Understanding Shared-Memory Concurrency Usage in Rust Systems

RQ) How is shared-memory concurrency used in Rust systems?

- **SQ1) Which shared-memory concurrency primitives are used?**
- **SQ2) What thread organisation models are used?**
- **SQ3) What workload types are present?**
- **SQ4) What functional roles are associated with shared-memory concurrency?**
- **SQ5) What trade-offs exist between implementations?**

1) Introduction:

- Shared-memory concurrency lets **multiple threads coordinate** through shared state.
- Rust provides **memory-safe** concurrency through **ownership** and **synchronisation** primitives.
- Little is known about how shared-memory concurrency is **used in real-world Rust systems**.
- As more concurrent software is **developed or migrated to Rust**, understanding these implementation patterns becomes increasingly important.

2) Dataset:

- Popular **open-source Rust application** repositories selected from **Awesome Rust.**
- Filtered for **substantial shared-memory** concurrency usage.
- Final dataset: **12 repositories**, **59 files**.

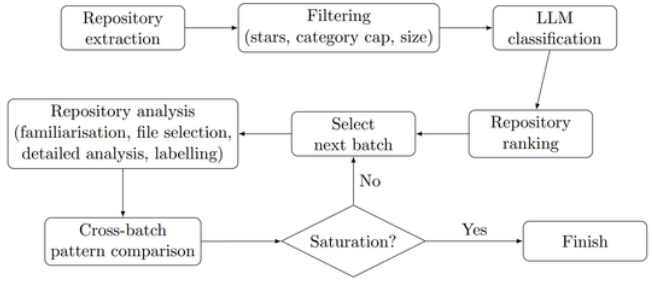

Figure 1: Overview of the repository selection and iterative qualitative analysis workflow.

3) Analysis:

- **Manual qualitative analysis** of selected files.
- Files were labelled by **concurrency primitives, thread organisation model, workload type,** and **functional role**.
- Cross-batch pattern **comparison** until **saturation.**

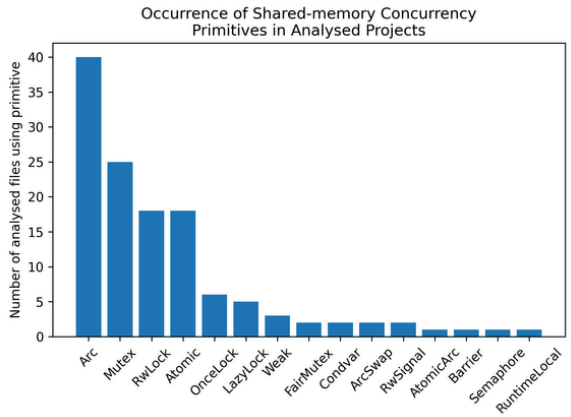

Figure 2: Frequency distribution of shared-memory concurrency primitives across the analysed files. Arc, Mutex, RwLock and Atomic dominate.

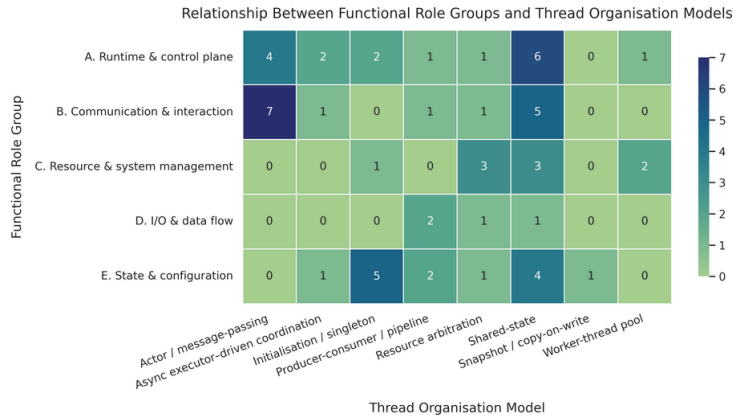

Figure 3: Different thread organisation models support different functional roles. Actor systems emphasise communication, while shared-state systems emphasise runtime and control plane.

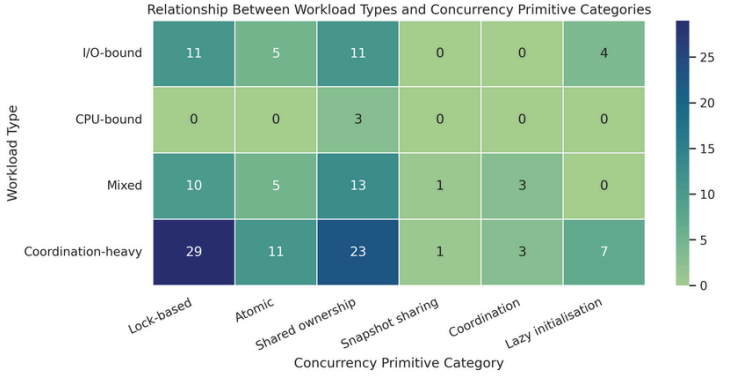

Figure 4: Shows how workload types relate to concurrency primitive usage. Coordination-heavy workloads account for most shared-memory concurrency usage and rely heavily on shared ownership and locking primitives.

4) Findings

- *SQ1)* **Arc, Mutex**, **RwLock,** and **Atomic** are the **most** used primitives.
- Primitive usage follows a **Zipf-like distribution.**
- *SQ2)* **Shared-state concurrency** dominates, followed by **actor/message-passing** models.
- *SQ3)* Concurrency usage is primarily associated with **coordination-heavy workloads** (not computation).
- I/O-bound and mixed workloads show the same reliance on **locking** and **shared ownership** primitives.
- **CPU-bound** workloads are **rarely observed .**
- *SQ4)* Functional roles are dominated by **runtime**, **resource management**, and **communication**.
- Role-thread organisation models **patterns**:
  - Actor systems: communication and interaction.
  - Shared-state systems: runtime and control plane.
  - Initialisation systems: state and configuration.
- *SQ5)* Trade-offs:
  - Shared-state: **simple** and **flexible**, but prone to **contention** and synchronisation **overhead**.
  - Actor/message-passing: **reduces shared state**, but still requires **coordination.**
  - Atomics: **efficient** but **harder to reason** about due to memory ordering.

5) Conclusions

Shared-memory concurrency in Rust is dominated by a **small set of primitives** and is primarily used for system **coordination** rather than computation.

6) Future work:

- Extend the dataset to more **repositories** and application **domains**
- Compare shared-memory concurrency with **different concurrency practices**
- **Quantitatively** evaluate performance, scalability, and maintainability **trade-offs**
- Investigate concurrency design evolution in **larger production systems**.

LOTTE TRIJBELS — A.J.T.P.TRIJBELS@STUDENT.TUDELFT.NL