Empirical Investigation of Data-Parallel Concurrency in Rust using Rayon

1. Introduction

- Rust has become a popular choice as systems language
- Compile time ownership and borrowing enable concurrency
- Data parallelism is performing an operation on multiple pieces of data at once
- Rayon is a Rust library that provides simple data parallelism

2. Research Question

How is Rayon used in open-source Rust projects to realize data-parallel concurrency?

RQ1 Most common usages
RQ2 Non-trivial workloads
RQ3 Usage with unsafe Rust
RQ4 Limitations of parallel iterators

3.1 Methodology

Data selection

- Top 1000 Rust GitHub repositories were considered
- Filter to only consider repositories referencing Rayon → 172 results, 154 results using the identified primitives
- Purposive sampling for analysis according to criteria:
  a. Stargazer count
  b. Unique Rayon features used
  c. Usage of lower level Rayon primitives
  d. Usage of unsafe in files using Rayon
- Rayon primitives for search derived from Rayon documentation, iteratively changed to minimize false positives

3.2 Methodology

Data Processing

- The top 5 repositories ranked according to the above criteria will be analyzed.
- From each selected repository, 5 random instances of every occurring primitive are considered.
- For each repository, continue to next primitive on three consecutive instances of a primitive without new information.

Manual Analysis

- Form for each repository
  - Repository domain (e.g. data processing)
  - Size (LoC)
  - Description Rayon purpose
- Survey for each instance
  - Textual description of snippet
  - True/False relevance to sub-questions
  - Textual analysis of Rayon usage in relation to relevant sub-questions

Thematic Analysis

- Three stages:
  - Generation of initial codes
  - Group codes to identify themes
  - Ensure themes are internally coherent and distinct

4. Results - Frequency Analysis

Table 1: Number of unique repositories containing primitives

Primitive	Count
.par_iter	114
.into_par_iter	98
rayon::ThreadPoolBuilder	73
.par_bridge	39
rayon::spawn	31
rayon::scope	25
rayon::join	20

Table 2: Primitive usage counts

Primitive	Count
.par_iter	1562
.into_par_iter	1253
rayon::ThreadPoolBuilder	245
.par_chunks	195
.par_bridge	119
rayon::spawn	94
rayon::join	91
rayon::scope	71

Table 3: Top repositories by normalized even weight ranking

Repository	Unsafe	Primitives	Low level	Stars	Score
pola-rs/polars	390	10	Yes	38364	0.760
astral-sh/uv	36	4	Yes	84298	0.508
zed-industries/zed	154	3	Yes	81450	0.487
ruvnet/RuVector	120	7	Yes	3886	0.478
typst/typst	0	6	Yes	53263	0.469

5. Results - Thematic Analysis

Ordering constraints - RQ2, RQ4

- Non-trivial workload that cannot always be expressed using parallel iterators
- Presence of ordering constraints can cause programmers to reach for lower level primitives

Nested iteration - RQ2, RQ4

- Non-trivial workload, where parallel iterators can be used to express concurrency, but introduces unit of parallelism considerations

Custom parallel iterator implementation - RQ2, RQ4

- Non-trivial workloads may be handled by parallel iterators through proprietary implementations of the interface
- Encapsulates concurrency in idiomatic parallel iterators

Unsafe for performance - RQ3

- Usage of an unsafe block to realize a gain in runtime or memory efficiency
- Allows avoidance of bounds checking and local allocation costs

Unsafe for method call - RQ3

- Partially explains frequent co-occurrence between unsafe and Rayon usage
- Not directly related to concurrency

6. Conclusions

- Parallel iterators are observed as the most common Rayon usage.
- Non-trivial workloads may require developers to use lower level constructs, extending parallel iterators provides an alternative.
- Unsafe Rust may be used to realize performance, though unsafe usage is dominated by non-concurrency related tasks

7. Limitations

- Repository selection was limited to GitHub.
- Purposive sampling may have biased thematic analysis results.
- Analysis was performed by a single researcher, and did not include peer review.