Comparing the Performance of Data Processing Implementations

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

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

Thomas Prantl
thomas.prantl@uni-wuerzburg.de
University of Würzburg

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

Abstract

This paper compares the execution speed of R, Python, and Rust implementations in the context of data processing. A real-world data processing task in the form of an aggregation of benchmark measurement results was implemented in each language, and the execution times were measured. Rust and Python showed significantly superior performance compared to the R implementation. Further, we compared the results of different Python interpreters (the most recent versions of CPython and PyPy), also resulting in measurable variations. Finally, a study of the effectiveness of multithreading was performed.

Keywords— software performance, data processing, python, R language, rust

1 Introduction

Data handling (reading, aggregating, etc.) is crucial in any domain. Inefficient processing of the data can potentially waste considerable time. This is particularly problematic when either lots of data have to be handled, 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. [4]; the data aggregation is a relatively simple task. However, the large file sizes slow down the process. The raw data is 65 GB, but the 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 there a benefit to utilize parallelization in this scenario?

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.

A comparison between the already mentioned versions has already been published [3]. There were strong suggestions to compare a more idiomatic Rust implementation. This paper presents the previous results together with the improved Rust variant.

There are related works that assess performance impact of different programming languages. For instance, in their work [1], 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 another paper [2], the authors compared different sorting algorithms in Python and C regarding their energy efficiency.

The rest of the paper is organized as follows: In Section 2, we describe the dataset and the utilized programming languages. Section 3, presents our methodoly; Section 4 the measurement environment and results. Finally, we summarize the paper in Section 5.

2 Background

This section introduces the dataset's structure followed by the considered languages.

Dataset The dataset utilized in this paper was recorded and provided by Traini et al. [4]. 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.

Programming Languages Python is a high-level, general-purpose programming language that has gained widespread popularity, particularly for data processing. A wide range of libraries for data processing exist, including Pandas, NumPy, and scikit-learn, which provide tools

[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, out_file) {
    raw_json = parse_json(read_file(raw_file))
    raw_data = raw_json[0] \
        ["primaryMetric"]["rawDataHistogram"]
    scale = get_scale([0] \
        ["primaryMetric"]["scoreUnit"])
    out_json = map(raw_data, run -> {
        map(run, batch -> {
            scale * process_batch(batch)
    })})
    out_text = json_to_string(out_json)
    write_file(out_file, out_text)
}
for (file : raw_folder)
    process_file(raw_folder "/" file, \
        target_folder "/" file)
```

Listing 1: Pseudo code of processing code [3]

for data manipulation, cleaning, and analysis. 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, resulting in a significant performance boost compared to CPython while being compatible with the majority of Python code and libraries.

R is a programming language and environment specifically designed for statistical computing and graphics and widely used among statisticians, data analysts, and data scientists. R has a rich ecosystem of libraries for data processing, e.g., dplyr and tidyr.

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 while also providing high-level code abstractions.

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.

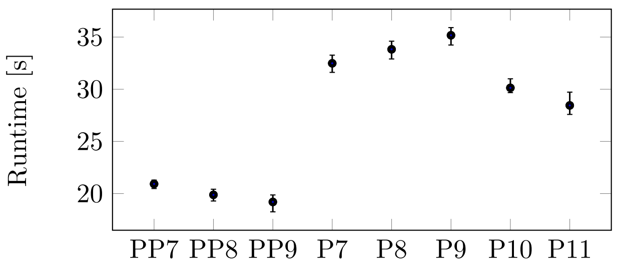

Figure 1: Python variants' runtimes for 1.9 GB file [3]

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 loop that iterates over all files.

Python and R as loosely typed languages are able to use JSON-parsed values without type casting or error checking. Rust is strongly typed and requires explicit handling of all errors, so there are a lot of cast and checks in the code, which hurts both readability and runtime speed. Feedback to [3] suggested to use strongly typed JSON parsing of `serde` to parse the JSON files into predefined data structures, allowing type check-free usage afterwards. We will refer to this as the improved Rust implementation.

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

The evaluation is split into two parts. Presented in Section 4.1, 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. Then, in Section 4.2, 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 4.3 discusses the findings. The details about hardware and software versions have been omitted in this paper, but can be found in [3], as they have not changed.

4.1 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 ex-

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

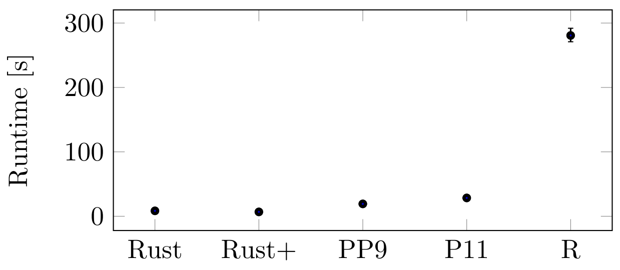

Figure 2: Different variants' runtimes for a 1.9 GB file

ecution 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 for the original version, and - yet notably faster - 6.75s for the improved.

4.2 Dataset Measurements

The single measured execution times for processing the whole dataset are listed in Table 1. 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.

4.3 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 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 similar implementations—1:45 min to 3 h. 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. 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 without code changes. Using strongly typed parsing in Rust required some upfront effort to define the data structures to parse to, but significantly improved the quality of the code and could decrease runtime by 18%. 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.

Table 1: 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_improved	00:04:04.52
rust_serde_2thread	00:02:48.65
rust_serde_3thread	00:02:04.94
rust_serde_4thread	00:01:45.59

5 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.[4]. The results show that there are significant differences among them, with a quick Rust, followed by Python and a significantly slower R.

References

[1] M. A. Arif et al. “An Empirical Analysis of C#, PHP, JAVA, JSP and ASP. Net regarding performance analysis based on CPU utilization”. In: *Banglavision Research Journal* 14.1 (2014).

[2] N. Schmitt et al. “Energy-Efficiency Comparison of Common Sorting Algorithms”. In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2021).

[3] L. Beierlieb et al. “Efficient Data Processing: Assessing the Performance of Different Programming Languages”. In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering.* 2023.

[4] L. Traini et al. “Towards effective assessment of steady state performance in Java software: are we there yet?” In: *Empirical Software Engineering* 28.1 (2023).