The ecological impact of high-performance computing in astrophysics

Computer use in astronomy continues to increase, and so also its impact on the environment. To minimize the effects, astronomers should avoid interpreted scripting languages such as Python, and favour the optimal use of energy-efficient workstations.

Simon Portegies Zwart

The third pillar of science, simulation and modelling, already had a solid foothold in fourth-century astronomy[1], but this discipline flourished with the introduction of digital computers. One of its challenges is the carbon emission resulting from this increased popularity. Generally unrecognized, the magnitude of the carbon footprint of computing in astrophysics should be emphasized. One purpose of this Comment is to raise this awareness, and present best practices for (super)computer usage and choice of programming language.

Carbon footprint of computing

In Fig. 1, we compare the average human production of CO_2 (red lines) with astronomical activities, such as telescope operation, the emission of an average astronomer[2] and finishing a (four year) PhD[3] (green points). While large observing facilities are cutting down on carbon footprint by offering remote operation, the increased speed of computing resources can hardly be compensated by their increased efficiency. This also is demonstrated in Fig. 1, where we compare measurements for several popular astronomical computing activities (turquoise points). These measurements are generated using the Astrophysical Multipurpose Software Environment[4], in which most of the work is done in optimized and compiled code. We include simulations of the Sun's evolution from birth to the asymptotic giant branch using a Henyey solver[5] and parametrized population synthesis[6].

We also present in Fig. 1 timings for simulating the evolution of a self-gravitating system of a million equal-mass point-particles in a virialized Plummer sphere for 10 dynamical timescales (labelled '*N*-body'). These calculations are performed by direct integration (with the fourth-order Hermite algorithm) and using a hierarchical tree-code (with leapfrog algorithm). Both calculations are performed on a CPU as well as with a graphics processing unit (GPU). Not surprisingly, the tree-code running a single GPU (second turquoise point from the left) is about a million times faster than the direct-force calculations on a CPU (right-most turquoise point); one factor

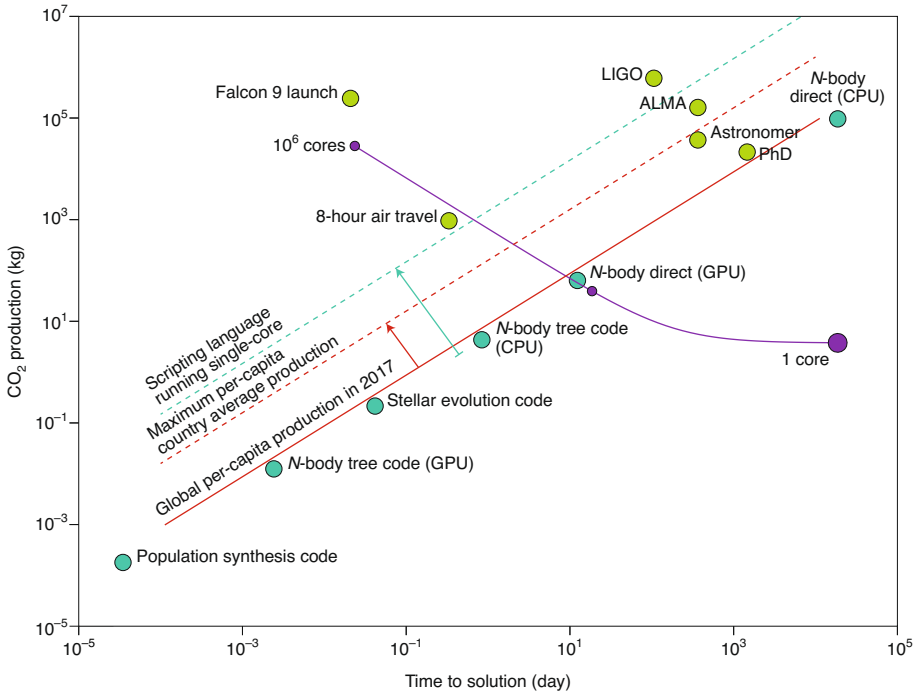

Fig. 1 | Carbon production of a number of common activities among astronomers. CO_2 production as a function of the time to solution for a variety of popular computational techniques employed in astrophysics (turquoise data points), and other activities common among astronomers[2,3] (green data points). The solid red curve gives the individual world-average production in 2017, whereas the dotted red curve give the maximum per-capita country average. The Laser Interferometer Gravitational-Wave Observatory (LIGO) carbon production is taken over its first 106-day run (using ~180 kW)[17], and for the Atacama Large Millimeter/submillimeter Array (ALMA) a 1-year average[18]. A Falcon 9 launch lasts about 32 minutes during which ~110,000 litres of highly refined kerosene is burned. The tree-code running on a GPU was performed using $N = 2^{20}$ particles. The direct *N*-body code on a CPU (right-most turquoise data point) was run with $N = 2^{13}$ particles[15], and the other codes with $N = 2^{16}$ particles. All performance results were scaled to $N = 2^{20}$ particles. The calculations were performed for 10 *N*-body time units[19]. The energy consumption was computed using the scaling relations of ref. [20] and converted from KWh to CO_2 using 0.283 kWh kg^{-1}. The turquoise dotted curve shows the estimated carbon emission when these calculations would have been implemented in Python running on a single core. The burgundy curve shows how the performance and carbon production changes while increasing the number of compute cores from 1 to 10^6 (out of a total of 7,299,072 of the world's fastest computer, left-most point) using the performance model of ref. [21]. Figure created with Matplotlib[22].

of 1,000 originates from the many cores of the GPU[7], and the other from the favourite scaling of the tree algorithm[8]. The trend in carbon production is also not surprising: a shorter runtime leads to less carbon. The emission of carbon while running a powerful workstation is comparable to the world's per-capita average.

Now consider the single-core versus multi-core performance of the direct *N*-body code in Fig. 1 (burgundy line). The turquoise data point to the right gives the single-core workstation performance, but the burgundy data point below it shows the single-core performance on today's largest supercomputer. The curve gives the multi-core scaling up to 10^6 cores (left-most data point). The relation between the computing time (time to solution) and the carbon footprint of the calculations is not linear. When running a single core, the supercomputer produces less carbon than a workstation (we assumed the supercomputer to be used to capacity by other users). Adopting more cores results in better performance, at the cost of producing more carbon. Similar performance to a single GPU is reached when running 1,000 cores, but when the number of cores is further increased, the performance continues to grow at an enormous cost in carbon production. When running a million cores, the emission of running a supercomputer by far exceeds air travel and approaches the carbon footprint of launching a rocket into space.

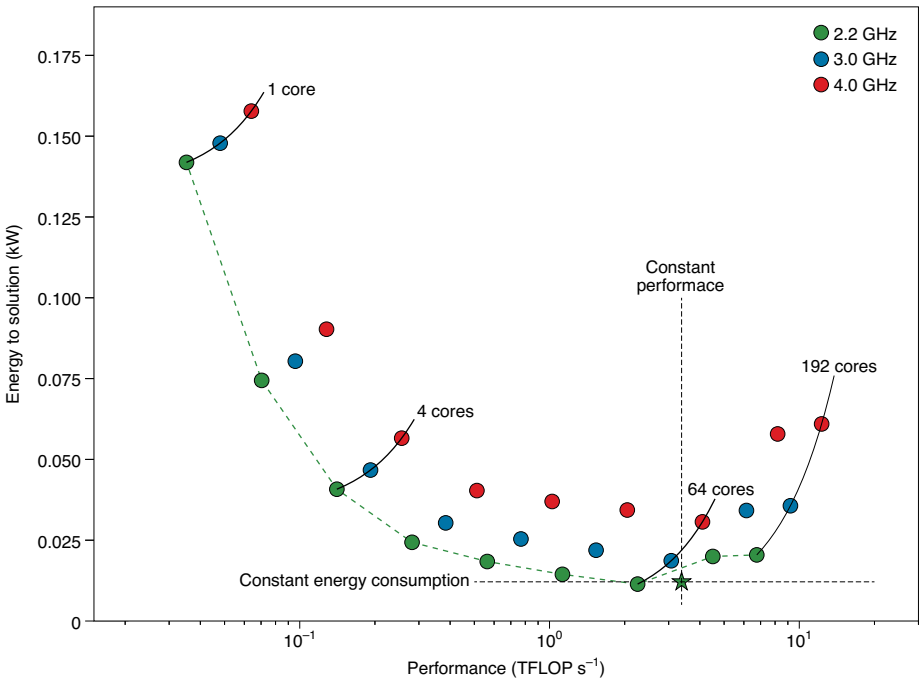

Fig. 2 | Energy to solution as a function of code performance. The Z-plot shows the number of processors (and processor frequencies) and the energy consumed as a function of performance[20]. The runs (green dots) were performed using a quad CPU 24-core (48 hyperthreaded) Intel Xeon E7-8890 v4 at 2.20 GHz calculated with 1, 2, 4, ..., 192 cores. Curves of constant core-count are indicated for 1, 4, 64 and 192 cores (black curves). The other coloured points (blue and red) give the relation for overclocking the processor to 3 GHz and 4 GHz, scaled from the measured points using overclocking emission relations from ref. [23]. Dotted curves give constant energy-requirement-to-solution (horizontal) and sustained processor performance (vertical). The star at the crossing of these two curves was measured using 96 physical cores. The calculations were performed using a Bulirsch–Stoer algorithm with a Leapfrog integration[24] at a tolerance in the energy E of $dE/E = 10^{-8}$ using a word length of 64 bits.

Concurrency for lower emission

When parallelism is optimally utilized, the highest performance is reached for the maximum core count, but the optimal combination of performance and carbon emission is reached for ~1,000 cores, after which the supercomputer starts to produce more carbon than a workstation. The improved energy characteristics for parallel operation and its eventual decline is further illustrated in the Z-plot presented in Fig. 2, showing energy consumption as a function of the performance of a 96-core (192 hyperthreaded) workstation.

Running single core on a workstation is inefficiently slow and produces more carbon than running multi-core. Performance continues to increase with core count until optimal energy consumption is reached when all physical cores are occupied (in Fig. 2 this happens around 96 physical cores, indicated by the green star). Runtime continues to drop when also using virtual cores, but at the cost of higher emission. Note that the carbon emission of the parallel calculation (burgundy curve in Fig. 1) does not drop with increased core count, because we assumed that the supercomputer is shared, whereas we assumed that the workstation used in Fig. 2 was private.

Scaling our measurements of the compute performance and energy consumption with the clock frequency of the processor (blue and red points for each core count) reduces wall-clock time, but costs considerably more energy (see also ref. [9]). Although not shown here, reducing the clock speed slows down the computer while increasing the energy requirement.

If the climate is a concern, we should prevent loading a supercomputer to capacity. The wish for more environmentally friendly supercomputers triggered the contest to find the greenest supercomputers[10]. Since the inauguration of the so-called Green500 ranking, the performance per watt has increased from 0.23 TFLOP kW^{-1} by a Blue Gene/L in 2007 (ref. [10]) to more than 20 TFLOP kW^{-1} by the MN-3 core server today (https://www.top500.org/lists/green500/). This enormous increase in efficiency is mediated by the further development of low-power many-core architectures, such as the GPU. The efficiency of modern workstations, however, has been lagging. A single core of the Intel Xeon E7-8890, for example, runs at ~4 TFLOP kW^{-1}, and the popular Intel core-i7 920 tops only 0.43 TFLOP kW^{-1}. Workstation processors have hardly kept up with the improved carbon characteristics of GPUs and supercomputers.

For optimal operation, run a few (~1,000) cores on a supercomputer or a GPU-equipped workstation. When running a workstation, use as many physical cores as possible, but leave the virtual cores alone. Over-clocking reduces wall-clock time but at a greater environmental impact.

The role of language on the ecology

So far, we assumed that astrophysicists invest in full code optimization that uses the hardware optimally. However, in practice, most effort is generally invested in developing the research question, after which designing, writing and running the code is not the primary concern. This holds as long as the code writing and execution are sufficiently fast. As a consequence,

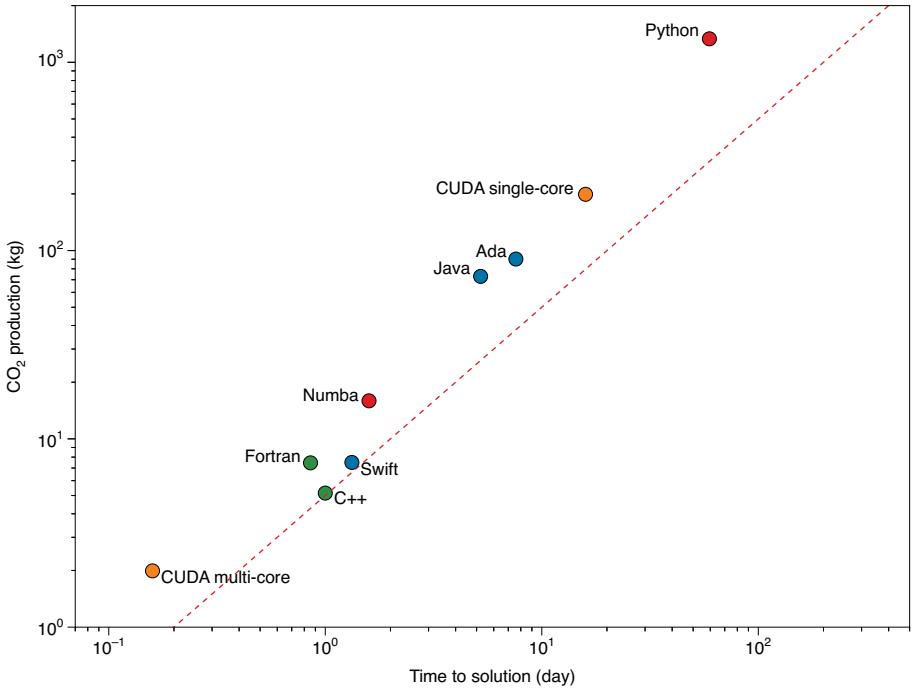

Fig. 3 | Programming language efficiency as a function of the time to solution. Here we used the suite of direct *N*-body codes from http://www.NBabel.org/ to measure execution speed and to determine the relative energy efficiency for each programming language using Table 3 of ref. [12]. Different language families are indicated with colours: green and blue for third-generation languages, orange for special-purpose and red for scripting languages. The dotted red curve gives a linear relation between the time to solution and carbon footprint (~5 kg CO_2 per day). The calculations were performed on a 2.7 GHz Intel Xeon E-2176M CPU and NVIDIA Tesla P100 GPU.

interpreted scripting languages, such as Python[11], rapidly grow in popularity.

According to the Astronomical Source Code Library (ASCL, https://ascl.net/), about half of the deposited code is written in Python, Java, IDL or Mathematica. Only 18%, 17% and 16% of codes are written in Fortran, C and C++, respectively. Python is popular because it is interactive, strongly and dynamicaly typed, modular, object-oriented and portable. But most of all, Python is easy to learn and it gets the job done without much effort, whereas writing in C++ or Fortran can be rather elaborate.

One major disadvantage of Python, however, is its relatively slow speed compared with compiled languages. In Fig. 3, we present an estimate of the amount of CO_2 produced when performing a direct *N*-body calculation of 2^{14} equal-mass particles in a virialized Plummer sphere. Each calculation was performed for the same amount of time and scaled to 1 day for the implementation in C++.

Python (and to a lesser extent Java) takes considerably more time to run and produces more CO_2 than C++ or Fortran. Python and Java are also less efficient in terms of energy per operation than compiled languages[12], which explains the offset away from the dotted curve (Fig. 3).

The popularity of Python is disquieting. Among 27 tested languages, only Perl and Lua are slower[12]. The runtime performance of Python can be improved in a myriad of ways. Most popular are the Numba[13] or NumPy[14] libraries, which offer pre-compiled code for common operations. In principle, Numba and NumPy can lead to an enormous increase in speed and reduced carbon emission. However, these libraries are rarely adopted for reducing runtime by more than an order of magnitude (according to the ASCL). NumPy, for example, is mostly used for its advanced array handling and support functions. Using these will reduce runtime and, therefore, also carbon emission, but optimization is generally stopped as soon as the calculation runs within an unconsciously determined reasonable amount of time, such as the coffee-refill timescale or a holiday weekend.

In Fig. 1 we presented an estimate of the carbon emission as a function of runtime for Python implementations (see turquoise dotted curve) of popular applications. The continuing popularity of Python should be juxtaposed with the ecological consequences. We teach Python to students, and researchers accept the performance punch without realizing the ecological impact. Using C++ and Fortran instead of Python would save enormously in terms of runtime and CO_2 production. Another reason why students should not learn to program using an inefficient language running on slow hardware but start with the fastest language on the biggest computers is provided by Jevons paradox. This paradox implies that a gradual increase in efficiency or capacity leads to a resource's oversubscription without added benefit: a gradual increase in computer speed and language efficiency lead to sub-optimal code.

Conclusions

The popularity of computing in research is proliferating. This impacts the environment by increased carbon emission. Running scripts on a single core of a powerful workstation is not environmentally friendly. Still, this mode of operation seems to be most popular among astronomers, as it is stimulated by the educational system and mediated by the ease of use and the availability of the hardware. This trend leads to an unnecessarily large carbon footprint for computationally oriented astrophysical research. The importance of rapid prototyping seems to outweigh the ecological impact of inefficient code.

The carbon footprint of computational astrophysics can be reduced substantially by running on GPUs. The development time of such code, however, requires considerable expertise, and it takes years of research[15] before a tuned instrument is production-ready[16].

As an alternative, one could run concurrently using multiple cores, rather than a single thread. It is important to share resources and to prevent the monopolization of powerful workstations. To reduce runtime and CO_2 emission, the environmentally concerned researcher might want to reconsider standard Python and either optimize using high-performance libraries or adopt a more environmentally friendly (compiled) alternative. Several interesting alternatives exist, such as Alice, Julia, Rust and Swift. These languages offer the flexibility of Python but with the performance of compiled C++. Educators may want to reconsider teaching Python to university students.

But maybe, while being aware of the ecological impact of high-performance computing, we should be more reluctant to perform specific calculations, and consider the environmental consequences

before starting a simulation. Scientists have a responsibility in assuring that their computing set-up is mostly harmless to the environment. ❐

Simon Portegies Zwart ✉
Leiden Observatory, Leiden University, Leiden, the Netherlands.
✉*e-mail: spz@strw.leidenuniv.nl*

Published online: 10 September 2020
https://doi.org/10.1038/s41550-020-1208-y

References

1. Ossendrijver, M. *Science* **351**, 482–484 (2016).
2. Stevens, A. R. H., Bellstedt, S., Elahi, P. J. & Murphy, M. T. *Nat. Astron.* https://doi.org/10.1038/s41550-020-1169-1 (2020).
3. Achten, W. M., Almeida, J. & Muys, B. *Ecol. Indic.* **34**, 352–355 (2013).
4. Portegies Zwart, S. & McMillan, S. *Astrophysical Recipes: The Art of AMUSE* (IOP Publishing, 2018).
5. Paxton, B. et al. *Astrophys. J. Suppl. Ser.* **192**, 3–38 (2011).
6. Portegies Zwart, S. F. & Verbunt, F. *Astron. Astrophys.* **309**, 179–196 (1996).
7. Gaburov, E., Bédorf, J. & Portegies Zwart, S. *Procedia Comput. Sci.* **1**, 1119–1127 (2010).
8. Barnes, J. & Hut, P. *Nature* **324**, 446–449 (1986).
9. Hofmann, J., Hager, G. & Fey, D. P. In *High Performance Computing* (eds Yokota, R. et al.) 22–43 (Springer, 2018).
10. Feng, W. & Cameron, K. *Computer* **40**, 50–55 (2007).
11. Van Rossum, G. & Drake, F. L. Jr *Python Reference Manual* (Centrum voor Wiskunde en Informatica, 1995).
12. Pereira, R. et al. In *Proc. 10th ACM SIGPLAN Int. Conf. on Software Language Engineering SLE 2017* 256–267 (Association for Computing Machinery, 2017).
13. Lam, S. K., Pitrou, A. & Seibert, S. In *Proc. Second Workshop on the LLVM Compiler Infrastructure in HPC LLVM '15* 1–6 (Association for Computing Machinery, 2015).
14. Oliphant, T. E. *A Guide to NumPy* Vol. 1 (Trelgol Publishing, 2006).
15. Portegies Zwart, S. F., Belleman, R. G. & Geldof, P. M. *New Astron.* **12**, 641–650 (2007).
16. Bédorf, J. et al. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis SC '14* 54–65 (IEEE, 2014).
17. *Advanced LIGO Reference Design: LIGO M060056-v2* (LIGO Scientific Collaboration, 2011).
18. D'Addario, L. in *Proceedings of Exascale Radio Astronomy* 302.01 (American Astronomical Society, 2014).
19. Heggie, D. C. & Mathieu, R. D. in *The Use of Supercomputers in Stellar Dynamics* (eds Hut, P. & McMillan, S. L. W.) 233–235 (Springer, 1986).
20. Wittmann, M., Hager, G., Zeiser, T., Treibig, J. & Wellein, G. *Concurr. Comput. Pract. Exper.* **28**, 2295–2315 (2016).
21. Heinrich, F. C. et al. in *IEEE International Conference on Cluster Computing* 92–102 (IEEE, 2017).
22. Hunter, J. D. *Comput. Sci. Eng.* **9**, 90–95 (2007).
23. Cutress, I. The Intel Xeon W-3175X review: 28 unlocked cores, $2999. *AnandTech* https://www.anandtech.com/show/13748 (30 January 2019).
24. Portegies Zwart, S. & Boekholt, T. *Astrophys. J. Lett.* **785**, L3–L7 (2014).

Acknowledgements

I thank A. Allen for providing data on the ASCL language usage and L. Butscher for comments and inviting me to present this discussion at the 2020 European Astronomical Society conference. Part of this work was performed using resources provided by the Academic Leiden Interdisciplinary Cluster Environment (Alice), TITAN (LANL) and LGM-II (NWO grant number 621.016.701) We used the Python, Matplotlib, NumPy, Numba and AMUSE open-source packages.

Competing interests

The author declares no competing interests.