Brief Announcement: Is the Problem-Based Benchmark Suite Fearless with Rust?

Javad Abdi
javad.abdi@mail.utoronto.ca
University of Toronto

Guowei Zhang
zhangguowei9@hisilicon.com
Huawei

Mark C. Jeffrey
mcj@ece.utoronto.ca
University of Toronto

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; *Parallel algorithms.*

KEYWORDS

Rust, fearless concurrency, regular parallelism, irregular parallelism

ACM Reference Format:
Javad Abdi, Guowei Zhang, and Mark C. Jeffrey. 2023. Brief Announcement: Is the Problem-Based Benchmark Suite Fearless with Rust?. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3558481.3591313

1 INTRODUCTION

> **fear** n. **1. a.** an unpleasant emotion caused by anticipation or awareness of danger —adapted from Merriam-Webster

Commodity architectures are now parallel by default, yet apart from exceptional cases, the notorious challenges of parallel programming endure. On one hand, a few application domains have sustained a performance boom since the shift to multicores, in part due to their abundant obvious sources of parallelism. On the other hand, parallel algorithm and implementation experts have uncovered surprising opportunities for task-level parallelism in conventionally challenging domains [4, 12, 13, 17, 18, 30, 33]. Algorithms in the former domains typically have abundant *regular* parallelism, where data and control dependences among tasks are statically identifiable, while algorithms in the latter are challenged with *irregular* parallelism, with dynamically manifesting data and control dependences [30]. Scheduling tasks and synchronizing irregular data accesses continues to challenge programmers with pitfalls such as non-determinism [26], deadlock, data races, and other concurrency bugs [15, 27, 36]. While a plethora of work in programming languages [5, 10], language extensions [6–8, 32], and type systems [16, 28] has sought to curtail concurrency bugs, few have reached mainstream adoption.

Rust is gaining traction as a systems programming language for building reliable and efficient applications [25]. It has been the most loved programming language on the Stack Overflow Developer Survey for seven consecutive years, and has been adopted into major open-source and commercial software [3, 14, 19–22, 29]. Rust brings together higher-level *safety* and lower-level resource control by leveraging its type system, built atop prior work on *regions* [34], to capture memory and concurrency bugs at compile time. The golden rule of the Rust type system is that aliasing comes at the cost of immutability: at any point in the program, every value has either one mutable or possibly several immutable references to it, i.e., *aliasing XOR mutability* [23] or `AXM` for short [35]. These restrictions enable Rust to statically provide memory safety without garbage collection and rule out data races. In fact, the Rust book introduces concurrency and parallelism features with a chapter entitled "Fearless Concurrency" [25, Chapter 16].[1]

Unfortunately, Rust's `AXM` restriction prevents instances of parallelism where tasks must mutate aliased state, so the language offers flexibility through **`unsafe`** code blocks. Best practice suggests that Rust programmers minimize their use of unsafe code and encapsulate it within safe APIs with run-time checks (*interior unsafe*) [2, 24, 25]. Prior work has considered the interaction between safe and unsafe Rust code, and we focus on concurrency and parallelism. RustBelt [23] and RustBelt Relaxed [11] prove the soundness of the Rust type system and provide tools for programmers to verify encapsulation of unsafe code. Qrates [2] analyzes how **`unsafe`** is used across 34,000+ Rust projects on crates.io, finding that unsafe concurrency blocks exist, but are rare. Qin et al. uncover concurrency and memory safety bugs in large-scale open-source systems software [31]. While prior work investigated Rust support in conventional multithreading contexts, as far as we are aware, Rust's purported fearless concurrency has yet to be studied *through the lens of regular vs. irregular bulk-synchronous parallelism.*

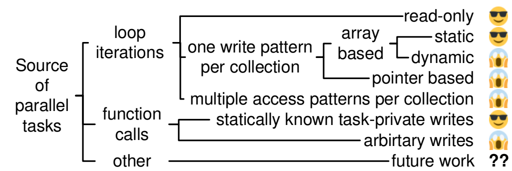

Figure 1: Analysis of Rust support per parallel pattern.
Emojis: Twemoji by Twitter, Inc and other contributors, licensed under CC-BY 4.0.

This brief announcement makes the following key contributions:

- A definition of fearless concurrency.
- A case study of Rust's support for concurrency considering regular vs. irregular parallelism, summarized in Fig. 1.
- A Rust benchmark suite of fine-grain regular and irregular parallel applications that can serve as a launchpad for future programming language, compiler, and runtime research.

We find that Rust makes programmers fearless when expressing patterns with statically known write sets. However, they still face significant challenges when expressing irregular parallelism.

[1]Klabnik et al. use "concurrency" as a stand-in for both concurrency and parallelism.

SPAA '23, June 17–19, 2023, Orlando, FL, USA

2 WHAT IS FEARLESS CONCURRENCY?

The anticipated danger that inspires fear in parallel programmers is the potential for concurrency errors that manifest at run time. Fearless concurrency is the Rust Team's nickname for the their goal that "[...] you can fix your code while you're working on it rather than potentially after it has been shipped to production" [25]. This nickname warrants analysis.

At one extreme, Rust will rule out all mixing of aliasing and mutability at compile time for any program devoid of **`unsafe`** blocks, including its libraries. For such a program, any data race is caught at compile time and deadlock is impossible (mutexes require unsafe Rust). However, data-race freedom does not imply freedom from atomicity violations [15] nor from order violations [27].

At the other extreme, Rust can rule out data races for programs requiring lock-based or lock-free synchronization [9]. However, the risk of atomicity and order violations remains, and the programmer must choose between the poor scaling of coarse-grain locks or the fear of deadlock and livelock.

Between these extremes are programs requiring barrier synchronization, with some interior unsafe APIs. Such functions should use dynamic checks to validate their contracts. In these cases, `rustc` could not catch all errors at compile time, so when validation fails, the error manifests at run time. Encapsulated dynamic checks move an error's symptom closer to the cause, but crashes in production remain possible, leaving fear with some hope for a clear postmortem.

Taken together, we find that fearless concurrency would be better interpreted as a spectrum: ideally eliminating any fear of data races[2] or deadlocks at compile time, and otherwise keeping run-time error symptoms close to their causes.

3 OUR CASE STUDY

We study how experts have expressed parallelism in their software, specifically turning to the Problem-Based Benchmark Suite (PBBS) [1] due to a lack of Rust benchmarks with irregular parallelism. PBBS comprises efficient C/C++ implementations of algorithms for a diverse set of problems, and importantly uses regular and irregular parallelism. We port 12 benchmarks to Rust and categorize the parallel patterns based on their writes to shared data. We assess the programmer's (our) fearlessness in expressing each pattern category. Table 1 summarizes the observed patterns and the programmer sentiments.

We use Rayon for runtime scheduling and for parallel operations such as `map`. Rayon is a Rust work-stealing-based data-parallelism library influenced by Cilk [6]. This makes it the right tool to express the types of parallelism found in PBBS. Moreover, Rayon is the de facto way to express parallelism in many major Rust projects [22].

We organize our case study from straightforward to more difficult types of parallelism. Irregular writes or the combination of irregular reads and regular writes to shared data preclude Rust's fearless features, necessitating the conventional synchronization that has scared programmers for decades.

Regular parallelism with Rust: When the set of tasks and all their data dependences are statically known or parameterized, this *regular parallelism* is feasible to validate at compile time and eliminates most run-time overheads of parallel scheduling.

At the simplest extreme, tasks that only read shared collections (`RO`) are trivial to check: *aliasing XOR 0* allows aliasing. Rust indisputably keeps read-only parallelism fearless by tracking reference mutability to detect any errors (unintended writes to shared data). Yet, immutability is sufficient but not necessary for parallelism.

Writes to shared data ultimately cause the dependences that constrain parallelism. When writes are statically analyzable by `rustc`, Rust enables fearless parallel expression among independent tasks. *Destructuring* allows `rustc` to track references at fine granularity, down to the individual element, for statically sized data structures like arrays. Since destructing rules out aliasing, then *0 XOR mutability* ideally permits task-private writes. However, destructuring does not support dynamically sized data structures and inhibits many parallel patterns—it is cumbersome for this purpose. `rustc` tracks references of dynamically sized structures (e.g., vectors) at the coarse granularity of the whole collection, making inter-task aliasing difficult to avoid. Rayon takes a different approach for patterns with non-overlapping writes such as `Stride`, `Block`, and `Fork`. Rayon uses interior-unsafe functions whose interfaces statically constrain which element(s) of a collection a task can mutate by passing element references as arguments to the task. These functions mutably borrow the full collection *before* launching tasks. Together, Rayon and `rustc` uphold `AXM` by preventing tasks from arbitrarily indexing into the collection.

Irregular parallelism with Rust: When the set of tasks or their data dependences are unknown at compile time, correctness of this *irregular parallelism* must be validated or enforced at run time. Programmer fear is only mitigated through expensive run-time checks, if at all, challenging Rust's claim of fearlessness.

Rust provides limited support when algorithm-specific properties guarantee task independence, but exact write locations are statically unknown. For example, the programmer can safely elide synchronization for `SngInd` and `RngInd` when all `offsets` are unique or increasing, respectively. Unfortunately, Rust puts the programmer in a predicament: they can *(i)* validate the offsets requirement with an expensive dynamic check within a new interior-unsafe function; *(ii)* bear the unnecessary fear and performance hit of conventional synchronization (Sec. 2); or *(iii)* maximize performance but forgo Rust's safety support with unsafe unchecked code. Fearlessness comes with a cost for these patterns.

Rust does not eliminate fear when tasks have irregular dependences. We so far considered tasks with independent read and write sets per phase. However, tasks can have occasional overlapping read and write sets (`AW`) in parallel applications spanning domains such as graph analytics, geometry, statistical inference, and optimization, among others [30]. The programmer strives to maximize parallelism while enforcing correct memory access interleavings through run-time mechanisms like synchronization. Through reference tracking, `rustc` will rule out data races, but placating the compiler will either sacrifice parallelism with coarse-grain locking, or risk deadlock, livelock, and other concurrency errors.

4 RUSTY-PBBS

Rusty-PBBS[3] is our Rust benchmark suite for bulk-synchronous regular and irregular parallelism. Table 2 lists the 12 benchmarks we

[2] A new term is warranted, given the lack of coverage for atomicity and order violations.

[3] https://github.com/mcj-group/rusty-pbbs

Table 1: Studied parallel access patterns and their fearlessness.

Abbr.	Write pattern	Task i writes to	Fearlessness
`RO`	Read only (`AXM`)	N/A	😎
`Stride`	Striding	`Array[i]`	😎
`Block`	Blocking	`Array[i*chunk_size..(i+1)*chunk_size]`	😎
`Fork`	Fork-join	a non-overlapping subset of `Array`	😎
`SngInd`	Single-valued indirection	`Array[offsets[i]]`	😱
`RngInd`	Ranged indirection	`Array[offsets[i]..offsets[i+1]]`	😃
`AW`	Arbitrary writes	pointers or random indices	😱

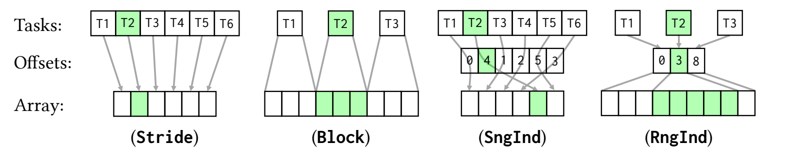

Table 2: Ported benchmarks and their patterns.

Benchmark	Regular: `RO`	Regular: `Stride`	Regular: `Block`	Regular: `Fork`	Irregular: `SngInd`	Irregular: `RngInd`	Irregular: `AW`
bwd	✔	✔			✔	✔	✔
dedup	✔	✔					✔
dr	✔	✔			✔	✔	✔
hist	✔	✔				✔	✔
isort	✔					✔	✔
lrs	✔	✔	✔		✔	✔	
mis	✔		✔		✔	✔	
mm	✔		✔		✔	✔	
msf	✔		✔	✔	✔	✔	
sa	✔	✔		✔	✔	✔	
sf	✔		✔		✔	✔	
sort	✔		✔	✔		✔	

have faithfully ported from C/C++ to Rust (PBBS has 22 total). Each benchmark uses the checked patterns following its name, either directly or through its building blocks. PBBS provides many algorithms for `sort`, but we only implemented sample sort because it makes use of the other algorithms under the hood. Rusty-PBBS has switches to replace safe implementations of patterns with unsafe (but sometimes faster) variants, such as for `SngInd` and `RngInd`.

Our hope is that Rusty-PBBS lowers the barrier for future work on parallelism in Rust. Compiler research could characterize the effect of Rust's high-level restrictions on optimizations. Runtime research could compare and augment work-stealing techniques among Rayon, Cilk, and OpenMP. Programming language user studies could compare the ease of parallel programming in Rust vs. C/C++. Future case studies should complete the port of PBBS and further interrogate Rust's support for other parallel patterns such as pipelines, static/dynamic dependence graphs [18], and priority scheduling.

ACKNOWLEDGMENTS

We sincerely thank Isidor R. Brkić, Leo X. Han, Gilead Posluns, Guozheng (Ray) Zhang, and the anonymous reviewers for their helpful feedback. We thank Boxuan (Dave) Wang for discussions on Rust code. This work was supported in part by the Digital Research Alliance of Canada, the University of Toronto, and NSERC.

REFERENCES

[1] Daniel Anderson, Guy E Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *Proc. PPoPP.*
[2] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA (2020).
[3] Jeff Barr. 2018. Firecracker - Lightweight Virtualization for Serverless Computing. AWS News Blog.
[4] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. PPoPP.*
[5] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. In *Proc. PPoPP.*
[6] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proc. PPoPP.*
[7] OpenMP Arch. Review Board. 2013. OpenMP Application Program Interface.
[8] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proc. OOPSLA.* 97–116.
[9] Mara Bos. 2023. *Rust Atomics and Locks.* O'Reilly Media, Inc.
[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. OOPSLA-20.*
[11] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2019).
[12] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proc. SPAA.*
[13] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *TOPC* 8, 1 (2021).
[14] The Dropbox Capture Team. 2021. Why we built a custom Rust library for Capture. Dropbox.tech.
[15] Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. In *Proc. PLDI.* 338–349.
[16] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proc. OOPSLA.* 21–40.
[17] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proc. SPAA.*
[18] Muhammad Amber Hassaan, Donald Nguyen, and Keshav Pingali. 2015. Kinetic Dependence Graphs. In *Proc. ASPLOS-XX.*
[19] Dave Herman. 2016. Shipping Rust in Firefox. Mozilla Hacks.
[20] Jesse Howarth. 2020. Why Discord is switching from Go to Rust. Discord Blog.
[21] Dana Jansens. 2023. Supporting the Use of Rust in the Chromium Project. Google Security Blog.
[22] Ján Jergus. 2019. HHVM 4.22.0. HHVM Blog.
[23] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL (2017).
[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021).
[25] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language* (2nd ed.). No Starch Press.
[26] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (2006).
[27] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. ASPLOS-XIII.*
[28] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A Type System for Borrowing Permissions. In *Proc. POPL.* 557–570.
[29] Steven Pack. 2018. Serverless Rust with Cloudflare Workers. Cloudflare Blog.
[30] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proc. PLDI.*
[31] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proc. PLDI.*
[32] James Reinders. 2007. *Intel Threading Building Blocks.* O'Reilly & Associates.
[33] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention through Priority Updates. In *Proc. SPAA.* 152–163.
[34] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation* 132, 2 (1997).
[35] Joshua Yanovsky, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: separating permissions from data in Rust. *Proc. ACM Program. Lang.* 5, ICFP (2021).
[36] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How Do Fixes Become Bugs?. In *Proc. ESEC/FSE'11.*