CuttleBench: A benchmarking tool for comparing programming languages' performances

Elana Courtines, Georges da Costa, Patricia Stolf

▶ To cite this version:

Elana Courtines, Georges da Costa, Patricia Stolf. CuttleBench: A benchmarking tool for comparing programming languages' performances. Toulouse 3 Paul Sabatier. 2023. hal-04610856

HAL Id: hal-04610856
https://hal.science/hal-04610856

Submitted on 13 Jun 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

CuttleBench: A benchmarking tool for comparing programming languages' performances

Courtines Elana[1], Da Costa Georges[1], and Stolf Patricia[1]

[1] *IRIT, SEPIA Team, Université Toulouse III - Paul Sabatier*

August 2023

Abstract

Energy consumption has been of growing concern in the past few years. Studies have compared the impact of programming languages on energy consumption and performances, and while they generally tend to agree that the most efficient language is CUDA, and that the least efficient one is Python, they have been heavily criticised for their lack of methodology, with some critics saying that the comparison was about the ability of the programmer to code rather than the language itself. With how important energy management has become, we believe that it is important to establish a clear comparison between programming languages so that developpers may be aware of the consequences of their choice when picking a programming language over another.

In this work, we developped a new benchmarking framework that allows for the fair comparison of the energy consumption of programming languages. This project did not yield any concrete results per se, but the first partial results suggest that, if execution time is not factored in, power consumption is fairly equivalent between every programming languages.

1 Introduction

The goal of this project is to improve the quality and the reproducibility of the previous studies on the impact of programming languages, all the while studying their energy consumption, which was mostly disregarded. To that end, a new benchmarking framework named CuttleBench was developed.

In this work, we assess the state of the art of programming languages comparisons in order to provide a proper methodology for how to compare programming languages with fairness. Using the newly defined methodology, we develop a benchmarking framework that allows for fair comparisons while ensuring that the framework is portable and that the results can easily be reproduced.

The main contributions of this work are the following:

- A framework to run multiple implementations of multiple benchmarks and to acquire energy, system and performance runtime data;
- A first set of benchmarks in multiple languages;
- A comparison of the energy and performance impact of programming languages.

The remainder of the paper is organised as follows: section 2 presents a brief descriptions of existing works. Section 3 provides information regarding the methodology followed in this present work. Section 4 presents the specificity of the experiments carried out in this work. Finally, section 5 presents the results obtained along with an analysis before concluding in Section 6.

2 State of the Art

2.1 Methodological approaches

In [2], the authors performed a comparison of the implementations of three algorithms in six different programming languages with a focus on their bioinformatics usage, on two different operating systems. Determining which languages and which operating systems are the best suited for this field would indeed be interesting, unfortunately, because the source files are not accessible, it is impossible to ascertain that the comparisons were fair, and the results cannot be reproduced.

In [7], the authors compared seven programming languages after gathering numerous implementations from computer science master students in a controlled experiment. While the way the implementations were gathered is interesting and ensures that they are effectively idiomatic, the results of this research are now outdated due to deprecated compiler versions, depriving them of their meaning.

In [5], the authors extend their work from [4] where they compare numerous algorithms with numerous programming languages based on implementations taken from The Computer Language Benchmarks Game. The results of this study became relatively famous and eventually led to various reactions in social medias regarding their methodology, often stating that the results are biased and that the study is comparing the programmers themselves rather than the languages. For some algorithms, they simply took the fastest implementation available without ensuring the equity between the different languages, leading to some comparisons having multi-threads code on one side, and single-thread code on the other side, for example.

In [6], the authors present results of high performance computing in astrophysics. They attempt to show how the performance of the computer relates to the power consumption, which is indeed of great concern when it comes to highly energy-intensive research fields. Unfortunately, because the source files are not accessible, it is impossible to reproduce the results and ascertain that they are correct.

2.2 Monitoring tools

In [3], the authors compare the different power-meters available to determine which ones are the most suited for which job. Their result seem to suggest that most software-based power-meters are roughly equivalent, but that having both

a software-based power-meter and a physical on-site power-meter is ideal for power consumption analysis.

3 Methodology

After analysing previous studies on energy consumption and programming language efficiency, a methodology to follow in order to develop the benchmarking framework was decided.

3.1 Gathering of algorithms and implementations

Despite the issues surrounding the use of The Computer Language Benchmarks Game[1] for research purposes, it was still decided to use some of its implementations to serve as a base for CuttleBench. In order to avoid unfair comparisons between languages, several criteria were agreed upon to select implementations. If the available implementations did not meet the criteria, modifications were brought to the source code until the criteria were met.

- Parallelism: while it is true that some languages heavily rely on parallelism to be efficient, it was deemed necessary to carry out the first experiments with no implementations making use of parallelism;

- External libraries: because the goal of the project is to compare languages themselves and not the ability of someone to code something that is efficient; the use of external libraries was prohibited so that every language would have to rely exclusively on their standard library;

- Idiomatic and simple code: the definition of idiomatic is very subjective, but to avoid unfair comparisons, implementations were chosen to match what an average programmer of the language would produce without excessive optimisations.

The languages and algorithms were selected based on more personal capabilities to understand the code while providing varied enough algorithms and languages to bench. A total of two algorithms were selected, one that focuses on memory usage, and one that focuses on raw computation, while a total of four languages were selected (see section 4).

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

3.2 The framework

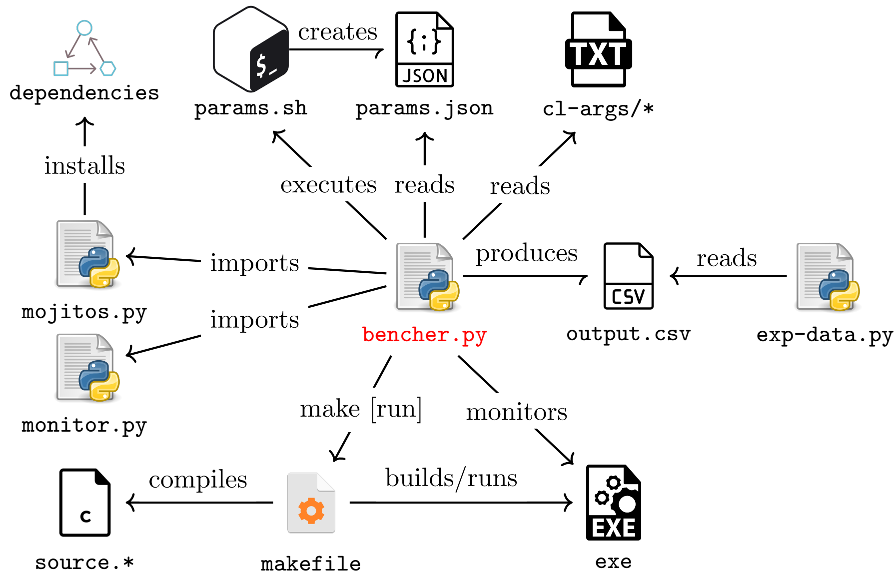

Figure 1: CuttleBench inner workings

Figure 1 presents a general idea of how CuttleBench functions internally. The main script, `bencher.py`, imports and installs all the necessary dependencies (excluding the installation of languages), reads all the configuration files and compiles and executes all the specified sources through the Makefiles. See the official documentation for more details about command-line usage[2].

Implementations are divided within CuttleBench by a folder for each algorithm, then further subdivided by a folder for each language. Each algorithm folder contains an input file and an output file (for result ascertainment), and each language folder contains the source file for the implementation along with a dedicated Makefile.

Additionally, several scripts were made to ease the setup work:

1. `gen-parameters.sh` is a script that generates all the possible options for the files in the `cl-args/` folder based on the implementations currently available in the underlying folders;

2. `launch.py` is a script dedicated to launching CuttleBench from within the Grid5000[3] infrastructure by submitting a dedicated job to run `bencher.py`. Using this script requires manually setting up the

[2] https://gitlab.irit.fr/sepia/stages/23-programming-languages

[3] Grid'5000 is a large-scale and flexible testbed for experiment-driven research in all areas of computer science. https://www.grid5000.fr/

`g5k-input.txt` file to specify which site and cluster to submit the job to;

3. `exp-data.py` is a result generator script that can read some monitoring tools' output files to generate various energy consumption summary files as CSV (see section 4.4)

3.3 Obtaining metrics on a particular benchmark

Before the monitoring of every benchmark, the *bencher* goes through a preheating phase (a CPU intensive phase, see section 4) along with a sleeping phase in an attempt to reproduce the same hardware state for every execution. It is of course impossible to guarantee that the hardware state will be exactly the same, but this step helps reduce inconsistencies observed in early results.

Execution time can vary greatly between different languages. Because it is important to keep the same inputs for every program, some of them end up being too fast to be monitored properly. In order to address this issue, the *bencher* automatically executes benchmarks that did not last long enough until the total execution time exceeds a specific threshold, all the while storing how many times the program was executed. Results thus show an additional column stating how many executions were monitored. That allows monitoring tools to have a sufficient amount of data for the results to be relevant.

A potential limit of this approach concerns the difference between running some benchmarks once (the slow ones) against running some benchmarks several time (the fast ones), as the subsequent executions of the faster benchmarks will effectively not start in the same initial hardware state as the slower benchmarks.

3.4 Monitoring tools

Most of the available software-based power-meters available being relatively equivalent[3], only the most relevant and easy to access RAPL power-meters were used. Physical power-meters were also used. Both measures are interesting as their scope is different: RAPL measures processor and memory power consumption which is usually assumed to be the most important dynamic element for the power consumption of servers, while whole-server power-meter evaluate the global power impact of the different executions, taking into account all elements inside the servers (PSU, motherboard, NIC, ...).

3.5 Outputs

The framework delegates the production of monitoring data to the monitors. The experiments datas are stored within CSV files for RAPL monitoring and JSON files for Grid5000's physical power-meters.

Taking these monitoring data files as inputs, the framework can produce many types of time-series results in the form of CSV files or graphs with different degrees of simplification as well as summaries.

4 Experiments

The CuttleBench Benchmarking tool was mainly implemented under Python 3.9.2 using the Expetator monitoring framework[4] (v0.3.20).

To ensure that programs are benched in the same conditions, they are all executed in the exact same way with the same inputs through a dedicated Makefile. While a universal Makefile was not achieved, the dedicated Makefiles were made as similar as possible.

A leverage–a function used before every execution benched with Expetator–also heats up the CPU and forces it to sleep before executing every program to ensure that the hardware conditions within which they are run are as similar as possible between the different experiments.

4.1 Experimental environment

Grid'5000 is a large-scale and flexible testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data and AI. All experiments were carried out on the Yeti Cluster from the Grenoble site[5].

This cluster was specifically selected because the associated machines all possess an accurate external physical power-meter required for this project. The machines have the following specifications:

- System model: Dell PowerEdge R940
- CPU: Intel Xeon Gold 6130 (Skylake, 2.10GHz, 4 CPUs/node, 16 cores/CPU)
- Memory: 768 GiB.
- External power-meter: Omegawatt[6] devices with a measurement frequency up to 50Hz and a precision below 1W.

Using the Grid5000 infrastructure also ensure that the experiments are actually reproducible by anyone who has access to the infrastructure.

4.2 Algorithms

A total of four algorithms/functions were selected to carry out the first experiments.

- Binary-Trees[7]: a memory usage focused algorithm;
- Fannkuch-Redux[8]: a raw computation focused algorithm on a relatively

[4] https://expetator.readthedocs.io/en/latest/index.html
[5] https://www.grid5000.fr/w/Grenoble:Hardware
[6] https://mv.omegawatt.fr/
[7] Binary-Trees on The CLBG
[8] Fannkuch-Redux on The CLBG

small objects;

- Sleep: a basic function calling Python 3.9.2's `time.sleep()` method;
- Stress: a basic function calling Linux's `stress` command (v1.0.4) on a single CPU core.

4.3 Languages and implementations

In order to achieve equity between the different languages, multi-threading and multi-processing has been prohibited. As such, none of the implementations make use of parallel programming. If a comparison of energy consumption is to be done for parallel implementations, then every single implementation should be parallel to make sure equity is maintained, or, the implementation itself should be marked as parallel so as to make sure it is known to be parallel when compared to other implementations.

While a wider range of languages could have been selected, the project was limited by the author's personal programming languages experience. As such, only four languages were selected, with the following specificity:

- C: a general-purpose compiled language. C implementations were restricted to using only the standard library. All experiments were done under gcc (Debian 10.2.1-6) 10.2.1 20210110 and using the -O3 flag only;
- Java: a high-level, class-based, object-oriented programming language. Java implementations were made within a single class containing the main method as well as other nested classes and methods as necessary. All experiments were done with javac 17.0.7 and command line compilation using no additional flag;
- Python: a general-purpose, high-level interpreted programming language. Python implementations are written as purely iterative with no object definition. All experiments were done with Python 3.9.2;
- OCaml: a general-purpose, high-level multi-paradigm programming language. OCaml implementations were written as purely functional using no external modules. All experiments were done under The OCaml toplevel, version 4.10.2 using the ocamlopt compiler.

It should be noted that OCaml implementations were not benched properly due to lack of time and OCaml not being available by default on the Grid5000 machines. They are thus **not included** in the results of this article.

4.4 Monitors

While a wider range of monitors could have been selected, this project limited itself to two RAPL[9] power-meters and one physical power-meter.

- MojitO/S[10][1]: is *An Open Source System, Energy and Network Monitoring Tool at the O/S level* developed at IRIT by G. Da Costa. MojitO/S outputs time-series in the form of a CSV file.
- Lperf[11]: is a modified version of the Linux Perf Tool[12] for use with the Expetator tool developed at IRIT by G. Da Costa. Lperf outputs time-series in the form of a CSV file.
- Kwollect[13]: a framework available on the Grid5000 infrastructure to collect data from physical wattmeters. Kwollect outputs time-series in the form of a Json file.

Due to the size of the data files produced by the Lperf tool, it was eventually decided to stop its usage despite the monitor working properly, MojitOS becoming the only monitor used for RAPL data. As mentionned previously, most RAPL power-meters are almost equivalent in terms of results, and so this decision was not deemed impactful.

Regardless, it is still possible to select Lperf as a monitor when launching experiments with CuttleBench.

5 Results

The first results obtained with CuttleBench are displayed in this section. They concern the power consumption and the total energy consumed for every language and algorithm combination, for both RAPL and Kwollect.

The first two figures of the sections 5.1 (figure 2 and figure 3) and 5.2 (figure 6 and figure 7) present both algorithms together along with the Sleep and Stress values for RAPL first, and then for Kwollect. Sleep and Stress supposedly having a constant power consumption, they are represented in the form of a straight line, while the values for each language and algorithm combination are the average values obtained from the experiments. Logarithmic scaling on the axis is used to make the results more readable.

The two next figures of each section (figure 4 and figure 5) (figure 8 and figure 9) present the power consumption evolution over time for both algorithms separately.

[9] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

[10] https://gitlab.irit.fr/sepia-pub/mojitos

[11] https://gitlab.irit.fr/sepia-pub/expetator/-/blob/master/expetator/monitors/lperf.py

[12] https://perf.wiki.kernel.org/index.php/Main_Page

[13] https://www.grid5000.fr/w/Monitoring_Using_Kwollect

5.1 RAPL Results

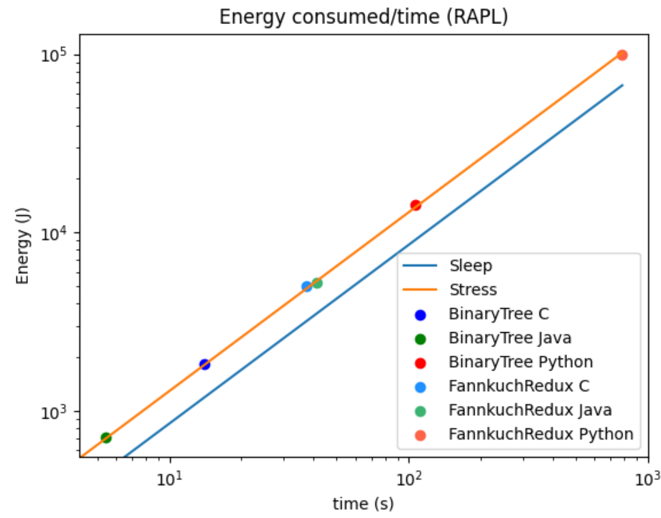

Figure 2: Energy Consumed summary (RAPL)

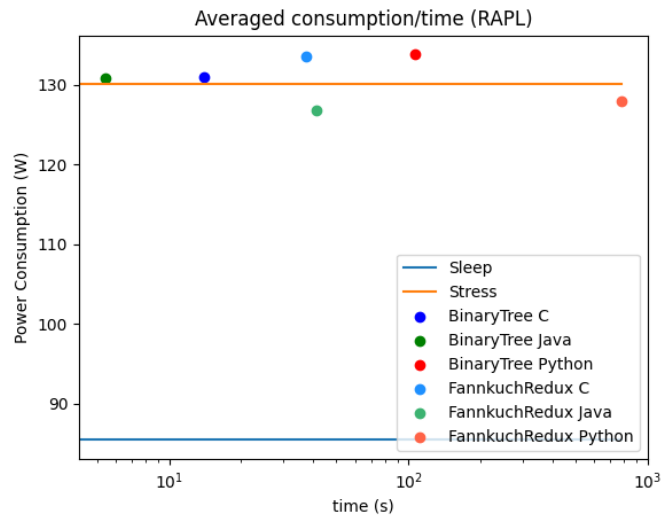

Figure 3: Averaged Power Consumption summary (RAPL)

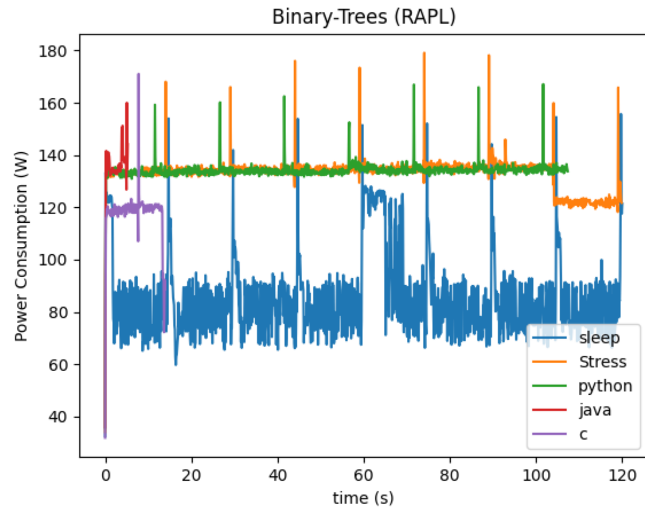

Figure 4: Power consumption for the Binary-Trees (RAPL)

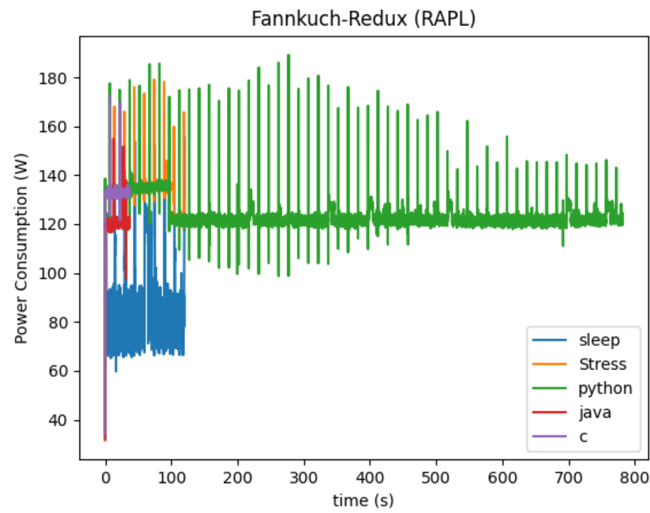

Figure 5: Power consumption for the Fannkuch-Redux (RAPL)

5.2 Kwollect Results

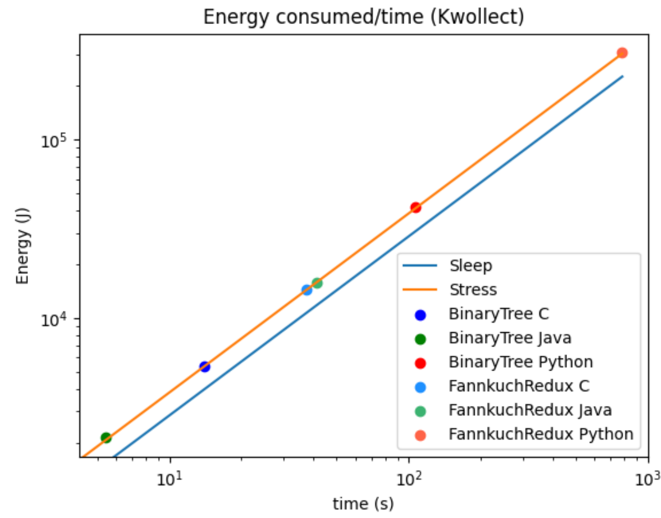

Figure 6: Energy Consumed summary (Kwollect)

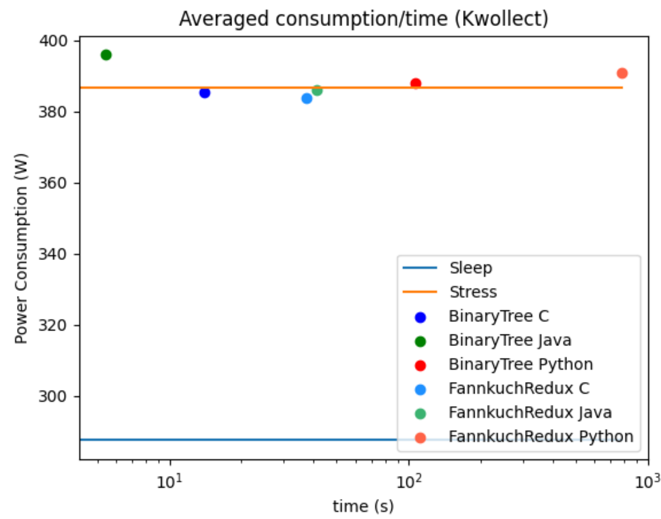

Figure 7: Averaged Power Consumption summary (Kwollect)

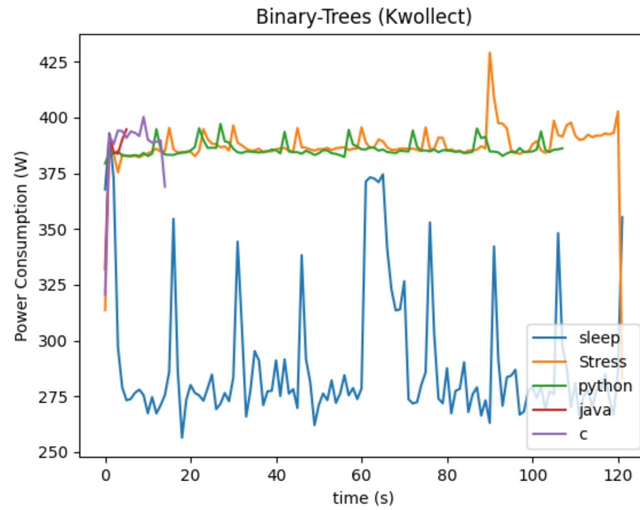

Figure 8: Power Consumption for the Binary-Trees (Kwollect)

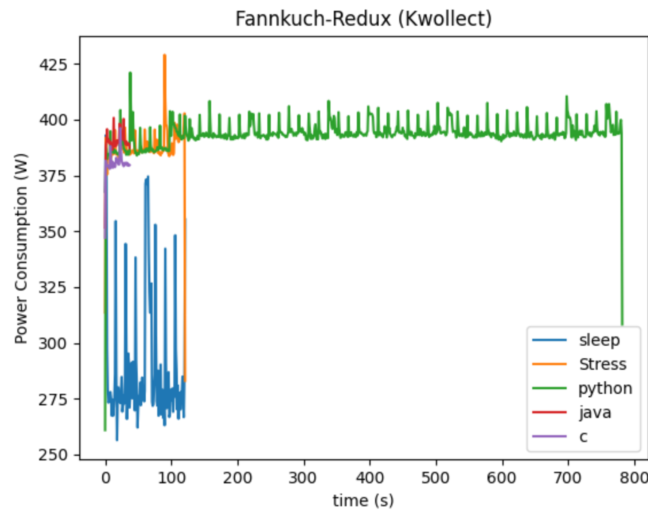

Figure 9: Power Consumption for the Fannkuch-Redux (Kwollect)

5.3 Result analysis

Figure 10 is the combination of figures 2 and 6, While figure 11 is a combination of figures 3 and 7.

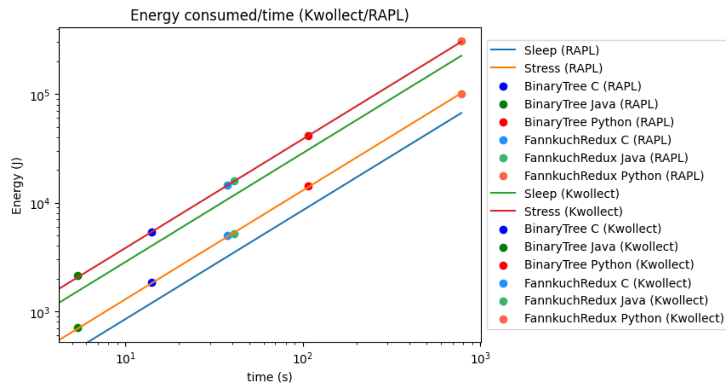

Figure 10: Energy Consumed summary (Kwollect/RAPL)

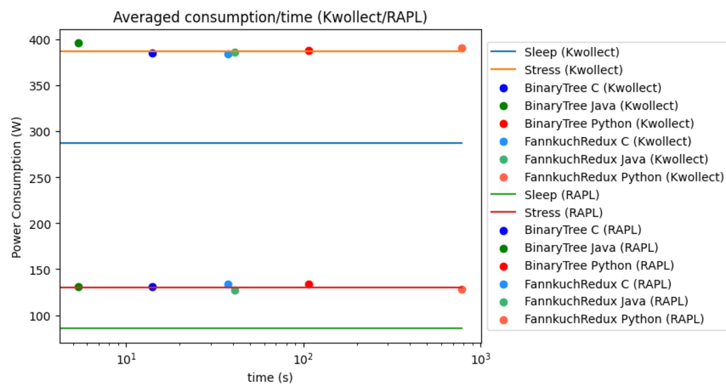

Figure 11: Averaged Power Consumption summary (Kwollect/RAPL)

The difference in values seen between RAPL and Kwollect measures (about 200% more for Kwollect) is due to the fact the Kwollect is a whole-server power-meter while RAPL is not. The Sleep benchmark help us see what the base values are when the machine is in idle, it also allows use to measure how much *more* did the other benchmarks consume, which is to say, how much they effectively consumed.

These results clearly suggest that, when running on a single processor, the language itself has no impact on the power consumption. Not only do every benchmark have the same amount of power consumption, they also have the same of power consumption as that of the Stress benchmark, which supposedly reaches 100% load on one core[14], implying that they can hardly consume any more.

[14]Command line: `stress -c 1 -t 120s -v`

	Duration	RAPL (W)	Kwollect (W)	RAPL (J)	Kwollect (J)
Binary-Trees C	13.99	131.02	385.25	1832.96	5389.65
Binary-Trees Java	5.38	130.9	395.9	704.24	2129.94
Binary-Trees Python	107.23	133.83	388.01	14350.59	41606.31
Fannkuch-Redux C	37.52	133.58	383.92	5011.92	14404.68
Fannkuch-Redux Java	41.08	126.87	385.93	5211.81	15854.00
Fannkuch-Redux Python	783.14	127.91	390.91	100171.43	306137.26
Stress	120.01	130.14	386.73	15618.10	46411.47
Sleep	120	85.43	287.47	10251.6	34496.40

Figure 12: Averaged values of RAPL and Kwollect for Binary-Trees and Fannkuch-Redux Algorithms.

Languages do, however, influence the time it takes for a given benchmark to complete, which results in a different total energy consumption. For example, despite Python having roughly the same Kwollect power consumption as C for Binary-Trees (see figure 12), Python took 107.23s while C only took 13.99s, and so Python consumed 41606.31J while C only consumed 5389.65J, which is almost 8 times more.

Moreover, this conclusion is reached for both RAPL and Kwollect measures, which further validates this analysis.

6 Conclusion

Comparing the energy efficiency of programming languages is a challenging topic as it involves the writing of multiple benchmarks in different languages, all the while having a reproducible and clear workflow measuring performance and energy of each benchmarks written in each languages. These measures must also be as fair as possible to remain valid, further increasing the challenging aspect of this topic.

The findings presented in this document imply that the main impact on the energy does not come from the instantaneous power consumption. From the comparisons, regardless of the algorithm and the language, the power consumption is quite similar to that of a classical processor stress program. Still, due to execution time being dependent on the language, and due to energy being the duration multiplied by the power, the total energy consumed is impacted by the language.

For future work, several ideas could improve the current study:

- make OCaml benchmarking possible, or, more generally, find new languages along with their implementations to bench, such as C++ and Rust, among others;
- find new algorithms to bench, preferably algorithms that involve AI or matrices;

- implement parallel versions of the existing implementations and compare them with Linux's stress command used on multiple cores;
- organise "Hackathons" to gather more idiomatic implementations of each language and algorithm;
- launch the experiments on different computer architectures to effectively compare the impact of architecture on power consumption. This could also be extended to comparing the power consumption of different Operating Systems.

Naturally, some of these ideas may require upgrading the proposed methodology to ensure that comparisons remain fair.

7 Acknowledgements

This work was supported in part by the ANR DATAZERO2 (contract "ANR-19-CE25-0016") project

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr/).

References

[1] Georges da Costa. MojitO/S, November 2021. https://hal.archives-ouvertes.fr/hal-03453537.

[2] Mathieu Fourment and Michael R. Gillings. A comparison of common programming languages used in bioinformatics. *BMC Bioinformatics*, 9(1):82, February 2008.

[3] Mathilde Jay, Vladimir Ostapenco, Laurent Lefèvre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. An experimental comparison of software-based power meters: focus on CPU and GPU. In *CCGrid 2023 - 23rd IEEE/ACM international symposium on cluster, cloud and internet computing*, pages 1–13, Bangalore, India, May 2023. IEEE.

[4] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 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*, pages 256–267, Vancouver BC Canada, October 2017. ACM.

[5] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, May 2021.

[6] Simon Portegies Zwart. The ecological impact of high-performance computing in astrophysics. *Nature Astronomy*, 4(9):819–822, September 2020.

[7] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000.