Efficient Data Processing: Assessing the Performance of Different Programming Languages

Lukas Beierlieb
University of Würzburg
Würzburg, Germany
lukas.beierlieb@uni-wuerzburg.de

André Bauer
University of Chicago
Chicago, United States
andrebauer@uchicago.edu

Robert Leppich
University of Würzburg
Würzburg, Germany
robert.leppich@uni-wuerzburg.de

Lukas Iffländer
University of Würzburg
Würzburg, Germany
lukas.ifflaender@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

This paper compares the performance of R, Python, and Rust in the context of data processing tasks. A real-world data processing task in the form of an aggregation of benchmark measurement results was implemented in each language, and their execution times were measured. The results indicate that while all languages can perform the tasks effectively, there are significant differences in performance. Even the same code showed significant runtime differences depending on the interpreter used for execution. Rust and Python were the most efficient, with R requiring much longer execution times. Additionally, the paper discusses the potential implications of these findings for data scientists and developers when choosing a language for data processing projects.

CCS CONCEPTS

• **General and reference** → **Measurement**; • **Software and its engineering** → **Software performance**; **General programming languages**.

KEYWORDS

software performance, data processing, python, R, rust

ACM Reference Format:
Lukas Beierlieb, André Bauer, Robert Leppich, Lukas Iffländer, and Samuel Kounev. 2023. Efficient Data Processing: Assessing the Performance of Different Programming Languages. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23 Companion), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3578245.3584691

1 INTRODUCTION

Data handling (reading, aggregating, etc.) is crucial in any domain. Especially, 60% time of a data science project is spent on data cleansing and organizing [1]. Consequently, inefficient processing of the data can potentially waste considerable time. This is particularly problematic when either lots of data have to be handled, where runtimes of hours and days can halt further analysis progress, or when it is desired that data streams have to be processed continuously. An illustrative example is given by the dataset provided by Traini et al. [5]; the data aggregation is a relatively simple task. However, the large file sizes slow down the process. More precisely, the raw data is 65 GB, but the actual data published for research is aggregated to 362 MB.

In this paper, we address the following questions: Given a similar implementation of the aggregation task in different suitable languages, how significant are the performance differences between them? Are there measurable differences in the runtime of the same code that is just differently compiled or interpreted? Is it possible to utilize parallelization in this particular scenario? How significant is the benefit?

To answer these questions, we used public information about the raw and processed dataset and experimentation with the actual data to understand exactly how the data is handled. Then, we choose two programming languages that are very present in the data science domain: Python and R. To contrast their interpreted nature, we also assess Rust - a low-level, compiled language mainly known for its speed and memory safety but also offers many libraries for data processing. The aspect of different interpreters is explored by using different versions of the default Python interpreter and an alternative one called PyPy. The Rust implementation is also modified to support parallel processing to reach as high performance as possible.

The rest of the paper is organized as follows: In Section 2, we describe the dataset and the utilized programming languages. In Section 3, we elaborate on our methodology. In Section 4, we describe measurement environment and discuss the results. In Section 5, we highlight related work before summarizing the paper in Section 6.

2 BACKGROUND

This section introduces the dataset's structure followed by the considered languages and the Python interpreter PyPy.

ICPE '23 Companion, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0072-9/23/04…$15.00
https://doi.org/10.1145/3578245.3584691

2.1 Dataset

The dataset utilized in this paper was recorded and provided by Traini et al. [5]. In their work, the authors investigated Java benchmarks. In the public available Zenodo repository[1], there are 600 raw files with a size of 65 GB. Fourteen files are empty or corrupt, leading to 586 files between 9 MB and 1.9 GB corresponding to the 586 investigated benchmarks. Each file was exported by the Java Microbenchmark Harness (JMH) in a JSON format, containing an array of ten measurement runs at the core. Each run includes 3000 measurement batches, while each batch has multiple measurements. Each measurement consists of a time stamp and a quantity. The processed files, which are available on GitHub[2], contain the ten runs with 3000 data points, each being the averaged value of a raw batch.

2.2 Programming Languages

Python is a high-level, general-purpose programming language that has gained widespread popularity recently, particularly for data processing. Python has a large and active community, which has led to the development of a wide range of libraries for data processing, including Pandas, NumPy, and scikit-learn, which provide a wide range of tools for data manipulation, cleaning, and analysis.

Furthermore, Python's popularity in data science has led to the development of specialized libraries and frameworks for machine learning, such as TensorFlow and PyTorch. In addition to its powerful libraries and frameworks, Python offers a wide range of visualization tools, such as Matplotlib and Seaborn.

PyPy is an alternative implementation of the standard Python interpreter (CPython), designed to be a faster and more efficient drop-in replacement. It is built using a Just-In-Time (JIT) compilation technique, which dynamically compiles Python bytecode into machine code at runtime, resulting in a significant performance boost compared to CPython while being compatible with the majority of Python code and libraries.

PyPy is particularly useful for computationally intensive applications (e.g., scientific computing, data processing, and machine learning). The PyPy JIT compiler can produce machine code faster than the equivalent code written in C or C++ and improve multithreaded code's performance. PyPy can also handle large memory footprints more efficiently than CPython, making it suitable for memory-intensive tasks.

R is a programming language and environment specifically designed for statistical computing and graphics. It is widely used among statisticians, data analysts, and data scientists for data manipulation, cleaning, and analysis. R has a rich ecosystem of libraries for data processing, including dplyr, tidyr, and ggplot2. One of the key strengths of R is its focus on data visualization. The ggplot2 library is a powerful tool for creating informative and aesthetically pleasing visualizations. Additionally, R's strong support for statistical modeling and inference makes it an excellent choice for data analysis tasks.

R has a large and active community, resulting in various packages and libraries for various data processing tasks. CRAN is a vast repository of R packages that provide additional functionalities, making it easy to perform complex data processing tasks. R's popularity in data science is also driven by its integration with other software, such as Hadoop and Spark, which efficiently process large datasets. R is also integrated with many popular data visualization software, such as Tableau and Power BI, allowing easy creation of interactive visualizations and dashboards.

Rust is a systems programming language designed for safety, speed, and concurrency. It has gained popularity in recent years, particularly in data processing, due to its emphasis on memory safety and low-level control. Rust's low-level capabilities make it well-suited for tasks that require fine-grained control over system resources, such as high-performance computing, embedded systems, and network programming.

Rust's standard library provides a wide range of features that are useful for data processing tasks, such as built-in support for parallelism and concurrency and the ability to work with raw memory. Additionally, Rust has a growing ecosystem of libraries and frameworks, such as the Rayon library, which provides data parallelism, and the Serde library, which supports serialization and deserialization of data. Rust's emphasis on safety and speed has made it an attractive option for building high-performance systems and tools for data processing, such as databases, data pipelines, and data analytics tools. Its unique combination of low-level control and memory safety makes it well-suited for tasks requiring performance and reliability.

3 APPROACH

To prevent that measurement results are biased towards one of the competing languages, two approaches can be considered. One option would be to optimize each language's implementation as much as possible. This would highlight the maximum potential of each language. However, the programmers responsible for implementation have to be experts to know how to achieve optimal performance, otherwise, there is a bias toward the better-understood languages.

Therefore, we chose the second approach: Keeping the code comparable between languages. Listing 1 shows our algorithm.

For every raw JSON file, we call `process_file()` to process the file and store the result in another JSON file in a designated folder. File processing starts with loading its content into memory, followed by letting a library parse it into a JSON data structure. The field `"scoreUnit"`, can hold the values `"s/op"`, `"ms/op"`, `"us/op"`, `"us/op"`. `"get_scale()"` returns the respective scaling factor to translate the units to seconds, i.e., `1`, `1e-3`, `1e-6`, `1e-9`. The measurement data under `"rawDataHistogram"` is then transformed such that all measurement batches are replaced with their average execution time, scaled to seconds. The minimal JSON representation (with no spaces and newlines) of the aggregated data is finally generated and written to disk.

Each implementation closely follows the pseudocode in a way that is idiomatic for the particular language. As an example, in Python, list comprehensions are used to iterate over the measurement data, R uses the `lapply` function, and Rust utilizes basic for loops. Python utilized its builtin `json` module, R the `rjson` library (as well as `purrr` to aggregate the batches), and Rust the `serde` framework for JSON. To parallelize the Rust code, `rayon`'s parallel iterator replaces the sequential loop that iterates over all files.

[1] Zenodo: https://zenodo.org/record/5961018
[2] GitHub: https://github.com/SEALABQualityGroup/icpe-data-challenge-jmh

```
fn get_scale(unit) {...}

fn process_batch(batch) {
    return sum_of_measurements(batch) /
        count_of_measurements(batch)
}

fn process_file(raw_file, processed_file) {
    raw_text = read_file(raw_file)
    raw_json = parse_json(raw_text)
    raw_data = raw_json[0]["primaryMetric"] \
        ["rawDataHistogram"]
    scale = get_scale([0]["primaryMetric"]["scoreUnit"])
    processed_json = map(raw_data, run -> {
        map(run, batch -> {
            scale * process_batch(batch)
        })
    })
    processed_text = json_to_string(processed_json)
    write_file(processed_file, processed_text)
}

fn main() {
    for file : raw_folder {
        process_file(raw_folder "/" file,
            target_folder "/" file)
    }
}
```

Listing 1: Pseudo code of processing code

We implemented scripts to build a docker image for each variant, as well as scripts to run containers of these images, and measure and store their execution times. The code is published on GitHub[3]. The performed measurements are presented in the next section.

4 EVALUATION

Executing multiple iterations for each variant for the whole 65 GB, 586 file dataset is very time-consuming, so we chose to split the evaluation into two parts. In the first part, the single-file measurements, we executed all 13 variants in succession on just the largest file (1.9 GB) of the raw dataset. Without breaks in between, this is repeated for 10 iterations. In the second part, we measure the runtime for each variant for the whole dataset, but only once, in order to get an estimate for the average runtime without requiring tens of hours of measurements.

Section 4.1 gives details about the test environment before the single-file results are presented in 4.2 and the dataset results in 4.3.

4.1 Hard- and Software

The relevant details about the hardware used to run the measurements are listed in Table 1.

The utilized software and corresponding version numbers are listed in Table 2.

[3]GitHub: https://github.com/lbeierlieb/icpe23data_challenge

Table 1: Hardware specifications used for all measurements

Category	Specification
Device	Lenovo ThinkPad P1 (2. Gen)
Processor	Intel Core i7-9850H
RAM	32GB DDR4 2666MHz
Storage	Samsung 970 EVO NVMe SSD
SSD max seq. read	3500 MB/s
SSD max seq. write	3300 MB/s

Table 2: Software used for all measurements

Software	Version
Arch Linux	-
Kernel	6.1.6-arch1-3
Docker	20.10.22
Python	3.7, 3.8, 3.9, 3.10, 3.11
PyPy	3.7, 3.8, 3.9
R	4.22
rjson	0.2.21
Rust	1.66
serde	1.0.152
rayon	1.6.1

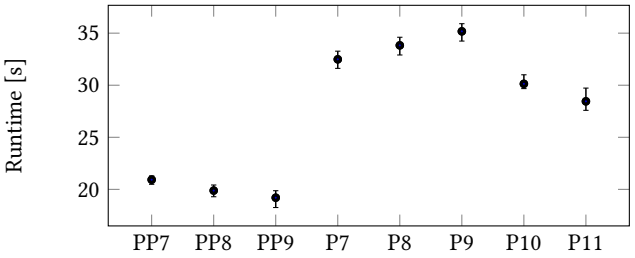

Figure 1: Python variants' runtimes for a 1.9 GB file

4.2 Single-File Measurements

Figure 1 shows the runtimes (y-axis) of the same Python code for a single 1.9 GB JSON file for 8 different Python interpreters, which are listed on the x-axis. PP stands for PyPy, P for default Python, and the number for the minor version, e.g., PP7 is PyPy 3.7. The thick dot represents the average of the 10 measured runtimes, and the bars above and below indicate the minimum and maximum execution time. The results show that PyPy significantly outperforms Python, with PyPy 3.9 at 19.20s and Python 3.11 at 28.45s. PyPy received slight performance boosts with newer versions, while Python 3.10 and 3.11 are great improvements over their previous versions.

Figure 2 presents a comparison between single-threaded Rust, the fastest version of PyPy and Python, and R. R is dramatically slower, requiring on average 280.73s, which is ten times longer than what Python 3.11 needs. Due to the scale, the difference between Rust and Pypy/Python is not well visible. Though, as to be expected, Rust is faster with a mean runtime of 8.40s.

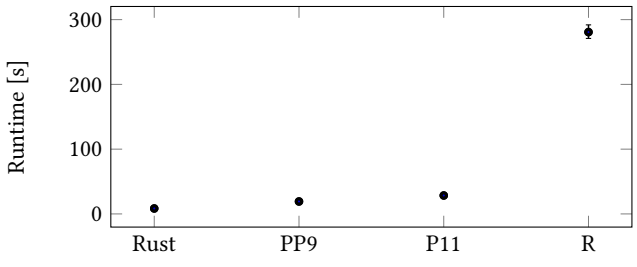

Figure 2: Different variants' runtimes for a 1.9 GB file

4.3 Dataset Measurements

The single measured execution times for processing the whole dataset are listed in Table 3. The durations correspond generally well with the single-file runtimes. However, the real-world impact is probably better recognizable. Waiting up to 20mins before being able to analyze 65 GB worth of data seems more reasonable than the 3-hour stall with R. In the multi-file dataset measurement, there are also results for parallelized Rust variants. The captured runtimes show that speed-up is significant but not linear with thread count, probably due to IO limitations.

4.4 Threats to Validity and Discussion

As our results were only produced on one dataset and only on one hardware setting, the results are surely not representative of every scenario. With more time available, more iterations could be executed to improve confidence in the consistency of the results. However, the 10 performed repetitions on the single file showed sufficient repeatability to recognize significant differences between variants. Thus, we also believe the single dataset measurements give a fairly representative runtime. The code and scripts from the GitHub repository can be used to replicate the results on similar hardware or assess the situation on different machines.

We want to note the following aspects we discovered during working on this paper. There can be huge differences between semantically identical implementations—we have seen 1:45 min to 3 h between parallel Rust and R. It is advisable to consider such aspects when choosing a technology stack for larger amounts of data. Writing Rust code requires more effort due to aspects like type declarations and data ownership, but in return, proper error handling comes more naturally with well integrated `Result` and `Option` types. Python and R make it simple to quickly get to a working program, and in such cases as transforming one dataset into another form, this might be preferable. Python users are advised to try to run their code with PyPy as there can be a significant performance boost with zero changes to the codebase.

Regarding parallelization, we noticed that libraries like `rayon` make it simple to gain performance when data parallelism is possible. In the dataset aggregation scenario considered in this paper, caution is required, though. Reading and parsing large JSON files needs a considerable amount of memory, and multiple threads simultaneously working on large files can overload the system. We had to limit the thread count to four because 32 GB of system memory did not suffice for more when working on the whole dataset.

Table 3: Runtimes for the whole dataset

Variant	Runtime [HH:mm:ss.SS]
pypy3_7	00:11:55.14
pypy3_8	00:11:13.73
pypy3_9	00:10:32.92
python3_7	00:18:55.79
python3_8	00:19:10.60
python3_9	00:19:54.83
python3_10	00:17:02.74
python3_11	00:16:00.72
r_rjson	03:00:56.25
rust_serde	00:04:30.78
rust_serde_rayon_2thread	00:02:48.65
rust_serde_rayon_3thread	00:02:04.94
rust_serde_rayon_4thread	00:01:45.59

5 RELATED WORK

Performance engineering is a wide field with lots of research. One of the domains within this field is the assessment of the performance impact of different programming languages. For instance, in their work [2], the authors compare the influence of the programming language on CPU performance. To this end, the authors wrote a database application in the languages C#, PHP, JAVA, JSP, and ASP.Net and compared their performance. In a similar work [3], the authors investigated ten programming problems while using 27 programming languages. To measure the performance, the monitor the memory usage, runtime, and energy consumption. In another paper [4], the authors compared different sorting algorithms in Python and C regarding their energy efficiency. To access the performance and state guidelines when which algorithm in which language should be used, the authors varied the input size and the underlying hardware. In contrast to our work, these works focus only on different programming languages and not also on the impact of different versions of a programming language, as we do with Python.

6 CONCLUSION

Data handling is a crucial task in any domain. However, the large file sizes can slow down the process, especially for continuous data streams that need to be processed in a timely manner. This study investigates the effect of using different programming languages (Python, R, and Rust) and versions on data aggregation utilizing the dataset from Traini et al.[5]. The results show that there are significant differences among them, with a quick Rust, followed by Python and a significantly slower R.

REFERENCES

[1] 2016. *Data Science Report 2016.* Technical Report. CrowdFlower.

[2] Md Ahsan Arif, Mohammad Shahazzat Hossain, Nazmun Nahar, and Mst Dilruba Khatun. 2014. An Empirical Analysis of C#, PHP, JAVA, JSP and ASP. Net regarding performance analysis based on CPU utilization. *Banglavision Research Journal* 14, 1 (2014), 173–187.

[3] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João de Sousa Saraiva. 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* (2017).

[4] Norbert Schmitt, Supriya Kamthania, Nishant Rawtani, Luis Mendoza, Klaus-Dieter Lange, and Samuel Kounev. 2021. Energy-Efficiency Comparison of Common Sorting Algorithms. *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2021), 1–8.

[5] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2023. Towards effective assessment of steady state performance in Java software: are we there yet? *Empirical Software Engineering* 28, 1 (2023), 1–57.