Reducing the ecological impact of computing through education and Python compilers

Pierre Augier[1], Carl Friedrich Bolz-Tereick[2], Serge Guelton[3], Ashwin Vishnu Mohanan[4] and Camille Noûs[5]

[1] University of Grenoble Alpes, CNRS, Grenoble INP, LEGI, Grenoble, France
e-mail: pierre.augier@univ-grenoble-alpes.fr

[2] Heinrich-Heine-Universität, Düsseldorf, Germany

[3] Namek, Paris, France

[4] Department of Meteorology, Stockholm University, Stockholm, Sweden

[5] Cogitamus laboratory, Earth

We read with attention the comment by Zwart on **The ecological impact of high-performance computing in astrophysics** [1]. We fully agree with its take-home message: scientists should be mindful of their carbon footprint. One of the proposed solutions is to avoid the Python programming language. We advocate that this would be counterproductive and that scientific programs written in Python can be efficient and energy friendly. We argue that advancement of compiler technology, human factors and education are much more important than choice of language.

To support his idea, Zwart presents a benchmark of the N-Body problem with an inefficient implementation in Python, running 50 times slower than a C++ implementation. As Python users concerned about our ecological impact, we worked on similar benchmarks on the same problem. In contrast to Zwart, we (i) consider efficient implementations in Python and Julia and (ii) properly measure the energy consumption with dedicated hardware equipped with wattmeters.

Before focusing on the N-Body problem, let us recall what "Python" is and why it is so successful. Indeed, Python is one of the most used and loved languages for science and data analysis. It is a general-purpose, dynamic language oriented towards communication between humans and fast prototyping. The language was designed to boost productivity. Strong open-source communities use Python and built a rich scientific ecosystem of efficient libraries.

Zwart (2020) characterizes languages as being "interpreted" or "compiled" but these categories make sense only for specific implementations. Some interpreters compile parts of the code during the execution (just-in-time compilation, JIT) [2] and dynamic languages can also be compiled ahead-of-time (AOT). However, the most standard way to execute Python code is to interpret it with the reference implementation called CPython. Due to the lack of a built-in JIT compiler, CPython is relatively slow. Note that this inefficiency of the interpreter has a weak effect on the overall performance of most programs since total elapsed time and energy consumption are often dominated by computations done in optimized libraries. For numerics, the NumPy library [3] is used to describe algorithms with high-level code, which avoids too frequent interactions with the interpreter.

In many cases, very few lines of code dominate the total computation. This is usually known as the 80/20 rule and provides support for two software development principles: (i) "premature optimization is the root of all evil" [4] and (ii) "measure, don't guess". These principles also apply for energy efficiency. For most Python programs, it would be counterproductive and expensive to manually rewrite them in C++, with a small gain/cost ratio.

However, some algorithms require low-level code and explicit loops. For example, for the N-Body problem, the computation of the acceleration of each particle involves a loop over all other particles. Few lines are repeated $N^2/2$ times per time-step. Zwart (2020) considered 10000 time-steps and $N = 16384$, so the program is dominated by 1,342,177,280,000 executions of a simple and inexpensive computation. Using CPython for this hot loop makes the whole program inefficient. Good news for Python: it is straightforward to use efficient alternatives. For our benchmark, we use three tools: (i) Pythran [5], a Python-NumPy AOT compiler transpiling to C++, (ii) Numba [6], a Python-NumPy JIT compiler based on LLVM (same compilation target as Julia) and (iii) PyPy [7], an alternative Python interpreter with a JIT.

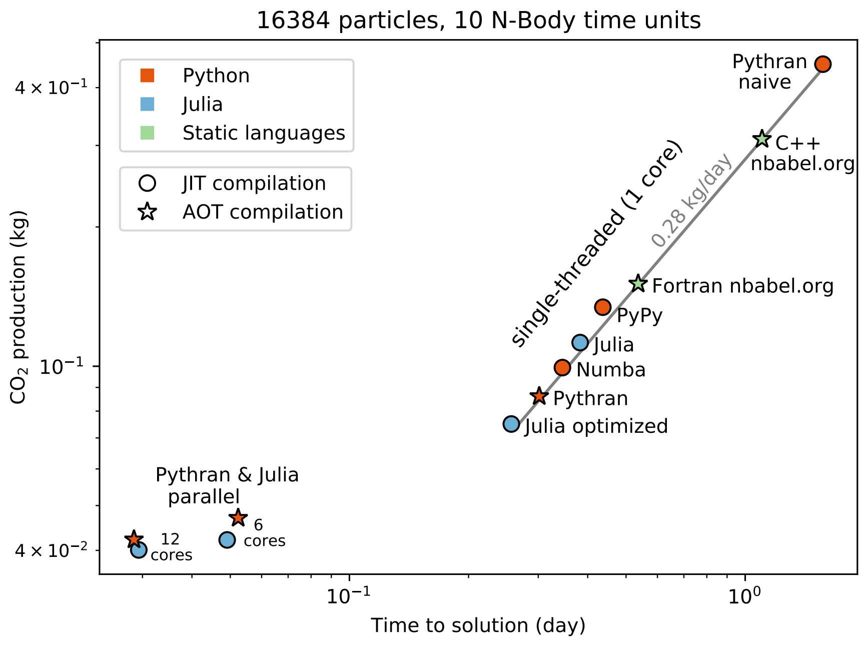

Figure 1: Efficiency in terms of CO_2 production and elapsed time for implementations in Python, Julia, C++ and Fortran. Energy consumption measurements were carried out on Grid'5000 clusters with 2.30 GHz Intel Xeon E5-2630 processors and converted from kWh to CO_2 using 283 g CO_2 / kWh. Optimizations were activated for all implementations with flags like `-Ofast`, `-march=native` and `--check-bounds=no`. We use gcc 8.3.0, Julia 1.5.3, Python 3.8.5, Pythran 0.9.8, Numba 0.52 and an unreleased version of PyPy including optimizations described in [8].

Fig. 1 is similar to Fig. 3 in Zwart (2020): the CO_2 production is plotted as a function of the elapsed time for ten implementations. The C++ and Fortran implementations (green stars) are taken from the website http://www.nbabel.org/ and were used by Zwart (2020). Note that these implementations could have been further optimized, but are representative of code written by many scientists. We consider five implementations in Python (red markers). We would like to emphasize that: (1) These implementations are fully written in Python. The implementations using Pythran and Numba are written in Python-NumPy but NumPy is only used for its arrays as a data-structure and not for high-level functions. (2) Four implementations in Python are faster than the C++ and Fortran implementations. The implementation labelled "Pythran naive" (simple NumPy code accelerated only by decorating one function with `@transonic.jit`

[9]) is only 3 times slower than the Fortran implementation. (3) All Python implementations are simpler to reason about, read and write than the C++ and Fortran implementations.

For comparison, we also consider three implementations in Julia (blue circles): the implementation labelled "Julia" is comparable with the "Pythran" and "Numba" implementations and could have been written by scientists with similar skills. We did not include a Julia implementation similar to "Pythran naive" because it is inefficient. "Julia optimized" and "Julia parallel" have been proposed by Julia users after a long discussion on a Julia forum.

The four points close to the bottom-left corner correspond to two parallel implementations using Pythran+OpenMP and Julia executed using 6 and 12 CPU cores. We consider in Fig. 1 the energy consumption of the cores used (6 or 12 for these runs and 1 for the sequential jobs), which make sense on shared clusters wherein one can reserve only the needed cores. We see that parallelism with threads has only a moderate impact on energy consumption since the increase in power consumption partly counterbalances the decrease of elapsed time. For example, the 12-threaded Pythran version is 10 times faster than the single-threaded one but produces only 2 times less CO_2.

Our work shows that the performance of scientific programs depends less on languages than on time spent on optimization and developer skills to correctly use the right tools. These benchmarks demonstrate that dynamic languages like Python can actually be good solutions to easily obtain good performance while retaining simplicity and readability. We think that minimizing the ecological impact of scientific computing is mainly limited by human factors: time, work, knowledge and skills. For example, scientists have to be able to use job schedulers and shared clusters optimized in terms of energy consumption. They should know how to profile their code to discover which parts can potentially be optimized. Therefore, time and money should be invested in education and tooling to minimize the overall ecological impact of computing, irrespective of the underlying language.

Acknowledgements

We thank the open-source communities who contributed to the tools used for this study. We thank Darley Barreto, Leandro Martínez and Andrey Oskin for working on optimized Julia implementations. Part of this work was performed using resources provided by Grid'5000 and CINES under GENCI allocation number A0080107567.

Competing interests

The authors have no competing interests to declare.

References

[1] Zwart, S. P. The ecological impact of high-performance computing in astrophysics. *Nature Astronomy* **4**, 819–822 (2020).

[2] Aycock, J. A brief history of just-in-time. *ACM Computing Surveys (CSUR)* **35**, 97–113 (2003).

[3] Harris, C. R. *et al.* Array programming with NumPy. *Nature* **585**, 357–362 (2020).

[4] Knuth, D. E. Structured programming with go to statements. *ACM Computing Surveys (CSUR)* **6**, 261–301 (1974).

[5] Guelton, S. *et al.* Pythran: Enabling static optimization of scientific Python programs. *Computational Science & Discovery* **8**, 014001 (2015).

[6] Lam, S. K., Pitrou, A. & Seibert, S. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6 (2015).

[7] Bolz, C. F., Cuni, A., Fijalkowski, M. & Rigo, A. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 18–25 (2009).

[8] Cheng, L., Ilbeyi, B., Bolz-Tereick, C. F. & Batten, C. Type freezing: exploiting attribute type monomorphism in tracing JIT compilers. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 16–29 (2020).

[9] Augier, P., Mohanan, A. V. & Bonamy, C. FluidDyn: A Python open-source framework for research and teaching in fluid dynamics by simulations, experiments and data processing. *Journal of Open Research Software* **7** (2019).