances are found to be unequal, we used Welch's *T*-test instead of the standard Student's *T*-test.

The Student's *T*-test, which assumes equal variances, is used to determine if there is a significant difference between the means of two groups. The null hypothesis for the Student's *T*-test is that the means of the two groups are equal. A *p*-value less than 0.005 leads to the rejection of this null hypothesis, suggesting a significant difference between the group means.

With this approach, we measure the influence of the compiler (gfortran, ifort, and ifx) and the FFT library (FFTW and MKL) on time and energy consumption. To achieve this, we isolate the parameter whose influence we wish to measure, ensuring all other variables remain constant.

5.1 *Time performance*

Table 1. Influence of gfortran, ifort and ifx on real time execution

Experiment	Mean real time (s)–gfortran	Std real time (s)–gfortran	Mean real time (s)–ifort	Std real time (s)–ifort	Mean real time (s)–ifx	Std real time (s)–ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	702.00[a]	7.53	1,094.88[a]	4.84	1,461.76[a]	5.24
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	365.28[a]	8.26	413.82[a]	6.12	401.54[a]	6.11
(gfortran/ifort/ifx)-FFTW-OMP-Debug	764.80[a]	12.37	1,258.16[a]	8.66	1,603.26[a]	9.86
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	404.22[a]	12.56	452.64[a]	9.67	443.18[a]	6.73
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	2,685.84[a]	58.56	3,762.24[a]	22.23	4,808.76[a]	30.53
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	1,416.10[a]	14.49	1,337.78[b]	11.95	1,341.68[b]	13.61
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637.82[a]	6.94	1,039.08[a]	4.62	1,365.10[a]	7.19
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296.10[a]	7.05	313.92[a]	7.00	329.92[a]	6.10
(gfortran/ifort/ifx)-MKL-OMP-Debug	747.92[a]	11.61	1,236.00[a]	8.43	1,559.24[a]	8.95
(gfortran/ifort/ifx)-MKL-OMP-noDebug	388.84[b]	8.49	393.34[b]	9.41	415.96[a]	8.74
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2,238.18[a]	5.96	3,349.02[a]	5.56	4,386.84[a]	10.24
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	986.60[a]	5.21	912.02[a]	6.88	927.08[a]	6.37
Overall mean	969.48	19.24	1,296.91	9.89	1,587.03[a]	11.96

a: values where the differences are statistically significant, b: when they are not

In Table 1, we compare the mean and variance of each experiment, varying only between gfortran, ifort, and ifx. In a, we have the mean values that are statistically different from each other (with a 99.5% confidence level). For example, the experiment with FFTW-OMP-Dyn-Debug options, using gfortran, took an average of 702 seconds to run, while it took 1,094 seconds and 1,461 seconds using ifort and ifx, respectively. As these values are shown in a, this implies that these differences are statistically significant, so we can conclude that gfortran leads to better performance in this case.

As observed, gfortran demonstrates considerable advantages in terms of computational time for experiments. Given that the experiments were conducted on an Intel CPU, it was initially anticipated that Intel compilers might offer a competitive edge. However, this does not appear to be the case.

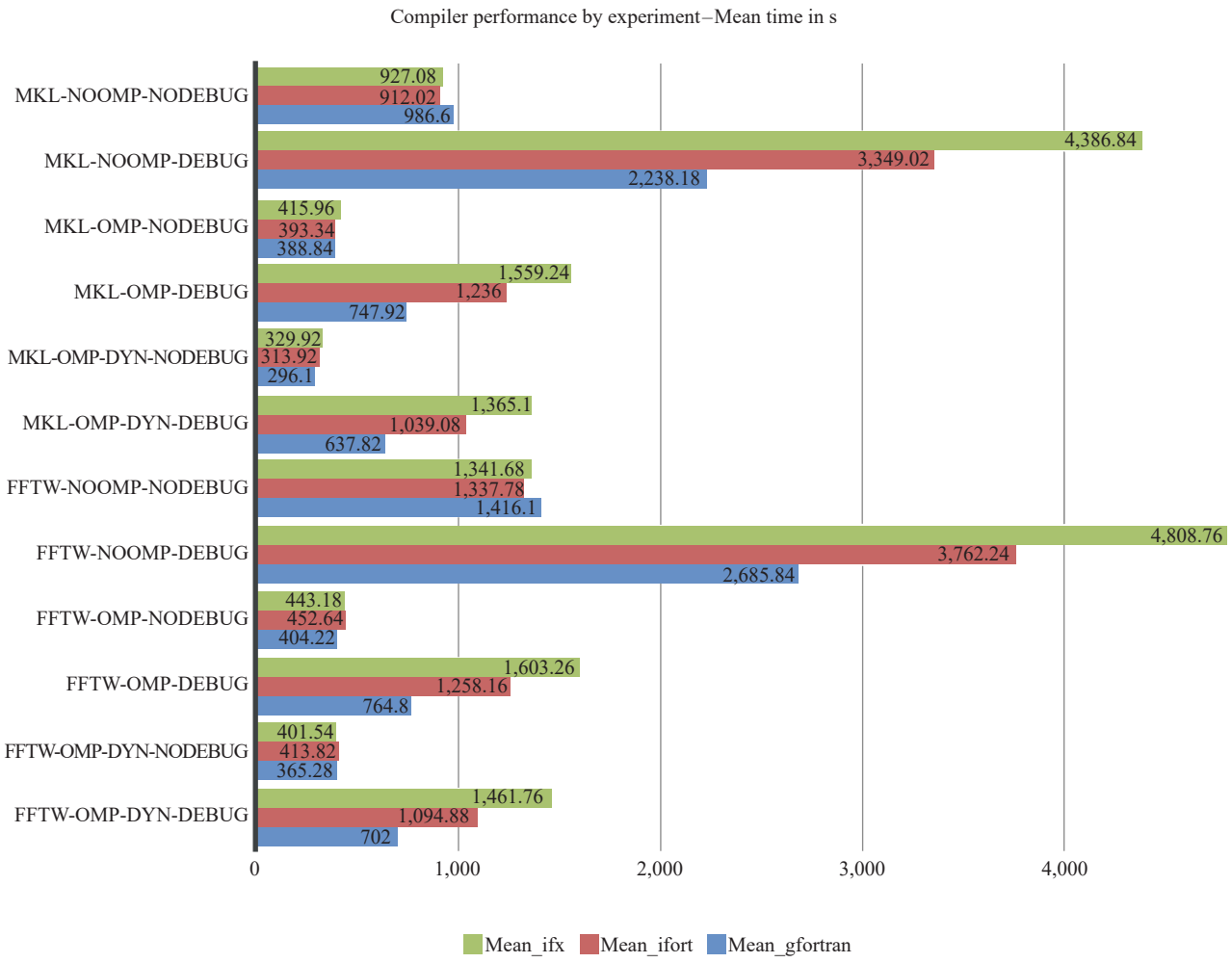

Figure 1. Influence of gfortran, ifort and ifx on real time execution, graph visualization

An additional noteworthy finding is that the new Intel Fortran compiler, ifx, exhibits slower performance compared to ifort (Figure 1). While ifort is 34% faster in real time compared to gfortran (considering the overall mean), ifx lags by 64%, indicating a significant efficiency advantage for gfortran for this application. In Table 2, we explore the same experiment, but we focus our attention on FFTW vs. MKL performances.

In this table, and in Figure 2, MKL appears to be consistently faster for all the experiments, in comparison with FFTW. MKL is approximately 13% more efficient overall.

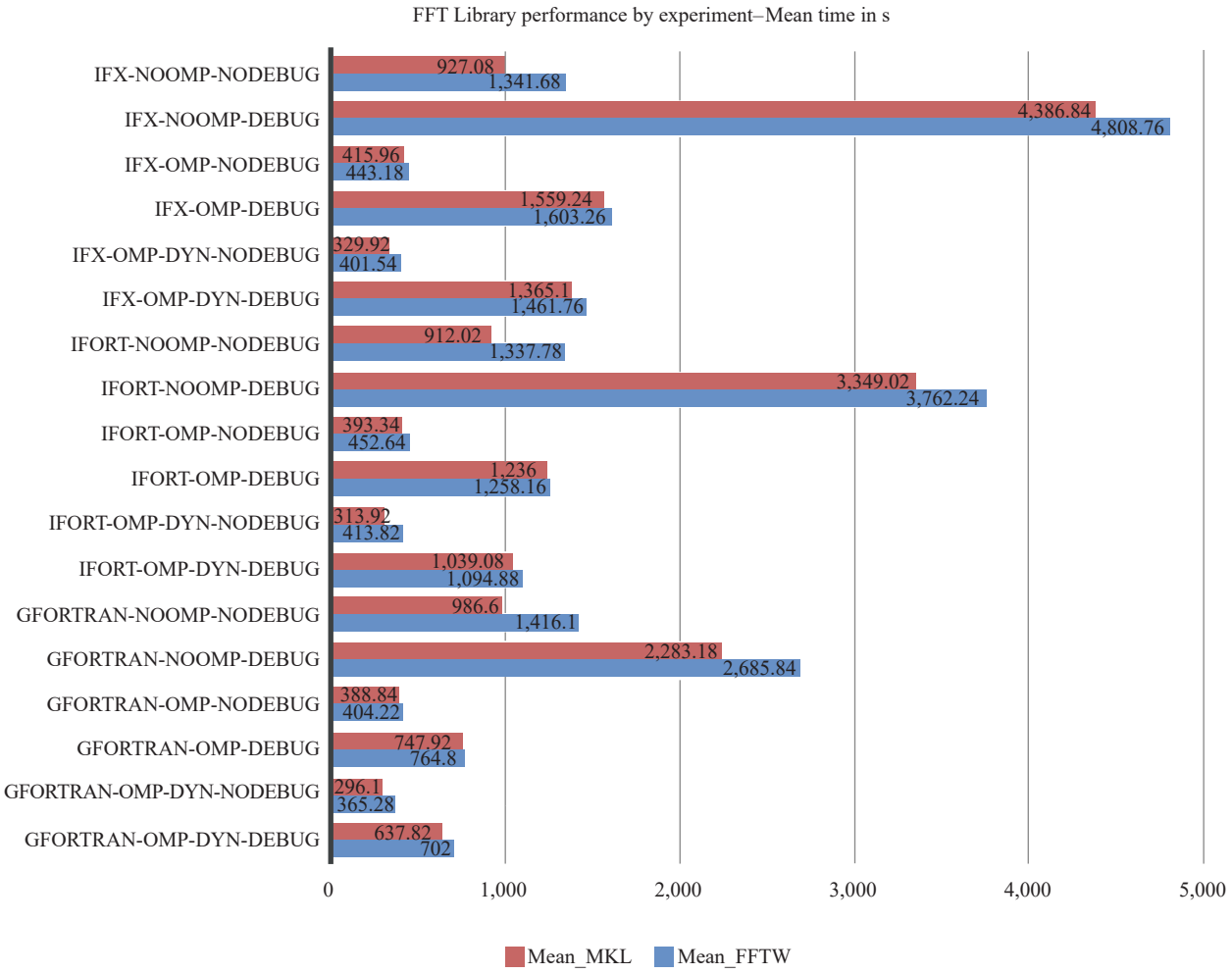

Figure 2. Influence of FFTW and MKL on real time execution, graph visualization

Table 2. Influence of FFTW and MKL on real time execution

Experiment	Mean real time (s)–FFTW	Std real time (s)–FFTW	Mean real time (s)–MKL	Std real time (s)–MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	702.00[a]	7.53	637.82[a]	6.94
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	365.28[a]	8.26	296.10[a]	7.05
gfortran-(FFTW/MKL)-OMP-Debug	764.80[a]	12.37	747.92[a]	11.61
gfortran-(FFTW/MKL)-OMP-noDebug	404.22[a]	12.56	388.84[a]	8.49
gfortran-(FFTW/MKL)-noOMP-Debug	2,685.84[a]	58.56	2,238.18[a]	5.96
gfortran-(FFTW/MKL)-noOMP-noDebug	1,416.10[a]	14.49	986.60[a]	5.21
ifort-(FFTW/MKL)-OMP-DYN-Debug	1,094.88[a]	4.84	1,039.08[a]	4.62
ifort-(FFTW/MKL)-OMP-DYN-noDebug	413.82[a]	6.12	313.92[a]	7.00
ifort-(FFTW/MKL)-OMP-Debug	1,258.16[a]	8.66	1,236.00[a]	8.43

Table 2. (cont.)

Experiment	Mean real time (s)–FFTW	Std real time (s)–FFTW	Mean real time (s)–MKL	Std real time (s)–MKL
ifort-(FFTW/MKL)-OMP-noDebug	452.64[a]	9.67	393.34[a]	9.41
ifort-(FFTW/MKL)-noOMP-Debug	3,762.24[a]	22.23	3,349.02[a]	5.56
ifort-(FFTW/MKL)-noOMP-noDebug	1,337.78[a]	11.95	912.02[a]	6.88
ifx-(FFTW/MKL)-OMP-DYN-Debug	1,461.76[a]	5.24	1,365.10[a]	7.19
ifx-(FFTW/MKL)-OMP-DYN-noDebug	401.54[a]	6.11	329.92[a]	6.10
ifx-(FFTW/MKL)-OMP-Debug	1,603.26[a]	9.86	1,559.24[a]	8.95
ifx-(FFTW/MKL)-OMP-noDebug	443.18[a]	6.73	415.96[a]	8.74
ifx-(FFTW/MKL)-noOMP-Debug	4,808.76[a]	30.53	4,386.84[a]	10.24
ifx-(FFTW/MKL)-noOMP-noDebug	1,341.68[a]	13.61	927.08[a]	6.37
Overall mean	1,373.22	18.66	1,195.72	7.70

5.2 *Energy consumption by minutes*

Table 3. Influence of gfortran, ifort and ifx on energy consumption my minutes

Experiment	Mean consumption (J)–gfortran	Std consumption (J)–gfortran	Mean consumption (J)–ifort	Std consumption (J)–ifort	Mean consumption (J)–ifx	Std consumption (J)–ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	3,489.23[b]	566.39	3,909.33[b]	576.31	3,765.81[b]	602.46
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	3,902.83[a]	560.89	4,462.09[b]	564.98	4,522.11[b]	759.47
(gfortran/ifort/ifx)-FFTW-OMP-Debug	4,281.07[b]	498.90	4,504.28[b]	507.90	4,517.33[b]	566.94
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	4,469.58[b]	509.26	4,613.95[b]	637.51	4,517.08[b]	541.76
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3,389.82[b]	607.27	3,379.13[b]	578.19	3,065.26[b]	629.05
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	3,388.09[b]	599.72	3,220.56[b]	676.94	3,025.12[b]	565.11
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	3,520.18[b]	482.43	4,015.35[a]	545.70	3,581.70[b]	460.08
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	4,015.48[a]	481.77	4,423.95[b]	432.14	4,302.50[b]	376.83
(gfortran/ifort/ifx)-MKL-OMP-Debug	4,092.32[a]	370.89	4,430.80[b]	434.36	4,405.47[b]	351.06
(gfortran/ifort/ifx)-MKL-OMP-noDebug	4,282.08[b]	416.46	4,525.75[b]	567.16	4,365.06[b]	351.71
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3,018.74[b]	475.80	3,444.45[b]	683.74	3,069.63[b]	640.62
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3,032.50[b]	576.00	3,355.50[a]	533.83	2,918.67[b]	458.51
Overall mean	3,740.16	516.89	4,023.76	566.83	3,837.98	539.36

Regarding the consumption of energy per minute, no discernible differences have been observed: faster computations, ostensibly resulting from more efficient utilization of existing physical resources, do not correspond to increased energy consumption per minute. Consequently, the overall energy consumption is primarily determined by the total duration required for a program to complete its task. In essence, this suggests that programs which operate faster are also more environmentally friendly. These findings corroborate the results reported by [6, 7]. The only significant difference observed is associated with the activation of OpenMP versus noOpenMP. Indeed, employing multiple CPUs concurrently elevates the per-minute energy consumption. However, this increase is relatively minor when compared to the time savings afforded by OpenMP. It can be concluded that hardware, when operational, incurs energy expenditure even if it remains idle. The optimal strategy for minimizing energy wastage involves a better usage of available hardware, trying not to leave it idle. We can see in Table 3 that nearly all values are in blue, which means that we do not see any advantages from gfortran, ifort, or ifx concerning the consumption per minute. However, as gfortran seems to be faster, it would lead to less energy consumed overall.

In Table 4, when studying FFTW and MKL, we obtain the same conclusions as above: we cannot see any difference, except for the case with gfortran and noOMP.

Table 4. Influence of FFTW and MKL on energy consumption by minutes

Experiment	Mean consumption (J)–FFTW	Std consumption (J)–FFTW	Mean consumption (J)–MKL	Std consumption (J)–MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	3,489.23[b]	566.39	3,520.18[b]	482.43
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	3,902.83[b]	560.89	4,015.48[b]	481.77
gfortran-(FFTW/MKL)-OMP-Debug	4,281.07[b]	498.90	4,092.32[b]	370.89
gfortran-(FFTW/MKL)-OMP-noDebug	4,469.58[b]	509.26	4,282.08[b]	416.46
gfortran-(FFTW/MKL)-noOMP-Debug	3,389.82[a]	607.27	3,018.74[a]	475.80
gfortran-(FFTW/MKL)-noOMP-noDebug	3,388.09[a]	599.72	3,032.50[a]	576.00
ifort-(FFTW/MKL)-OMP-DYN-Debug	3,909.33[b]	576.31	4,015.35[b]	545.70
ifort-(FFTW/MKL)-OMP-DYN-noDebug	4,462.09[b]	564.98	4,423.95[b]	432.14
ifort-(FFTW/MKL)-OMP-Debug	4,504.28[b]	507.90	4,430.80[b]	434.36
ifort-(FFTW/MKL)-OMP-noDebug	4,613.95[b]	637.51	4,525.75[b]	567.16
ifort-(FFTW/MKL)-noOMP-Debug	3,379.13[b]	578.19	3,444.45[b]	683.74
ifort-(FFTW/MKL)-noOMP-noDebug	3,220.56[b]	676.94	3,355.50[b]	533.83
ifx-(FFTW/MKL)-OMP-DYN-Debug	3,765.81[b]	602.46	3,581.70[b]	460.08
ifx-(FFTW/MKL)-OMP-DYN-noDebug	4,522.11[b]	759.47	4,302.50[b]	376.83
ifx-(FFTW/MKL)-OMP-Debug	4,517.33[b]	566.94	4,405.47[b]	351.06
ifx-(FFTW/MKL)-OMP-noDebug	4,517.08[b]	541.76	4,365.06[b]	351.71
ifx-(FFTW/MKL)-noOMP-Debug	3,065.26[b]	629.05	3,069.63[b]	640.62
ifx-(FFTW/MKL)-noOMP-noDebug	3,025.12[b]	565.11	2,918.67[b]	458.51
Overall mean	3,912.37	589.26	3,822.23	488.91

5.3 *Numerical reproducibility and its impact on performances*

As previously noted, we conducted fifty replications of each experiment to evaluate repeatability and to identify factors that might influence it's performance. We observed a loss of repeatability in some of the results files. For instance, using the "diff" Linux command to compare the results files, we found discrepancies such as:

File1: 0.00000 0.25529373E-08 0.83551334E-09 0.83553563E-09.

File2: 0.00000 0.25529376E-08 0.83551386E-09 0.83547498E-09.

From two different replications of the experiment gfortran-FFTW-OMP-DYN-Debug.

Or:

File1: 0.00000 0.25529374E-08 0.83551345E-09 0.83547134E-09.

File2: 0.00000 0.25529372E-08 0.83551506E-09 0.83546803E-09.

From two different replications of the experiment gfortran-FFTW-OMP-Debug.

The differences were relatively minor, approximately ranging from 10^{-7} to 10^{-4}. From a physicist standpoint, these variations can be significant or inconsequential, depending on the specific applications. Nevertheless, our objective is to achieve repeatable results, as deterministic machines are designed for this purpose. Such repeatability is crucial for debugging and identifying any silent errors that may occur.

After some investigations, we found that the issues with repeatability were associated with the FFTW library. We investigated why FFTW causes loss of repeatability. From the FFTW documentation, we found:

> "*If you use FFTW_MEASURE or FFTW_PATIENT mode, then the algorithm FFTW employs is not deterministic: it depends on runtime performance measurements. This will cause the results to vary slightly from run to run. However, the differences should be slight, on the order of the floating-point precision, and therefore should have no practical impact on most applications. If you use saved plans (wisdom) or FFTW_ESTIMATE mode, however, then the algorithm is deterministic and the results should be identical between runs*." (https://www.fftw.org/faq/section3.html#nondeterministic).

It appears that, by default, the FFTW library is not deterministic due to its adaptive algorithm selection, which varies based on runtime performance measurements that lack temporal consistency. Following the recommendations provided in the documentation, we modified the code of QDD to enable the deterministic mode of FFTW.

5.3.1 *Modifying the code of QDD to ensure FFTW repeatability*

Table 5. Influence of gfortran, ifort and ifx on real time execution, when setting FFTW to repeatable

Experiment	Mean real time (s)–gfortran	Std real time (s)–gfortran	Mean real time (s)–ifort	Std real time (s)–ifort	Mean real time (s)–ifx	Std real time (s)–ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	779.42[a]	6.29	1,173.54[a]	6.66	1,534.82[a]	5.11
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	441.18[a]	7.35	494.36[a]	7.41	477.12[a]	4.97
(gfortran/ifort/ifx)-FFTW-OMP-Debug	825.62[a]	10.44	1,315.78[a]	8.15	1,667.52[a]	9.36
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	458.90[a]	15.73	504.56[b]	18.51	500.46[b]	10.19
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3,211.00[a]	14.62	4,366.38[a]	71.69	5,423.02[a]	94.39
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	2,002.36[a]	13.63	1,927.26[b]	15.66	1,932.06[b]	9.16
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637.66[a]	7.72	1,046.90[a]	39.95	1,366.24[a]	6.79
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296.00[a]	7.63	314.06[a]	7.31	329.96[a]	6.05
(gfortran/ifort/ifx)-MKL-OMP-Debug	749.96[a]	9.81	1,233.46[a]	6.96	1,560.56[a]	9.28
(gfortran/ifort/ifx)-MKL-OMP-noDebug	387.78[a]	7.58	396.08[a]	11.06	415.64[a]	9.03
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2,262.38[a]	57.07	3,375.74[a]	66.67	4,425.20[a]	83.64
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	993.64[a]	20.92	911.76[a]	16.12	927.38[a]	10.52
Overall mean	1,087.16	14.90	1,421.66	23.01	1,713.33	21.54

In this sub-section, we consider the performance when slightly modifying the QDD codebase to obtain repeatable results with FFTW. In Table 5, we can see that performance are a bit worse than in Table 1; however, proportions between gfortran, ifort, and ifx are kept equal, as expected.

In Table 6, we focus on FFTW and MKL performance, such as in Table 2. The resulting data showed a decline in average performance by approximately 18% for FFTW (1,373 seconds in Table 2 and 1,613 seconds in Table 6), whereas MKL maintained consistent performance levels, as expected. However, this modification successfully enabled bitwise repeatability across all FFTW experiment replications.

Table 6. Influence of FFTW and MKL on real time execution, when setting FFTW to repeatable

Experiment	Mean real time (s)–FFTW	Std real time (s)–FFTW	Mean real time (s)–MKL	Std real time (s)–MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	779.42[a]	6.29	637.66[a]	7.72
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	441.18[a]	7.35	296.00[a]	7.63
gfortran-(FFTW/MKL)-OMP-Debug	825.62[a]	10.44	749.96[a]	9.81
gfortran-(FFTW/MKL)-OMP-noDebug	458.90[a]	15.73	387.78[a]	7.58
gfortran-(FFTW/MKL)-noOMP-Debug	3,211.00[a]	14.62	2,262.38[a]	57.07
gfortran-(FFTW/MKL)-noOMP-noDebug	2,002.36[a]	13.63	993.64[a]	20.92
ifort-(FFTW/MKL)-OMP-DYN-Debug	1,173.54[a]	6.66	1,046.90[a]	39.95
ifort-(FFTW/MKL)-OMP-DYN-noDebug	494.36[a]	7.41	314.06[a]	7.31
ifort-(FFTW/MKL)-OMP-Debug	1,315.78[a]	8.15	1,233.46[a]	6.96
ifort-(FFTW/MKL)-OMP-noDebug	504.56[a]	18.51	396.08[a]	11.06
ifort-(FFTW/MKL)-noOMP-Debug	4,366.38[a]	71.69	3,375.74[a]	66.67
ifort-(FFTW/MKL)-noOMP-noDebug	1,927.26[a]	15.66	911.76[a]	16.12
ifx-(FFTW/MKL)-OMP-DYN-Debug	1,534.82[a]	5.11	1,366.24[a]	6.79
ifx-(FFTW/MKL)-OMP-DYN-noDebug	477.12[a]	4.97	329.96[a]	6.05
ifx-(FFTW/MKL)-OMP-Debug	1,667.52[a]	9.36	1,560.56[a]	9.28
ifx-(FFTW/MKL)-OMP-noDebug	500.46[a]	10.19	415.64[a]	9.03
ifx-(FFTW/MKL)-noOMP-Debug	5,423.02[a]	94.39	4,425.20[a]	83.64
ifx-(FFTW/MKL)-noOMP-noDebug	1,932.06[a]	9.16	927.38[a]	10.52
Overall mean	1,613.08	18.29	1,201.69	21.34

Concerning reproducibility between the different experiments, despite these adjustments to settings, we observed that reproducibility remains unattainable across all options—including the Debug setting—as results varied between each experiment without exception (we do not have bitwise identical results between different experiments, but we do have bitwise identical results between replications of the same experiment). This means that, for example, when running the experiment with ifx-MKL-noOMP-noDebug options, we do not have bitwise identical results to when running with ifx-MKL-noOMP-Debug options. Compiling the program with debug compilation options does not produce exactly the same numerical result as when compiling without these options. For this application, the execution when debugging the

program does not give the exact same results as when we run it without debugging compilation options. This would be a problem if an error occurs when the program is running in normal mode, but not in debug mode.

5.3.2 *Modifying compilation options to improve reproducibility*

Subsequently, we endeavored to determine whether reproducibility could be achieved between various configuration options. Although we did not anticipate identical results across different compilers and FFT libraries, given their primary role in computations, we did expect to achieve reproducibility between experiments conducted with differing settings of OpenMP versus NoOpenMP and Debug versus NoDebug.

In pursuit of this goal, we adhered to the recommendations of [13-15] to modify compilation options. Reference [18] provided several restrictive compilation settings for both gfortran and ifort.

For gfortran, we implemented the following settings:

-w -O0 -ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite- math-only -fno-rounding-math -fno-cx-limited-range.

For debug mode:

pg -w -g -fbacktrace -O0 -ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal- math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range.

For ifort and ifx, we used:

-O0 -fp-model strict -fp-speculation=strict -mp1 -no-vec -no-simd.

For debug mode:

-g -CB -traceback -O0 -fp-model strict -fp-speculation=strict -mp1 -no-vec -no-simd.

These compilation options are specifically chosen to suppress compiler optimizations that could lead to variability in floating-point computations. For instance, flags such as -fno-associative-math and -fp-model strict prevent the compiler from reordering operations or applying approximations that might violate IEEE precision, while -ffloat-store helps avoid discrepancies from extended precision in temporary registers. Likewise, disabling vectorization (-no-vec, -no-simd) ensures that loop transformations do not alter the order of arithmetic operations, which is critical for reproducibility in parallel or high-throughput settings.

Table 7. Influence of gfortran, ifort and ifx on real time performance, when disabling all optimizations

Experiment	Mean real time (s)–gfortran	Std real time (s)–gfortran	Mean real time (s)–ifort	Std real time (s)–ifort	Mean real time (s)–ifx	Std real time (s)–ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	1,157.80[a]	6.02	1,210.82[a]	4.41	1,537.56[a]	6.19
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	1,157.34[a]	6.06	842.94[a]	6.39	728.98[a]	5.38
(gfortran/ifort/ifx)-FFTW-OMP-Debug	1,285.22[a]	9.07	1,345.42[a]	7.13	1,682.96[a]	17.30
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	1,283.40[a]	10.45	966.82[a]	8.98	807.32[a]	10.10
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	4,489.76[a]	14.01	4,833.34[a]	16.80	5,517.20[a]	15.78
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	4,487.72[a]	12.47	3,716.08[a]	24.89	3,309.40[a]	135.87
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	1,009.60[a]	7.19	1,059.08[a]	5.03	1,421.32[a]	6.15
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	1,011.24[a]	7.96	699.90[a]	5.30	585.90[a]	6.06
(gfortran/ifort/ifx)-MKL-OMP-Debug	1,187.14[a]	8.48	1,264.46[a]	7.16	1,595.02[a]	8.56
(gfortran/ifort/ifx)-MKL-OMP-noDebug	1,189.22[a]	8.48	878.12[a]	9.33	727.50[a]	10.08
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3,504.38[a]	6.04	3,850.58[a]	16.43	4,566.86[a]	16.61
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3,506.68[a]	7.71	2,739.50[a]	7.38	2,356.56[a]	23.48
Overall mean	2,105.79[a]	8.66	1,950.59[a]	9.93	2,069.72[a]	21.80

It is noteworthy, however, that the options -no-simd and -mp1 were unavailable for the ifx compiler. This might lead to a more restrictive context for gfortran or ifort than for ifx.

Upon deactivating all optimization features, we can observe in Table 7 that the compilers gfortran, ifort, and ifx perform comparably (Figure 3). This similarity in performance may suggest that the initially superior performance of gfortran could be attributed to the compilation options provided with QDD being more compatible with gfortran, potentially indicating that the performance of the Intel Fortran compiler could also be enhanced similarly. Alternatively, this could be due to more restrictive compilation options on gfortran that we are trying to apply to obtain reproducibility between Debug and OMP options. However, MKL continues to outperform FFTW, the latter remaining in repeatable mode, thereby sacrificing some performance (Table 8 and Figure 4).

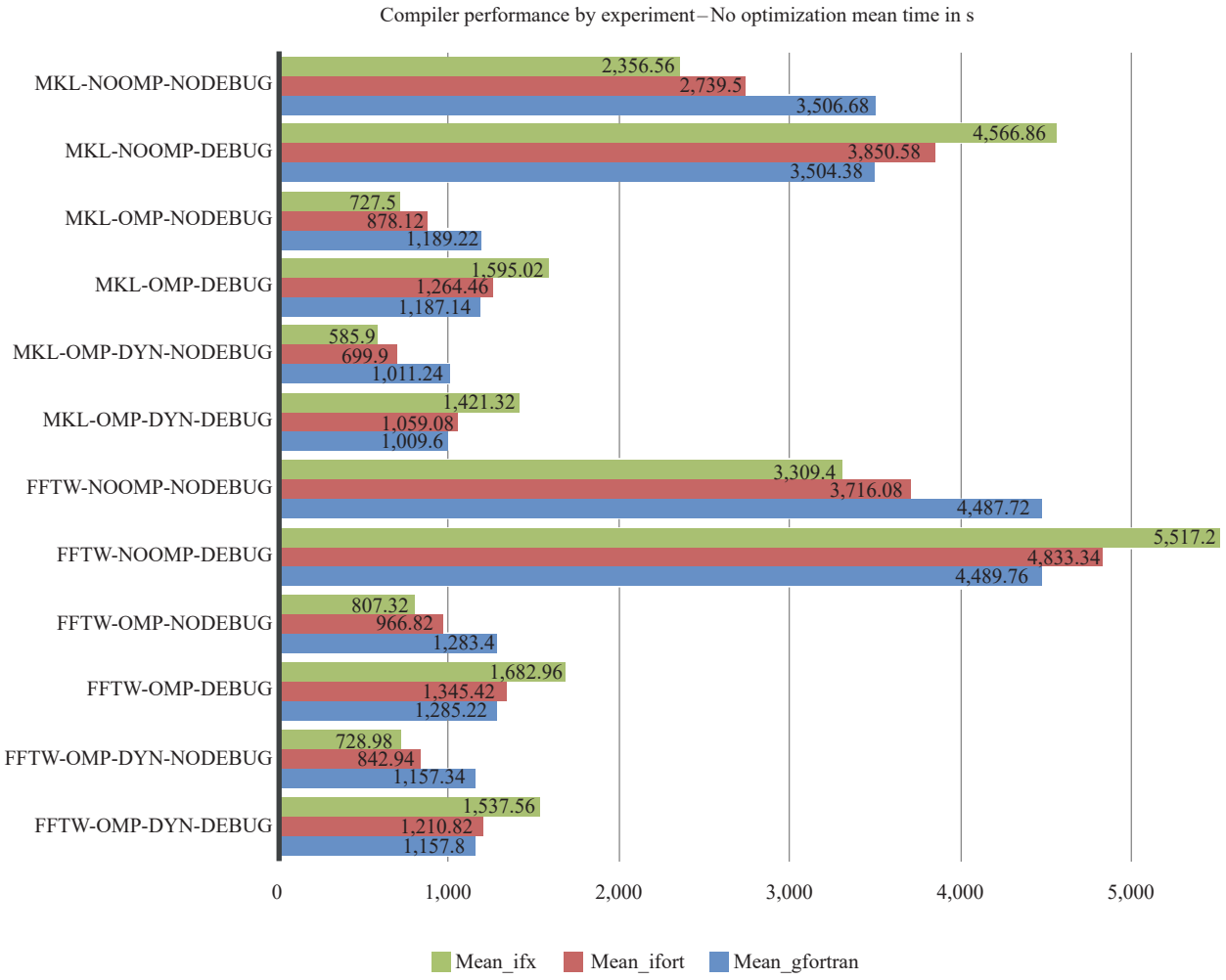

Figure 3. Influence of gfortran, ifort and ifx on real time execution, when disabling all optimizations, graph visualization

We have successfully achieved reproducibility across OpenMP and Debug settings. The presence of numerical repeatability indicates that the lack of reproducibility is not due to intrinsic uncertainties within the code, but is due to differences in the execution, based on the compilation options. Program instructions are not executed in the exact same order, and this can lead to discrepancies, especially in floating-point arithmetic.

Table 8. Influence of FFTW and MKL on real time performance, when disabling all optimizations

Experiment	Mean real time (s)–FFTW	Std real time (s)–FFTW	Mean real time (s)–MKL	Std real time (s)–MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	1,157.80[a]	6.02	1,009.60[a]	7.19
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	1,157.34[a]	6.06	1,011.24[a]	7.96
gfortran-(FFTW/MKL)-OMP-Debug	1,285.22[a]	9.07	1,187.14[a]	8.48
gfortran-(FFTW/MKL)-OMP-noDebug	1,283.40[a]	10.45	1,189.22[a]	8.48
gfortran-(FFTW/MKL)-noOMP-Debug	4,489.76[a]	14.01	3,504.38[a]	6.04
gfortran-(FFTW/MKL)-noOMP-noDebug	4,487.72[a]	12.47	3,506.68[a]	7.71
ifort-(FFTW/MKL)-OMP-DYN-Debug	1,210.82[a]	4.41	1,059.08[a]	5.03
ifort-(FFTW/MKL)-OMP-DYN-noDebug	842.94[a]	6.39	699.90[a]	5.30
ifort-(FFTW/MKL)-OMP-Debug	1,345.42[a]	7.13	1,264.46[a]	7.16
ifort-(FFTW/MKL)-OMP-noDebug	966.82[a]	8.98	878.12[a]	9.33
ifort-(FFTW/MKL)-noOMP-Debug	4,833.34[a]	16.80	3,850.58[a]	16.43
ifort-(FFTW/MKL)-noOMP-noDebug	3,716.08[a]	24.89	2,739.50[a]	7.38
ifx-(FFTW/MKL)-OMP-DYN-Debug	1,537.56[a]	6.19	1,421.32[a]	6.15
ifx-(FFTW/MKL)-OMP-DYN-noDebug	728.98[a]	5.38	585.90[a]	6.06
ifx-(FFTW/MKL)-OMP-Debug	1,682.96[a]	17.30	1,595.02[a]	8.56
ifx-(FFTW/MKL)-OMP-noDebug	807.32[a]	10.10	727.50[a]	10.08
ifx-(FFTW/MKL)-noOMP-Debug	5,517.20[a]	15.78	4,566.86[a]	16.61
ifx-(FFTW/MKL)-noOMP-noDebug	3,309.40[a]	135.87	2,356.56[a]	23.48
Overall mean	2,242.23	17.63	1,841.84	9.30

The current compilation parameters now allow us to achieve bitwise identical results between experiments conducted with and without OpenMP, as well as between those conducted in debug and non-debug modes (this can be useful to debug efficiently, avoiding a situation where the non-debug program behaves differently from the debug one). Nevertheless, regarding compilers and FFT libraries, bitwise identical outcomes are not achieved, although this discrepancy is to be expected, as compilers and FFT libraries are not supposed to execute instructions in the exact same order, since they perform differently.

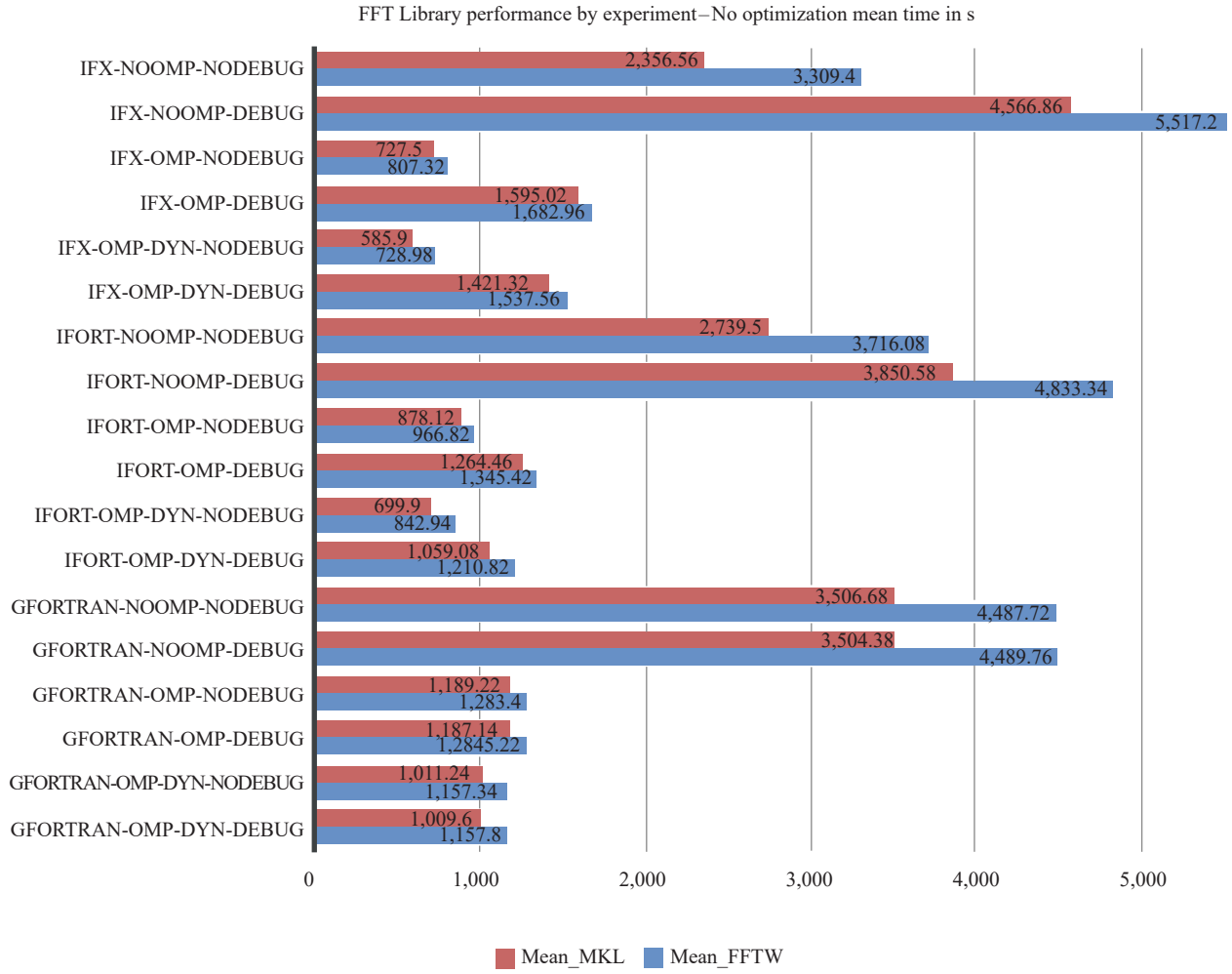

Figure 4. Influence of FFTW and MKL, on real time execution, when disabling all optimizations, graph visualization

6. Discussions

Our observations reveal that MKL not only performs better but also maintains repeatability, contrasting with FFTW, which shows performance degradation when repeatability is enforced. Despite the proprietary nature of MKL, it remains freely available. On the other hand, FFTW is open source.

Our results corroborate the findings from [8], who reported better performance from MKL over FFTW. In terms of compiler efficiency, the gfortran compiler exhibits high performance relative to its commercial counterparts, challenging the prevailing data from the Polyhedron benchmarks [15] and findings by [16].

Consequently, we recommend the integration of gfortran with MKL for optimal performance.

The utilization of OpenMP has demonstrated substantial benefits. Despite potential concerns, parallelization within this library did not compromise repeatability and significantly enhanced performance, particularly notable on a system where we used 16 of 96 physical cores for OpenMP parallelization.

Energy consumption considerations suggest that optimizing resource use and accelerating code execution may be the most effective strategy for minimizing energy usage. However, the potential for increased overall energy consumption when optimizing its usage due to the rebound effect and Jevons' paradox [25, 26] must be considered. These phenomena state that optimization of energy usage will ultimately lead to an increase in energy consumption.

6.1 *Limitations*

The assessment of FFT performance, particularly using FFTW and MKL, might be influenced by dimensional factors. In our analysis, the application was confined to QDD, where the problem is in 3D. This study could benefit from more study cases, with different kinds of physics applications. However, [8] corroborates our findings.

Furthermore, the comparison of gfortran and ifort performance might vary depending on the application type, as suggested by [12], who observed superior performance with ifort across multiple applications, while we observe the opposite conclusion in our case with QDD.

We kept the initial compilation options from the QDD packages. However, as ifx is new, some optimizations might exist that are not available with ifort, and this might lead to better performance from ifx over ifort. In addition, ifx might perform better with more modern Intel CPU features. The CPU we used in our tests is from 2017 (8 years ago).

6.2 *Perspectives*

Our paper is focused on the simulation of the dynamics of a Na_2^+ ion within the QDD framework. Future experiments could expand on these findings to determine the consistency of results across a broader range of ions and electrons. Additionally, more complex physics simulations may require enhanced precision and bitwise identical outcomes between computational runs, where a precision greater than 10^{-6} is necessary. Such investigations could further underline the critical importance of repeatability in scientific computing.

7. Conclusion

This paper has explored the performance and energy consumption of different compilers and libraries within the field of scientific computing, particularly emphasizing the importance of repeatability and reproducibility in high-performance computing. Our findings demonstrate that MKL outperforms FFTW in terms of time performance and maintains bitwise repeatability. In addition, we discovered that the default functioning of FFTW can produce non-deterministic results. This is critical for scientific tasks where consistent results are paramount. Despite the proprietary status of MKL, its availability at no cost provides a viable option for researchers seeking efficient computational tools. Our study also highlights the superior performance of the gfortran compiler over its proprietary counterparts, contradicting common benchmarks and previously published results in the case of our application. This could be due to some specificity of the QDD package, or to better compilation options for gfortran. This suggests that gfortran, when paired with MKL, is an effective combination for fast and reliable scientific computing that is aware of performance and energy consumption. In this study, the use of OpenMP did not impact repeatability. However, we have noted that the pursuit of faster processing speeds might conflict with the objectives of repeatability and reproducibility, presenting a philosophical dilemma in scientific computing practices. We faced repeatability challenges when using the Debug/No-Debug option. We had to disable optimization options in order to obtain bitwise identical results. This was similar to the issues we faced with the default non-deterministic functioning of FFTW. Fixing the FFTW repeatability issue implies a loss of performance. The decrease in performance is often the price to pay for obtaining reliable scientific results.

Acknowledgments

The authors would like to thank Prof. Eric Suraud and Dr. Marc Vincendon form the French National Center for Scientific Research (CNRS) and the Theoretical Physics Laboratory of Toulouse University (UPS). Their interest in reproducibility, their sharing of code, data and knowledge, and their care for Open Science have made this study possible. We also thank the reviewers for their insightful comments and constructive suggestions, which helped improve the quality of this paper. This research was conducted at LIMOS UMR CNRS 6158 and benefited from the use of its computing resources.

Conflict of interest

The authors declare that they have no conflict of interest.

References

[1] Dinh PM, Vincendon M, Coppens F, Suraud E, Reinhard PG. Quantum Dissipative Dynamics (QDD): A real-time real-space approach to far-off-equilibrium dynamics in finite electron systems. *Computer Physics Communications*. 2022; 270: 108155. Available from: https://doi.org/10.1016/j.cpc.2021.108155.

[2] Chakkour T. Parallel computation to bidimensional heat equation using MPI/CUDA and FFTW package. *Frontiers in Computer Science*. 2024; 5: 1305800. Available from: https://doi.org/10.3389/fcomp.2023.1305800.

[3] Zhang B. *Optimizing Complex BLAS and Implementing Fault Tolerance for Complex Level-1 BLAS on Intel CPUs with AVX512*. University of California; 2024.

[4] Nagrath B, Gupta A, Garg S, Mohanty A. Leveraging intel developer cloud: Leaf disease detection using CNN. In: *2024 3rd International Conference on Artificial Intelligence for Internet of Things (AIIoT)*. Vellore, India: IEEE; 2024. p.1-5. Available from: https://doi.org/10.1109/AIIoT58432.2024.10574670.

[5] Torres LA, Barrios HCJ, Denneulin Y. Evaluation of computational and power performance in matrix multiplication Libraries-MKL vs cuBLAS. In: *Latin American High Performance Computing Conference*. Cham: Springer; 2024. p.83-95. Available from: https://doi.org/10.1007/978-3-031-80084-9_6.

[6] Frigo M, Johnson SG. The design and implementation of FFTW3. *Proceedings of the IEEE*. 2005; 93(2): 216-231. Available from: https://doi.org/10.1109/JPROC.2004.840301.

[7] Donoho DL, Stodden V. Reproducible research in the mathematical sciences. In: Higham NJ, Dennis MR, Glendinning P, Martin PA, Santosa F, Tanner J. (eds.) *The Princeton Companion to Applied Mathematics*. Princeton, NJ, USA: Princeton University Press; 2015. p.916-925.

[8] Nikolić M, Jović A, Jakić J, Slavnić V, Balaž A. An analysis of FFTW and FFTE performance. In: *High-Performance Computing Infrastructure for South East Europe's Research Communities: Results of the HP-SEE User Forum 2012*. Cham: Springer International Publishing; 2014. p.163-170. Available from: https://doi.org/10.1007/978-3-319-01520-0_20.

[9] Chandra R. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers; 2001.

[10] Memeti S, Li L, Pllana S, Kołodziej J, Kessler C. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. New York, NY, United States: Association for Computing Machinery; 2017. p.1-6. Available from: https://doi.org/10.1145/3110355.3110356.

[11] Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, et al. Energy efficiency across programming languages: how do energy, time, and memory relate? In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, United States: Association for Computing Machinery; 2017. p.256-267. Available from: https://doi.org/10.1145/3136014.3136031.

[12] Gambron P, Thorne S. *Comparison of several FFT libraries in C/C++*. Swindon, UK. Science and Technology Facilities Council (STFC). Technical Report RAL-TR-2020-003, 2020.

[13] Aseeri S, Batrašev O, Icardi M, Leu B, Liu A, Li N, et al. Solving the Klein-Gordon equation using Fourier spectral methods: A benchmark test for computer performance. *arXiv:1501.04552*. 2015. Available from: https://doi.org/10.48550/arXiv.1501.04552.

[14] Aseeri S, Muite BK, Takahashi D. Reproducibility in benchmarking parallel fast Fourier transform based applications. In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. New York, NY, United States: Association for Computing Machinery; 2019. p.5-8. Available from: https://doi.org/10.1145/3302541.3313105.

[15] Fortran UK. *Polyhedron Benchmarks*. Available from: https://fortran.uk/fortran-compiler-comparisons/polyhedron-benchmarks-linux64-on-intel/ [Accessed 19th April 2024].

[16] Young SLE, Muruganandam P, Adhikari SK, Lončar V, Vudragović D, Balaž A. OpenMP GNU and Intel Fortran programs for solving the time-dependent Gross–Pitaevskii equation. *Computer Physics Communications*. 2017; 220: 503-506. Available from: https://doi.org/10.1016/j.cpc.2017.07.013.

[17] Charpilloz C, Arteaga A, Fuhrer O, Montek T, Harrop C. Reproducible climate and weather simulations: An application to the cosmo model. In: *Platform for Advanced Scientific Computing (PASC) Conference*. Lugano,

Switzerland; 2017.
[18] Li R, Liu L, Yang G, Wang B. Bitwise identical compiling setup: prospective for reproducibility and reliability of Earth system modeling. *Geoscientific Model Development*. 2016; 9(2): 731-748. Available from: https://doi.org/10.5194/gmd-9-731-2016.
[19] Lionel S. *Improving Numerical Reproducibility in C/C++/Fortran*. SuperComputing. 2013. Available from: https://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp129s1.pdf [Accessed 3rd June 2024].
[20] Marwick B. How computers broke science—and what we can do to fix it. *The Conversation*. 2015. Available from: https://theconversation.com/how-computers-broke-science-and-what-we-can-do-to-fix-it-49938 [Accessed 3rd June 2024].
[21] Popper K. *The Logic of Scientific Discovery*. Routledge; 2005.
[22] Hill DRC, Mazel C, Passerat-Palmbach J, Traore MK. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*. 2013; 25(10): 1427-1442. Available from: https://doi.org/10.1002/cpe.2942.
[23] Antunes B, Hill DRC. Reproducibility, replicability and repeatability: A survey of reproducible research with a focus on high performance computing. *Computer Science Review*. 2024; 53: 100655. Available from: https://doi.org/10.1016/j.cosrev.2024.100655.
[24] Noureddine A. Powerjoular and joularjx: Multi-platform software power monitoring tools. In: *2022 18th International Conference on Intelligent Environments (IE)*. Biarritz, France: IEEE; 2022. p.1-4. Available from: https://doi.org/10.1109/IE54923.2022.9826760.
[25] Alcott B. Jevons' paradox. *Ecological Economics*. 2005; 54(1): 9-21. Available from: https://doi.org/10.1016/j.ecolecon.2005.03.020.
[26] Sorrell S. Jevons' paradox revisited: The evidence for backfire from improved energy efficiency. *Energy Policy*. 2009; 37(4): 1456-1469. Available from: https://doi.org/10.1016/j.enpol.2008.12.003.