Performance and reproducibility assessment of quantum dissipative dynamics framework: a comparative study of Fortran compilers, MKL, and FFTW

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

PERFORMANCE AND REPRODUCIBILITY ASSESSMENT OF QUANTUM DISSIPATIVE DYNAMICS FRAMEWORK: A COMPARATIVE STUDY OF FORTRAN COMPILERS, MKL, AND FFTW

Benjamin Antunes
Claude Mazel
David R.C. Hill

Université Clermont Auvergne, CNRS,
Clermont Auvergne INP,
Mines Saint-Etienne, LIMOS,
1 rue de la Chebarde, Aubière, F-63178
FRANCE

ABSTRACT

This paper seeks to assess the performance, energy consumption and repeatability of the framework Quantum Dissipative Dynamics. We have observed some trouble with repeatability and reproducibility in such programs. QDD is using MKL and FFTW for discrete Fourrier transform, in addition with the new Intel Fortran compiler relative 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. 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. In addition, it is noted that FFTW sometimes lacks determinism, which compromises repeatability essential for debugging. This paper provides performance comparisons and recommendations aimed at enhancing the repeatability and reproducibility of computing scientific experiments. In 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

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 QDD (Quantum Dissipative Dynamics) (Dinh et al., 2022). 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 from the widely recognized implementations of the discrete Fourier transform algorithm, FFTW and MKL. The purpose of this paper is to provide valuable technical insights for physics simulationists, facilitating informed decisions regarding the implementation of their simulations.

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 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 in modeling 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 are using known implementations of FFT. FFTW (Frigo & Johnson, 2005) is the one of the most used open source library to make DFT (Donoho & Stodden, 2015; Nikolić et al., 2014). On the other hand, Intel offers a proprietary solution with Math Kernel Library (MKL) that is free to use and can also performs fast Fourier transform, with its application programming interface (API) adapted for codes that are using FFTW (you can then easily switch from FFTW to MKL).

To fasten the computation, we can also parallelize it. QDD is using 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 (Chandra, 2001). 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.

2 RELATED WORK

Several studies have addressed the performance, energy consumption, and reproducibility of computational tools in high-performance computing (HPC). (Memeti et al., 2017) highlight that programming with OpenMP requires less effort compared to OpenCL and CUDA, with similar performance and energy efficiencies 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., 2017) examined the computing time and energy consumption on 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. This 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 than many other languages except for 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 (Frigo & Johnson, 2005). In (Gambron & Thorne, 2020), 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 (Nikolić et al., 2014), authors write talking about 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 of the West, FFTW) is widely used, mainly due to its excellent performance, comparable to the vendor-supplied libraries*."

Considering the different Fortran compilers, the Polyhedron benchmarks (*Polyhedron benchmarks*, n. d.) indicates 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 (Young-S et al., 2017), authors are comparing ifort and gfortran, resulting in ifort being more performant overall, on a single and on multiple CPU cores.

Concerning reproducibility when trying to achieve high performance, (Charpilloz et al., 2017) describe 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 (Li et al., 2016) 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 makes it a viable option still used in HPC. The discussions at conferences and in literature, such as at the supercomputing conference in 2013 (Lionel, 2013), underscore the importance of reproducibility in Fortran applications, advocating for specific compilation strategies to enhance repeatability.

3 THE IMPORTANCE OF REPRODUCIBILITY FOR HPC APPLICATIONS

Complex software can act as black boxes (Marwick, 2015), 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 (Popper, 2005). Moreover, having diverse researchers redo the experiments helps control for biases and variations in setups, although exact reproduction is not always possible or necessary, especially for high performance computing, where large computations can take weeks or months, the reproduction of such work could be very costly.

In high-performance computing research, we have sometimes seen 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 the 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 (Hill et al., 2013). 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 making physics simulation, you will probably be using parallel computing to fasten the computation. Parallel computing simultaneously executes multiple tasks for more efficient processing but faces challenges like out-of-order floating-point arithmetic, causing non-reproducible results. Variations in task execution order due to factors like scheduling and multithreading introduce nondeterminism, which can alter outcomes unpredictably, complicating debugging and validation. The non-associativity of floating-point operations like addition and multiplication further complicates this, as changing operation orders can lead to different outcomes, making exact reproducibility difficult in large-scale systems like exascale supercomputers.

Finally, to be able to face silent errors, you will need to have a repeatable code. Else, you will not be able to detect that any silent error occurs and that your results are corrupted. Silent errors significantly impact the reliability of HPC. Disturbances like electrical or magnetic interference can cause bits flip in dynamic random access memory (DRAM), leading to unintentionally flip states. Error-correcting code (ECC) memory is able to correct one bit at a time and avoids many troubles. 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 purpose, 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 deals with reproducible research in HPC explores all mentioned points in detail (Antunes & Hill, 2024).

4 MATERIALS AND METHODS

Our research was conducted on an Intel server, which operates under the Linux distribution (Debian 6.1.76-1, kernel 6.1.0-18-amd64). The server is equipped with an Intel Xeon Platinum 8160 CPU at 2.10GHz, 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 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. It supports 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 (enabled/disabled OpenMP, 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 SIMD instructions and maximum speed. Additional flags such as -ip for inter-procedural 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 the combinations of configuration options available. 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 the fifty replications, we can assess the repeatability of results across multiple runs (within the exact same experimental setup), and thanks to the different compilation options folders, we can 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 before being 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 of each replication was 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 were not modifying numerical results from time experiment. Once all the results are in each folders, we use a Jupyter Notebook to compile and analyze all the results.

To measure energy consumption of programs, we use the tool PowerJoular (Noureddine, 2022). We had to modify the PowerJoular tool code to take into account the multithreading energy consumption. Originally, Powerjoular was designed to monitor the resource consumption of processes based solely on the primary process ID (PID), extracting data from /proc/PID/stat. This approach, however, did not account for the additional resource usage of multiple threads within a process, which are common in modern applications to enhance performance, and specifically in our case of OpenMP parallelization for quantum physics simulation. To overcome this limitation, we enhanced PowerJoular to include monitoring of all threads associated with a process. This was achieved by adding functionality to read from /proc/PID/task, a directory that contains subdirectories for each thread ID (TID) related to the PID. Powerjoular now iterates through these subdirectories, collecting data from each stat file of TID and aggregating this information to capture the total resource consumption more accurately. Our modification has been merged, and here is the Github merge request: https://github.com/joular/powerjoular/pull/36.

The code of 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 theses 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 were 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

In table 1, we compare the mean and variance of each experiment, varying only gfortran, ifort and ifx. In green, 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 702 seconds in average to run, while it took 1094 seconds and 1461 seconds using ifort and ifx, respectively. As values are written in green, this suppose that these differences are statistically different, so we can conclude that gfortran leads to better performance in this case.

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	7,53	1094,88	4,84	1461,76	5,24
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	365,28	8,26	413,82	6,12	401,54	6,11
(gfortran/ifort/ifx)-FFTW-OMP-Debug	764,80	12,37	1258,16	8,66	1603,26	9,86
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	404,22	12,56	452,64	9,67	443,18	6,73
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	2685,84	58,56	3762,24	22,23	4808,76	30,53
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	1416,10	14,49	1337,78	11,95	1341,68	13,61
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637,82	6,94	1039,08	4,62	1365,10	7,19
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296,10	7,05	313,92	7,00	329,92	6,10
(gfortran/ifort/ifx)-MKL-OMP-Debug	747,92	11,61	1236,00	8,43	1559,24	8,95

(gfortran/ifort/ifx)-MKL-OMP-noDebug	388,84	8,49	393,34	9,41	415,96	8,74
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2238,18	5,96	3349,02	5,56	4386,84	10,24
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	986,60	5,21	912,02	6,88	927,08	6,37
Overall Mean	969,48	19,24	1296,91	9,89	1587,03	11,96

Table 1: Influence of gfortran, ifort and ifx on real time execution. In green, mean values where the differences are statistically significant. In blue, when they are not.

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.

An additional noteworthy finding is that the new Intel Fortran compiler, ifx, exhibits slower performance compared to ifort. While ifort is 34% slower 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.

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	7,53	637,82	6,94
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	365,28	8,26	296,10	7,05
gfortran-(FFTW/MKL)-OMP-Debug	764,80	12,37	747,92	11,61
gfortran-(FFTW/MKL)-OMP-noDebug	404,22	12,56	388,84	8,49
gfortran-(FFTW/MKL)-noOMP-Debug	2685,84	58,56	2238,18	5,96
gfortran-(FFTW/MKL)-noOMP-noDebug	1416,10	14,49	986,60	5,21
ifort-(FFTW/MKL)-OMP-DYN-Debug	1094,88	4,84	1039,08	4,62
ifort-(FFTW/MKL)-OMP-DYN-noDebug	413,82	6,12	313,92	7,00
ifort-(FFTW/MKL)--OMP-Debug	1258,16	8,66	1236,00	8,43
ifort-(FFTW/MKL)--OMP-noDebug	452,64	9,67	393,34	9,41
ifort-(FFTW/MKL)-noOMP-Debug	3762,24	22,23	3349,02	5,56
ifort-(FFTW/MKL)--noOMP-noDebug	1337,78	11,95	912,02	6,88
ifx-(FFTW/MKL)-OMP-DYN-Debug	1461,76	5,24	1365,10	7,19
ifx-(FFTW/MKL)-OMP-DYN-noDebug	401,54	6,11	329,92	6,10
ifx-(FFTW/MKL)-OMP-Debug	1603,26	9,86	1559,24	8,95
ifx-(FFTW/MKL)-OMP-noDebug	443,18	6,73	415,96	8,74
ifx-(FFTW/MKL)-noOMP-Debug	4808,76	30,53	4386,84	10,24
ifx-(FFTW/MKL)-noOMP-noDebug	1341,68	13,61	927,08	6,37
Overall Mean	1373,22	18,66	1195,72	7,70

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

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

5.2 Energy consumption by minutes

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 by minutes. 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 (Memeti et al., 2017) and (Pereira et al., 2017). 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 by minute. However, as gfortran seems to be faster, it would lead to less energy consumed overall.

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	3489,23	566,39	3909,33	576,31	3765,81	602,46
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	3902,83	560,89	4462,09	564,98	4522,11	759,47
(gfortran/ifort/ifx)-FFTW-OMP-Debug	4281,07	498,90	4504,28	507,90	4517,33	566,94
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	4469,58	509,26	4613,95	637,51	4517,08	541,76
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3389,82	607,27	3379,13	578,19	3065,26	629,05
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	3388,09	599,72	3220,56	676,94	3025,12	565,11
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	3520,18	482,43	4015,35	545,70	3581,70	460,08
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	4015,48	481,77	4423,95	432,14	4302,50	376,83
(gfortran/ifort/ifx)-MKL-OMP-Debug	4092,32	370,89	4430,80	434,36	4405,47	351,06

(gfortran/ifort/ifx)-MKL-OMP-noDebug	4282,08	416,46	4525,75	567,16	4365,06	351,71
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3018,74	475,80	3444,45	683,74	3069,63	640,62
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3032,50	576,00	3355,50	533,83	2918,67	458,51
Overall Mean	3740,16	516,89	4023,76	566,83	3837,98	539,36

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

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

Experiment	Mean Consumption (J) – FFTW	Std Consumption (J) – FFTW	Mean Consumption (J) – MKL	Std Consumption (J) – MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	3489,23	566,39	3520,18	482,43
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	3902,83	560,89	4015,48	481,77
gfortran-(FFTW/MKL)-OMP-Debug	4281,07	498,90	4092,32	370,89
gfortran-(FFTW/MKL)-OMP-noDebug	4469,58	509,26	4282,08	416,46
gfortran-(FFTW/MKL)-noOMP-Debug	3389,82	607,27	3018,74	475,80
gfortran-(FFTW/MKL)-noOMP-noDebug	3388,09	599,72	3032,50	576,00
ifort-(FFTW/MKL)-OMP-DYN-Debug	3909,33	576,31	4015,35	545,70
ifort-(FFTW/MKL)-OMP-DYN-noDebug	4462,09	564,98	4423,95	432,14
ifort-(FFTW/MKL)-OMP-Debug	4504,28	507,90	4430,80	434,36
ifort-(FFTW/MKL)-OMP-noDebug	4613,95	637,51	4525,75	567,16
ifort-(FFTW/MKL)-noOMP-Debug	3379,13	578,19	3444,45	683,74
ifort-(FFTW/MKL)-noOMP-noDebug	3220,56	676,94	3355,50	533,83
ifx-(FFTW/MKL)-OMP-DYN-Debug	3765,81	602,46	3581,70	460,08
ifx-(FFTW/MKL)-OMP-DYN-noDebug	4522,11	759,47	4302,50	376,83
ifx-(FFTW/MKL)-OMP-Debug	4517,33	566,94	4405,47	351,06
ifx-(FFTW/MKL)-OMP-noDebug	4517,08	541,76	4365,06	351,71
ifx-(FFTW/MKL)-noOMP-Debug	3065,26	629,05	3069,63	640,62
ifx-(FFTW/MKL)-noOMP-noDebug	3025,12	565,11	2918,67	458,51
Overall Mean	3912,37	589,26	3822,23	488,91

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

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 its performance. We observed a loss of repeatability in some of the results files. For instance, using the "diff" command to compare the results files, we found discrepancies such as:

*File1: 0.00000 0.2552937**3**E-08 0.83551334E-09 0.835**53563**E-09*
*File2: 0.00000 0.2552937**6**E-08 0.83551386E-09 0.835**47498**E-09*
Files ./gfortran-FFTW-OMP-DYN-Debug/repli1/pdip.Na2-egs
and ./gfortran-FFTW-OMP-DYN-Debug/repli2/pdip.Na2-egs
are different.

Or:
*File1: 0.00000 0.2552937**4**E-08 0.8355134**5**E-09 0.8354**7134**E-09*
*File2: 0.00000 0.2552937**2**E-08 0.8355150**6**E-09 0.8354**6803**E-09*
Files ./gfortran-FFTW-OMP-Debug/repli1/pdip.Na2-egs
and ./gfortran-FFTW-OMP-Debug/repli7/pdip.Na2-egs
are different.

The differences were relatively minor, approximately from 10^{-7} to 10^{-4}. From a physician 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 out that the issues with repeatability were associated with the FFTW library. We investigated about why FFTW makes us lose repeatability. We found from FFTW documentation "*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 altered the QDD codebase to engage the deterministic mode of FFTW.

5.3.1 Modifying QDD codebase for FFTW repeatability

In this sub-section, we consider the performances 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 equals, as expected.

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	6,29	1173,54	6,66	1534,82	5,11
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	441,18	7,35	494,36	7,41	477,12	4,97
(gfortran/ifort/ifx)-FFTW-OMP-Debug	825,62	10,44	1315,78	8,15	1667,52	9,36

(gfortran/ifort/ifx)-FFTW-OMP-noDebug	458,90	15,73	504,56	18,51	500,46	10,19
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3211,00	14,62	4366,38	71,69	5423,02	94,39
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	2002,36	13,63	1927,26	15,66	1932,06	9,16
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637,66	7,72	1046,90	39,95	1366,24	6,79
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296,00	7,63	314,06	7,31	329,96	6,05
(gfortran/ifort/ifx)-MKL-OMP-Debug	749,96	9,81	1233,46	6,96	1560,56	9,28
(gfortran/ifort/ifx)-MKL-OMP-noDebug	387,78	7,58	396,08	11,06	415,64	9,03
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2262,38	57,07	3375,74	66,67	4425,20	83,64
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	993,64	20,92	911,76	16,12	927,38	10,52
Overall Mean	1087,16	14,90	1421,66	23,01	1713,33	21,54

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

In Table 6, we focus on FFTW and MKL performances, such as in Table 2. The resulting data showed a decline in average performance by approximately 18% for FFTW (1373 seconds in Table 2, and 1613 seconds in Table 6), whereas MKL maintained consistent performance levels, as expected. However, this modification successfully enabled bitwise repeatability across all FFTW experiment replications.

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	6,29	637,66	7,72
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	441,18	7,35	296,00	7,63
gfortran-(FFTW/MKL)-OMP-Debug	825,62	10,44	749,96	9,81
gfortran-(FFTW/MKL)-OMP-noDebug	458,90	15,73	387,78	7,58
gfortran-(FFTW/MKL)-noOMP-Debug	3211,00	14,62	2262,38	57,07
gfortran-(FFTW/MKL)-noOMP-noDebug	2002,36	13,63	993,64	20,92
ifort-(FFTW/MKL)-OMP-DYN-Debug	1173,54	6,66	1046,90	39,95
ifort-(FFTW/MKL)-OMP-DYN-noDebug	494,36	7,41	314,06	7,31
ifort-(FFTW/MKL)--OMP-Debug	1315,78	8,15	1233,46	6,96
ifort-(FFTW/MKL)--OMP-noDebug	504,56	18,51	396,08	11,06

ifort-(FFTW/MKL)-noOMP-Debug	4366,38	71,69	3375,74	66,67
ifort-(FFTW/MKL)--noOMP-noDebug	1927,26	15,66	911,76	16,12
ifx-(FFTW/MKL)-OMP-DYN-Debug	1534,82	5,11	1366,24	6,79
ifx-(FFTW/MKL)-OMP-DYN-noDebug	477,12	4,97	329,96	6,05
ifx-(FFTW/MKL)-OMP-Debug	1667,52	9,36	1560,56	9,28
ifx-(FFTW/MKL)-OMP-noDebug	500,46	10,19	415,64	9,03
ifx-(FFTW/MKL)-noOMP-Debug	5423,02	94,39	4425,20	83,64
ifx-(FFTW/MKL)-noOMP-noDebug	1932,06	9,16	927,38	10,52
Overall Mean	1613,08	18,29	1201,69	21,34

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

Concerning reproducibility between the different experiments, despite these settings adjustments, 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 the different experiments, but we have bitwise identical results between the 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 than when running with ifx- MKL-noOMP-Debug options.

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 main 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 (Charpilloz et al., 2017; Li et al., 2016; Lionel, 2013) to modify compilation options. (Li et al., 2016) 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

It is noteworthy, however, that the options -no-simd and -mp1 were unavailable for the ifx compiler. This might lead to 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. 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 be enhanced similarly. Or 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 a repeatable mode, thereby sacrificing some performance.

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	1157,80	6,02	1210,82	4,41	1537,56	6,19
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	1157,34	6,06	842,94	6,39	728,98	5,38
(gfortran/ifort/ifx)-FFTW-OMP-Debug	1285,22	9,07	1345,42	7,13	1682,96	17,30
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	1283,40	10,45	966,82	8,98	807,32	10,10
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	4489,76	14,01	4833,34	16,80	5517,20	15,78
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	4487,72	12,47	3716,08	24,89	3309,40	135,87
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	1009,60	7,19	1059,08	5,03	1421,32	6,15
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	1011,24	7,96	699,90	5,30	585,90	6,06
(gfortran/ifort/ifx)-MKL-OMP-Debug	1187,14	8,48	1264,46	7,16	1595,02	8,56
(gfortran/ifort/ifx)-MKL-OMP-noDebug	1189,22	8,48	878,12	9,33	727,50	10,08
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3504,38	6,04	3850,58	16,43	4566,86	16,61
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3506,68	7,71	2739,50	7,38	2356,56	23,48
Overall Mean	2105,79	8,66	1950,59	9,93	2069,72	21,80

Table 7: Influence of gfortran, ifort and ifx 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	1157,80	6,02	1009,60	7,19
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	1157,34	6,06	1011,24	7,96
gfortran-(FFTW/MKL)-OMP-Debug	1285,22	9,07	1187,14	8,48
gfortran-(FFTW/MKL)-OMP-noDebug	1283,40	10,45	1189,22	8,48
gfortran-(FFTW/MKL)-noOMP-Debug	4489,76	14,01	3504,38	6,04
gfortran-(FFTW/MKL)-noOMP-noDebug	4487,72	12,47	3506,68	7,71
ifort-(FFTW/MKL)-OMP-DYN-Debug	1210,82	4,41	1059,08	5,03
ifort-(FFTW/MKL)-OMP-DYN-noDebug	842,94	6,39	699,90	5,30
ifort-(FFTW/MKL)--OMP-Debug	1345,42	7,13	1264,46	7,16
ifort-(FFTW/MKL)--OMP-noDebug	966,82	8,98	878,12	9,33
ifort-(FFTW/MKL)-noOMP-Debug	4833,34	16,80	3850,58	16,43
ifort-(FFTW/MKL)--noOMP-noDebug	3716,08	24,89	2739,50	7,38
ifx-(FFTW/MKL)-OMP-DYN-Debug	1537,56	6,19	1421,32	6,15
ifx-(FFTW/MKL)-OMP-DYN-noDebug	728,98	5,38	585,90	6,06
ifx-(FFTW/MKL)-OMP-Debug	1682,96	17,30	1595,02	8,56
ifx-(FFTW/MKL)-OMP-noDebug	807,32	10,10	727,50	10,08
ifx-(FFTW/MKL)-noOMP-Debug	5517,20	15,78	4566,86	16,61
ifx-(FFTW/MKL)-noOMP-noDebug	3309,40	135,87	2356,56	23,48
Overall Mean	2242,23	17,63	1841,84	9,30

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

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 on floating-point arithmetic.

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 that the non-debug program has a different behavior that 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 the exact same order of instructions, as they perform differently.

6 DISCUSSIONS/PERSPECTIVES/LIMITATIONS

Discussion

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 (Gambron & Thorne, 2020), 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 (*Polyhedron benchmarks*, n. d.) and findings by (Young-S et al., 2017). Consequently, we recommend for 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 (Alcott, 2005; Sorrell, 2009) must be considered. It states that the optimization of the energy usage will lead, in the end, to an increase of the energy usage.

The pursuit of faster processing speeds might conflict with the objectives of repeatability and reproducibility, presenting a philosophical dilemma in computing scientific practice. Reproducibility challenges with OpenMP/No-OpenMP and Debug/No-Debug (obtaining bitwise same results from these options) have notably diminished performance, similar to the issues faced with FFTW.

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 the QDD, where the problem is in 3D. However, (Gambron & Thorne, 2020) corroborate our findings.

Furthermore, the comparison of gfortran and ifort performance might vary depending on the application type, as suggested by (Young-S et al., 2017), 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 QDD packages. However, as ifx is new, some optimization might exists 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 CPUs features. The CPU we used in our tests is from 2017 (7 years ago).

Perspectives

Our current study was limited to a single application involving a small Na2+ ion within the QDD framework. Future experiments could expand on these findings to determine the consistency of results across a broader range of applications. 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.

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 a bitwise repeatability. 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 better compilation option for gfortran. This suggests that gfortran, when paired with MKL, is an effective combination for scientific computing aware of performance and energy consumption.

REFERENCES

Alcott, B. (2005). Jevons' paradox. *Ecological economics*, *54*(1), 9-21.

Antunes, B., & Hill, D. R. C. (2024). Reproducibility, Replicability, and Repeatability : A survey of reproducible research with a focus on high performance computing. *arXiv preprint arXiv:2402.07530*.

Chandra, R. (2001). Parallel programming in OpenMP. *Morgan kaufmann*.

Charpilloz, C., Arteaga, A., Fuhrer, O., Harrop, C., & Thind, M. (2017). Reproducible climate and weather simulations : An application to the cosmo model. *Platform for Advanced Scientific Computing (PASC) Conference*.

Dinh, P. M., Vincendon, M., Coppens, F., Suraud, E., & Reinhard, P.-G. (2022). Quantum Dissipative Dynamics (QDD) : A real-time real-space approach to far-off-equilibrium dynamics in finite electron systems. *Computer Physics Communications*, *270*, 108155.

Donoho, D. L., & Stodden, V. (2015). Reproducible research in the mathematical sciences. *The Princeton Companion to Applied Mathematics, Nicholas J. Higham, Mark R. Dennis, Paul Glendinning, Paul A. Martin, Fadil Santosa, and Jared Tanner, editors, Princeton University Press, Princeton, NJ, USA*, 916-925.

Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, *93*(2), 216-231.

Gambron, P., & Thorne, S. (2020). *Comparison of several FFT libraries in C/C++*. STFC.

Hill, D. R. C., Mazel, C., Passerat-Palmbach, J., & Traore, M. K. (2013). Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, *25*(10), 1427-1442.

Li, R., Liu, L., Yang, G., Zhang, C., & Wang, B. (2016). Bitwise identical compiling setup : Prospective for reproducibility and reliability of Earth system modeling. *Geoscientific Model Development*, *9*(2), 731-748.

Lionel, S. (2013). *Improving Numerical Reproducibility in C/C++/Fortran*. SuperComputing, Denver. https://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp129s1.pdf

Marwick, B. (2015). How computers broke science—And what we can do to fix it. *The Conversation*.

Memeti, S., Li, L., Pllana, S., Kołodziej, J., & Kessler, C. (2017). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 1-6.

Nikolić, M., Jović, A., Jakić, J., Slavnić, V., & Balaž, A. (2014). An analysis of FFTW and FFTE performance. *High-Performance Computing Infrastructure for South East Europe's Research Communities: Results of the HP-SEE User Forum 2012*, 163-170.

Noureddine, A. (2022). Powerjoular and joularjx : Multi-platform software power monitoring tools. *2022 18th International Conference on Intelligent Environments (IE)*, 1-4.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages : How do energy, time, and memory relate? *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*, 256-267.

Polyhedron benchmarks. (s. d.). Consulté 19 avril 2024, à l'adresse https://fortran.uk/fortran-compiler-comparisons/polyhedron-benchmarks-linux64-on-intel/

Popper, K. (2005). *The logic of scientific discovery*. Routledge.

Sorrell, S. (2009). Jevons' Paradox revisited : The evidence for backfire from improved energy efficiency. *Energy policy*, *37*(4), 1456-1469.

Young-S, L. E., Muruganandam, P., Adhikari, S. K., Lončar, V., Vudragović, D., & Balaž, A. (2017). OpenMP GNU and Intel Fortran programs for solving the time-dependent Gross–Pitaevskii equation. *Computer Physics Communications*, *220*, 503-506.

AUTHOR BIOGRAPHIES

BENJAMIN ANTUNES is a Phd Student at Clermont Auvergne University (UCA). He holds a Master in Computer Science (head of the list). His thesis subject is about the reproducibility of numerical results in the context of high performance computing. He is especially working on stochastic simulations. His email address is benjamin.antunes@uca.fr and his homepage is https://perso.isima.fr/~beantunes/

CLAUDE MAZEL is Associate Professor since 1989 at Clermont Auvergne University (before 2021, Blaise Pascal university) and he joined the ISIMA Computer Science and Modelling Institute in 1993, where he managed the Computer-Aided Decision Processes, Information and Manufacturing Systems Department. His main scientific interests concern modelling, parallel stochastic simulations and statistical quality of their results. His email address is claude.mazel@uca.fr.

DAVID R.C. HILL is doing his research at the French Centre for National Research (CNRS) in the LIMOS laboratory (UMR 6158). He earned his Ph.D. in 1993 and Research Director Habilitation in 2000 both from Blaise Pascal University and later became Vice President of this University (2008-2012). He is also past director of a French Regional Computing Center (CRRI) (2008-2010) and was appointed two times deputy director of the ISIMA Engineering Institute of Computer Science – part of Clermont Auvergne INP, #1 Technology Hub in Central France (2005-2007 ; 2018-2021). He is now Director of an international graduate track at Clermont Auvergne INP. Prof Hill has authored or co-authored more than 250 papers and has also published several scientific books. He recently supervised research at CERN in High Performance Computing. (https://isima.fr/~hill/ - david.hill@uca.fr)