# When Is Parallelism Fearless and Zero-Cost with Rust?


Javad Abdi*
Cerebras
Toronto, Canada
javad.abdi@cerebras.net

Gilead Posluns
University of Toronto
Toronto, Canada
gil.posluns@mail.utoronto.ca

Guozheng Zhang
University of Toronto
Toronto, Canada
guozheng.zhang@mail.utoronto.ca

Boxuan Wang
Columbia University
New York, USA
bw2812@columbia.edu

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



## ABSTRACT

The Rust programming language is lauded for enabling *fearless concurrency* with zero cost: detecting concurrency errors at compile time. Given the enduring difficulty of parallel programming in other languages, this implied panacea warrants analysis. In particular, the efficacy of Rust across types of parallelism remains unexplored. Is parallel programming always devoid of fear with Rust? We answer this question through a case study, porting 14 benchmarks with abundant regular and irregular parallelism from C++ to Rust and reporting our experience and observations. We find that Rust, with the Rayon library, indeed delivers fearlessness for program phases comprising only regular parallelism, e.g., *prefix-sum*. However, for applications with *any* irregular parallelism, the programmer must choose between unsafe code or high-overhead dynamic checks with errors that manifest at run time, leaving the arduous task of parallel programming as scary with Rust as with its predecessors.



## CCS CONCEPTS

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

## KEYWORDS

Rust; fearless concurrency; zero-cost abstraction; regular parallelism; irregular parallelism

**ACM Reference Format:**
Javad Abdi, Gilead Posluns, Guozheng Zhang, Boxuan Wang, and Mark C. Jeffrey. 2024. When Is Parallelism Fearless and Zero-Cost with Rust?. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3626183.3659966


*This work was done while Javad Abdi was at the University of Toronto.



*SPAA '24, June 17–21, 2024, Nantes, France*



ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659966


## 1 INTRODUCTION

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

Computer 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 performance and productivity booms since the shift to multicores [122] in part due to their abundant "obvious" [3] sources of parallelism. On the other hand, parallel algorithm experts have uncovered surprising opportunities for task-level parallelism in conventionally challenging domains [7, 29, 30, 46, 47, 87, 112]. Algorithms in the former domains typically have abundant *regular* parallelism, where data and control dependences among tasks are statically identifiable. Algorithms in the latter face *irregular* parallelism, with dynamically manifesting data and control dependences [87]. Scheduling tasks and synchronizing irregular data accesses continue to tax programmers with pitfalls such as nondeterminism [66], deadlock, data races, and other concurrency bugs [68, 134]. A plethora of work in programming languages [8, 19], extensions [9, 11, 26, 93], and type systems [44, 78] has sought to curtail these errors, yet few have reached mainstream adoption.

Rust is gaining traction as a systems programming language for building fast and reliable applications [70]. It has been the most loved [115] or admired [116] language on the Stack Overflow Developer Survey for eight consecutive years, and has been adopted into major open-source and proprietary software [6, 18, 33, 48, 52, 54, 55, 83, 103, 118]. Rust unites higher-level *safety* and lower-level resource control by leveraging its type system, built atop prior work on ownership [45, 124], to capture memory and concurrency bugs at compile time. The golden rule of the Rust type system is that aliasing implies 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* [58] or `AXM` for short [133]. 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" [63, Chapter 16].

Unfortunately, Rust's `AXM` restriction precludes important instances of parallelism: sometimes tasks must mutate aliased state. Like some type-safe languages, Rust offers flexibility through **`unsafe`** code blocks where ownership rules can be violated, raw pointers manipulated, and other unsafe operations performed [102]. Conventional wisdom suggests that Rust programmers *(i)* minimize

the footprint of their unsafe code, and *(ii)* encapsulate unsafe code within *safe APIs* with run-time checks [4, 59, 63].

The interaction between safe and unsafe Rust code has garnered attention in prior work and we focus on concurrency and parallelism. RustBelt [58] and RustBelt Relaxed [27] prove the soundness of the Rust type system and provide tools for verifying encapsulation of unsafe code. Qrates [4] 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 (e.g., an OS, browser, and blockchain) [89]. While prior work investigated Rust support for multi-threaded systems software, as far as we are aware, Rust's purported fearless concurrency has yet to be studied *through the lens of regular vs. irregular parallelism* [87].

This paper presents a case study of Rust's parallelism support across 14 benchmarks with fine-grain regular and irregular task-level parallelism, drawn from the Problem Based Benchmark Suite (PBBS) [2, 113] or using a MultiQueue [95]. In one view, our work grapples with the promise of Rust for parallelism as prior work did with transactional memory [69, 99]. We provide a spectrum of fear in parallel programming (Sec. 3). We report our observations in porting C++ benchmarks to Rust and present Rust's strengths, shortcomings, and trade-offs along the way. We find that Rust indeed makes us fearless when expressing applications with regular accesses (Sec. 4) but fear remains in case of irregular accesses (Sec. 5). Nevertheless, the regularity of task scheduling does not impact the level of fearlessness that Rust provides (Sec. 6). Resulting from this study is the Rust Parallel Benchmarks suite (RPB) [98], with switches to toggle unsafe parallel features. We characterize the frequency of regular and irregular access patterns in the suite and what Rust techniques they demand. Our Rust-based benchmarks are gmean 1.44× slower than their equivalents in C++ (Sec. 7).

In short, this paper makes the following key contributions:

- A case study of Rust's support for regular vs. irregular task-level parallelism, focusing on compile-time vs. run-time knowledge of *(i)* accesses to shared data, and *(ii)* task scheduling.
- Recommended Rust expression of regular/irregular patterns.
- The first Rust benchmark suite with fine-grain regular/irregular parallelism to seed language, compiler, and runtime research.
- A performance evaluation of RPB at 24 cores, showing it is competitive with C++ equivalents of the benchmarks.

# 2 BACKGROUND

## 2.1 Rust's type system makes it safe

Rust is a safe and fast programming language. Rust safety is rooted in its three ownership rules: each value *(i)* has an owner (e.g., named function parameter or anonymous temporary); *(ii)* has only one owner at a time; and *(iii)* gets dropped after its owner goes out of scope [63]. A value's ownership can be transferred (e.g., from caller to callee function or from producer to consumer thread), but ownership transfer alone is too restrictive for common programming patterns (e.g., a caller losing access). To relax these rules, Rust code can temporarily *borrow* references to values. Each reference is coupled with a *lifetime*, the scope in which the reference is valid. The heart of a Rust compiler (e.g., `rustc`), is the *borrow checker* that checks reference validity and importantly forbids aliased mutable

```
let mut sum = 0;
thread::scope(|s| { // First mutable reference ---+-'1
  s.spawn(|| { // Second reference = Error --+-'2 |
    sum += vector[..mid].iter().sum(); //    |    |
  });                                  // ---+    |
  sum += vector[mid..].iter().sum();   //         |
});                                    // --------+
```

a) The `rustc` borrow checker catches a data race on `sum` due to overlapping lifetimes `'1` and `'2` with mutable references to `sum`.

```
let locked_sum = RwLock::new(0usize);
thread::scope(|s| {
  s.spawn(|| {
    let local_sum = vector[..mid].iter().sum();
    *locked_sum.write().unwrap() += local_sum;
  });
  let local_sum = vector[mid..].iter().sum();
  *locked_sum.write().unwrap() += local_sum;
});
let sum = *locked_sum.read().unwrap();
```

b) Synchronization, a form of interior mutability, solves the issue.

**Listing 1: Two-threaded summation of a vector in Rust.**

references with overlapping lifetimes. A reference can write to a value at a given time iff it is the only reference to that value (AXM).

Ownership and borrowing rules enable Rust to enforce memory safety and statically rule out data races. Our paper focuses on the latter. A program has a data race if there exists an execution in which at least one access in a pair of conflicting accesses [110] from different threads is not atomic [53, 107]. Since Rust forbids mutable aliasing (AXM), it rules out the possibility of threads making concurrent accesses to mutable shared state. For example, Listing 1(a) shows an incorrect parallel vector summation, as increments to `sum` are not atomic. The `rustc` borrow checker catches this bug at compile time, because `sum` is mutably aliased between two threads.

*Interior mutability* enables programmers to mutate shared data when it is safe to do so [63]. For example, Listing 1(b) eliminates the data race by synchronizing with a `RwLock`, but Rust's type system does not actually differentiate between synchronized and unsynchronized accesses. Instead, the `read` and `write` methods of `RwLock` take immutable references to the lock while allowing mutations of the underlying data. This instance of interior mutability is safe because `RwLock` *dynamically* enforces AXM for the underlying data.

## 2.2 Unsafe Rust is more expressive than Rust

Although the Rust type system has been proven sound [27, 58], it is incomplete [96]. Its strictness precludes implementing essential patterns and data structures, like a sequential doubly linked list [4, 76]. *Unsafe* Rust is an embedded language that expands expressiveness to code for which `rustc` cannot guarantee safety. Programmers can demarcate **unsafe** blocks where they perform operations that `rustc` cannot guarantee to be safe, such as dereferencing pointers or accessing arrays without bounds checking, among others [102].

When **unsafe** blocks are necessary, best practice minimizes code base pollution through the use of safe abstractions, known as *interior unsafe* [89], that move safety checking from compile time to run time [4, 59, 63]. In fact, the vectors, threads, and `RwLock` of Listing 1 have many interior unsafe functions. Yet the programmer is unaware or unconcerned about their **unsafe** blocks because the interfaces encapsulate them. Listing 2 shows an interior-unsafe function that removes an element from a vector (`self`). Unsafe

```
1 fn remove(&mut self, index: usize) -> T {
2   assert!(index < self.len); // run-time safety check
3   unsafe {
4     let ptr = self.as_mut_ptr().add(index);
5     let ret = ptr::read(ptr);
6     ptr::copy(ptr.add(1), ptr, self.len - index - 1);
7     self.len -= 1;
8     return ret;
9   }
10 }
```

**Listing 2: A Rust `std` library interior-unsafe function that validates unsafe code with a run-time assertion.**

operations must be wrapped in an **`unsafe`** block. Conventional wisdom suggests preceding them by checks that validate the contract of the safe abstraction. Line 2 verifies having a valid `index`. Interior unsafe does not necessarily crash upon failed validation: `Vec::pop` returns `None` for an empty vector. Interior unsafe is also common in multithreaded code that mutates shared state (Sec. 4, 5, 6).

## 2.3 Rust libraries ease expression of parallelism

Rust's standard library provides basic tools for parallelism like threads, channels, locks, and async tasks, but external libraries build on these to provide safe higher-level abstractions [65] to make parallelism easier. Rayon [71] expresses data parallelism through constructs such as parallel iterators. Tokio [125] asynchronously performs network and filesystem I/O operations. Other libraries provide expanded [24] or optimized [86, 114] synchronization primitives, concurrent data structures [28], and thread pools [123].

These libraries commonly use interior-unsafe functions to circumvent Rust's strict enforcement of AXM, and broaden its capabilities. Rayon's mutable parallel iterators use interior-unsafe to share mutable references to *disjoint* parts of a data structure among tasks (Sec. 4.2). *crossbeam* [24] provides a thread-safe alternative to `Cell` with explicit annotations of thread safety in unsafe Rust.

These efforts target common use cases and relieve beginners of the burden of parallel programming without the AXM guardrails. Yet, the typically avoided complex cases remain as challenging as ever.

# 3 STUDY OVERVIEW

We investigate when Rust delivers or fails in its promise of fearless concurrency through two lenses: *(i)* regular vs. irregular accesses, and *(ii)* static vs. dynamic task scheduling. Fig. 1 frames our case study, adapting the Tao Analysis of Algorithms [87] beyond graphs. Three dimensions affect the regularity of task-level parallelism.

- *Data Structure:* Tasks might access a shared data structure, whose topology influences regularity of accesses. *Structured* data (e.g., arrays or matrices) are described using few parameters (e.g., length or rows/columns, respectively). *Unstructured* data (e.g., arbitrary graphs) require verbose description (e.g., CSR).
- *Operator:* Tasks may operate on shared data within a phase. Tasks are *local read-write* operators of a data structure if they make task-private accesses to its sub-elements (e.g., each task reads and/or writes a distinct vertex in a graph). Tasks are *arbitrary read-write* operators of a data structure if they read and write (potentially) overlapping sub-elements (e.g., each task reads a neighborhood of vertices and updates one or more of them). Tasks are *readers* of a data structure when none of them write to it.

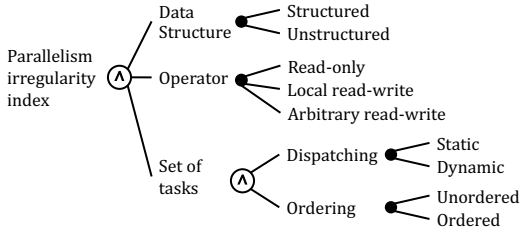



**Figure 1: Analysis of Rust support for parallel patterns.**

- *Set of Tasks:* Whereas the previous dimensions affect regularity of data dependences, two factors affect control dependences: *(i)* when tasks are discovered and *(ii)* if tasks must execute in (partial) order. Tasks are *statically dispatched* when the set of tasks is known before entering a parallel phase, but are *dynamically dispatched* if tasks find and schedule new work on the fly.

Navigating this space implies the regularity of an algorithm or a phase. For example, any operator on structured data with static task scheduling is regular (e.g., a parallel reduction on an array or a stencil computation). In contrast, a parallel relaxed Dijkstra's algorithm [88] is irregular: reading and writing arbitrary vertices of unstructured data with dynamic and ordered task scheduling.

We find that the regularity of task accesses to shared data is the primary factor to facilitate or impede fearless parallelism. Rust excels at making programmers fearless when expressing parallel patterns with regular write sets (Sec. 4). Challenges emerge with local read-write irregular parallelism and arbitrary read-write operations remain as scary as ever. (Sec. 5). Interestingly, dynamic task scheduling does not affect fearlessness, only task accesses (Sec. 6).

## 3.1 What is fearless concurrency?

The anticipated danger that inspires fear in parallel programmers is the potential for concurrency errors that manifest at run time. Sequential errors are bad enough, but it is the nondeterminism [66] of concurrency errors that is so nefarious. Detecting a bug is elusive and reproducing can be even harder. Fearless concurrency is the Rust Team's nickname for their goal that "[...] you can fix your code while you're working on it rather than potentially after it has been shipped to production" [63]. 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 concurrency error is caught at compile time. This restrictive situation is likely rare.

At the other extreme, Rust can rule out data races for programs requiring lock-based or lock-free synchronization [15]. However, data-race freedom does not imply freedom from atomicity violations [39], order violations [68], deadlocks, and livelocks.

Between these extremes are programs that algorithmically elide synchronization but require interior unsafe APIs to placate `rustc`. Such functions use static and/or dynamic checks to validate their contracts. With dynamic checks, validation failures move from compile time to run time. Although encapsulated dynamic checks move an error's symptom close to the cause, crashes in production remain possible, leaving fear with some hope for a clear postmortem.

Taken together, we find that fearless concurrency is better interpreted as a spectrum, illustrated in Fig. 2: ideally eliminating

**Fearless:** Concurrency errors get caught at compile time
**Comfortable:** Errors get caught at run time (symptoms close to cause)
**Scared:** Concurrency errors may happen without being detected

**Figure 2: A spectrum of fear in parallel programming.**

any fear of concurrency errors at compile time (fearless), but if not possible, keeping run-time error symptoms close to their causes (comfortable), or otherwise providing no guarantees of reproducing the cause nor the symptom (scared).

## 3.2 Study methodology

We derive our findings by studying how parallel algorithms experts express parallelism. Unfortunately, there is a lack of Rust benchmark suites with abundant irregular parallelism. Therefore, we port 12 benchmarks from PBBS [2, 113] (a representative subset) and 2 using the MultiQueue priority scheduler [95]. This methodology is analogous to how Ruan et al. replaced locks with transactions in legacy code (memcached) [99]. The key distinction is that we target algorithm implementations instead of legacy systems software code. This enables us to evaluate various parallel patterns. For each pattern, we show that its naive implementation in Rust that imitates C++ may require unsafe Rust. We then provide idiomatic Rust solutions, categorize their fearlessness, and report on performance.

PBBS is a C++ benchmark suite that encompasses diverse parallel patterns and application domains such as text, geometry, graphs, and more. PBBS provided or influenced the implementations of several state-of-the-art parallel algorithms [7, 30, 130]. PBBS incorporates both regular and irregular parallelism as inseparable parts of each benchmark, making it an appropriate choice for our study.

The MultiQueue broadens our study with dynamic task scheduling, which is lacking in PBBS. This relaxed priority queue grants probabilistic rank guarantees and scales well to large numbers of threads [88]. We use the MQ as a task scheduler in two benchmarks.

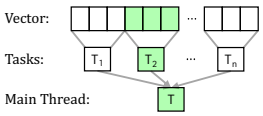
We use Rayon for some parallel operations and runtime scheduling. Rayon is a Rust work-stealing-based data-parallelism library that is adopted in major Rust projects including Firefox, Servo, Solana, Meilisearch, HHVM, and the `rustc` compiler. Rayon is written by Rust experts who made contributions to `rustc` [71], drawing inspiration from Cilk [9]. We extensively use Rayon's parallel iterators, which resemble standard Rust iterators, but its consumers (e.g., `map`, `reduce`, and `for_each`) execute in parallel by interior-unsafe functions. Appendix A elaborates on our rationale for using Rayon. When Rayon lacks our desired functionality, we use Rust's standard library tools such as threads or mutexes.

We organize our case study from straightforward types of parallelism (Sec. 4) to more difficult (Sec. 5 and Sec. 6). Overall, we find that irregular writes or the combination of regular writes with irregular reads from shared data preclude Rust support for fearlessness, necessitating the conventional synchronization that has scared programmers for decades.

# 4 REGULAR ACCESSES TO SHARED DATA

When the set of tasks and their data dependences are statically known or parameterized, this *regular parallelism* can be validated at compile time, eliminating most run-time overheads of parallel scheduling. This includes read-only operators on any data structure

a) Each task reads its chunk; main thread gathers intermediate sums.




```
int* sums = par_for (i=0; i<chunks_no; i++) {
  int sum = 0, s = i*chunk_size, e = (i+1)*chunk_size;
  for (j=s; j<e; j++) sum += vector[j];
  return sum;
}
int result = 0;
for (i=0; i<chunks_no; i++) result += sums[i];
```

b) C-like code

```
let result = vector.par_chunks(chunk_size)
                   .map(|chunk| chunk.iter().sum())
                   .sum();
```

c) Rust easily expresses read-only parallelism.

```
let mut result = 0;
vector.par_chunks(chunk_size)
      .for_each(|chunk| result+=chunk.iter().sum()); // error
```

d) `rustc` detects a data race: unsynchronized writes to `result`.

**Listing 3: `AXM` implementation of parallel sum.**

or local read-write operators on structured data (Fig. 1). For now, we assume that the set of tasks is statically known and they are unordered. At the simplest extreme, tasks that only read shared collections are trivial to check for errors. Rust indisputably keeps read-only parallelism fearless by tracking reference mutability (Sec. 4.1) to detect any errors. Writes to shared data cause dependences, but when they are statically analyzable (Sec. 4.1) or constrained (Sec. 4.2), Rust and Rayon, respectively, enable fearless parallel expression among independent tasks.

## 4.1 Statically safe patterns `rustc` understands

Rust provides its strongest guarantees for phases where `rustc` validates conformance to `AXM`. `AXM` is trivially true for read-only parallelism: *aliasing XOR 0* allows aliasing. However, Rust still shines when expressing tasks with local reads and writes on statically sized, structured data like arrays. `rustc` tracks ownership at fine granularity for these data structures, down to individual elements. With no aliasing, *0 XOR mutability* permits task-private writes.

**Immutable shared data accessors:** With read-only parallelism, tasks do not mutate shared data, but instead summarize a collection into a small value to be returned and sequentially processed and/or merged into an array. Listing 3(b) demonstrates this pattern in a C-like `reduction`. Each task reads a chunk of `vector` and returns its `sum`. This pattern is easily implemented in Rust as it upholds `AXM`. Listing 3(c) shows the reduction in Rust, where each iteration's closure immutably borrows a chunk and accesses no other data.[1]

Read-only parallelism is fearless in Rust because the compiler detects any unintended writes to shared data. In Listing 3(d), a task's attempt to mutate `result` is rejected by `rustc` due to potential data races. In read-only parallelism, tasks do not mutate shared references. If the borrow checker detects that the `for_each` closure borrows any mutable references, it will reject compilation.

Read-only parallelism is the most straightforward to express and easily aligns with Rust's `AXM` rules. However, it constitutes only

[1]Rayon may use unsafe Rust to collect the intermediate results, but those unsafe blocks are encapsulated behind safe APIs and do not generate run-time errors.

11% of the parallel code in our benchmarks. A language supporting fearless parallelism must support a broader set of patterns.

**Observation 1:** *Safe Rust **enables fearless** read-only parallelism where mutable references to shared data are absent.*

**Statically analyzable writes:** Rust's *destructuring* mechanism allows `rustc` to track references at a fine granularity for statically sized structured data like arrays. Since destructuring rules out aliasing, then *0 XOR mutability* ideally permits task-local writes. The following example destructures `array` into its `first` and `last` elements, and creates slice `middle` from the remaining elements. Concurrent tasks can mutably borrow these three parts.

```
let [first, middle@.., last] = &mut array;
```

Unfortunately, destructuring has two limitations: *(i)* it does not support dynamically sized data structures like vectors, crucial for writing input-size-dependent applications, and *(ii)* it inhibits many parallel patterns. The following shows destructuring failing to split `array` into three equal-sized chunks to evenly distribute work.

```
let [a@0..3, b@3..6, c@6..9] = &mut array;  // compile error
```

**Observation 2:** *Although destructuring **grants fearlessness**, it is cumbersome for expressing load-balanced parallelism.*

## 4.2 Statically safe interior unsafe patterns

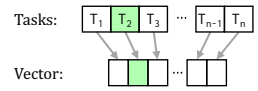
Rayon enables Rust to provide fearless parallel programming for patterns with local reads and writes to *dynamically sized* structured data. `rustc` tracks references of dynamically sized data structures (e.g., vectors) at the granularity of the whole data structure, causing inter-task aliasing. Rayon mitigates this with interior-unsafe functions that statically constrain each task's accesses to a unique subset of elements, enabling some parameterized access patterns to conform to `AXM` at a fine per-element or chunk granularity. These are zero-cost abstractions: the programmer pays no run-time overhead for gaining safety [63]. Two patterns in PBBS fit this category.

**Striding writes (`array[i] = f()`):** In this `Stride` pattern, each task $T_i$ performs its local access to index `i` of the mutably aliased target array[2] and only reads from other immutable shared data structures. The `C` code in Listing 4(b) stores the square of each `vector` element in place, using `Stride`. This pattern is statically known to be safe and race-free, but `rustc` does not understand that the accesses are non-overlapping because of the aliased references to the whole `vector`. Correctly but naively implementing this pattern in Rust (Listing 4(c)) triggers the borrow checker to reject the program. This conservatism remains useful when the programmer actually makes an error, like the data race in Listing 4(d).

Rayon convinces `rustc` that this pattern is safe through careful use of interior-unsafe functions that restrict each task to access only the element of the mutably shared `vector` passed to it. Each task in Listing 4(e) can only access `vector` through reference `vi` (`vector[i]`). Using unsafe blocks, Rayon's `par_iter_mut` interior unsafe function creates a mutable parallel iterator over `vector` and passes each `vector` element, one by one, to the closure passed to `for_each`. `par_iter_mut` is a zero-cost abstraction that only requires static checks. While iterations of the loop are executing, the mutable reference to the `vector` remains alive (lifetime `'1`). If

[2]In general, striding indexes are $k * i$, but PBBS only uses $k = 1$.

a) Task $T_i$ accesses element `i` of the `vector`.




```
par_for (int i=0; i<n; i++) { vector[i] *= vector[i]; }
```

b) C-like code

```
(0..n).into_par_iter()
      .for_each( |i|
        vector[i] *= vector[i]
      );
```

c) `rustc` throws compile error: `vector` is mutably aliased.

```
(0..n).into_par_iter()
      .for_each( |i|
        vector[i] *= vector[i-1]
      );
```

d) Not allowing mutable aliases, `rustc` catches a data race.

```
vector.par_iter_mut() // -+-'1
      .for_each( |vi| //  |
        *vi *= *vi    //  |
      );              // -+
```

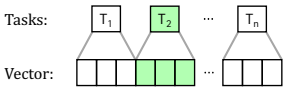
e) Rayon helps Rust to safely express this pattern.

```
vector.par_iter_mut()
      .enumerate()
      .for_each( |(i, vi)|
        *vi *= vector[i-1]
      );
```

f) Using Rayon, data races will be caught as compile errors.

**Listing 4: Squaring elements of `vector` using `Stride`.**

Tasks access equally sized non-overlapping chunks.




```
vector.par_chunks_mut(size) // size == each chunk's size
      .for_each(|a| ...);   // a == vector[i*size..(i+1)*size]
```

**Listing 5: Tasks access equally sized chunks using `Block`.**

any closure attempts to access `vector` directly (Listing 4(f)), the borrow checker rejects compilation to avoid data races.

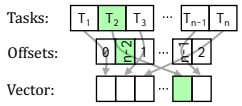
**Blocking writes (`array[i*size..(i+1)*size] = f()`):** Whereas `Stride` pattern tasks operate on individual elements, `Block` pattern tasks perform local accesses on equally sized chunks of the mutably aliased array. Like `Stride`, these chunks are statically known to be non-overlapping. Rayon provides the `par_chunks_mut` function to express this pattern, shown in Listing 5.

**Observation 3:** *Using elegant static checks, Rayon interior-unsafe functions express tasks with local reads and writes on structured data, **retaining fearless parallelism** despite Rust's coarse-grain ownership tracking.*

# 5 IRREGULAR ACCESSES TO SHARED DATA

When data dependences among tasks are known only at run time, correct interleaving of their irregular accesses must be dynamically validated or enforced. We call these local read-write operators on unstructured data or arbitrary read-write operators on any data structure (Fig. 1). We address ordered tasks and dynamic scheduling in Sec. 6. On one hand, algorithm invariants can guarantee task independence within a phase, yet the exact write locations depend on run-time values. Run-time checks validate the algorithm-to-code translation (Sec. 5.1). On the other hand, reads and writes to overlapping data cause dependences that require synchronized access interleavings (Sec. 5.2). In both cases, `rustc` highlights potential data races by tracking reference mutability and lifetimes. However, the former eliminates zero-cost abstraction and the latter leaves programmers with fear of synchronization.

a) Tasks indirectly access independent elements of the vector.




```
par_for (int i=0; i<n; i++) out[offsets[i]] = input[i];
```

b) C-like code

```
(0..out.len()).into_par_iter()
              .for_each(|i| out[offsets[i]] = input[i]);
```

c) Naive implementation in Rust that causes a compile error.

```
(0..out.len())
  .into_par_iter()
  .for_each( |i| unsafe { *out_ptr.add(offsets[i]) = input[i]; });
```

d) Unsafe implementation in Rust: scary.

```
(0..out.len()) // Here, out is a Vec<AtomicU32>
  .into_par_iter()
  .for_each(|i| out[offsets[i]].store(input[i], Relaxed));
```

e) Atomics can placate `rustc`: scary.

```
out.par_ind_iter_mut(offsets)
   .enumerate()
   .for_each(|i, oi| *oi = input[i]); //oi==out[offsets[i]]
```

f) Interior-unsafe implementation: comfortable but slow. `par_ind_iter_mut` checks the offsets to be unique at run time.

**Listing 6: `SngInd` as a part of a *sort* routine.**

## 5.1 Local reads and writes of unstructured data

Rust offers limited support when tasks are independent based on algorithmic guarantees. For two common patterns, the programmer must choose among three undesirable solutions: *(i)* unsafe Rust (fear of races), *(ii)* unnecessary synchronization (fear of synchronization), or *(iii)* run-time checks (comfort with overheads).

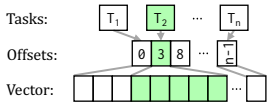
**Single-valued indirect writes (`array[B[i]] = f()`):** In this `SngInd` [119] pattern, tasks read and write to independent indices of a shared array. Tasks could be read-only operators of other data structures. Listing 6(b) showcases `SngInd` in `C++` as a part of a *sort* routine. Arrays `input` and `offsets` are immutable. Unlike `Stride`, indices into mutable `out` are indirectly determined via `offsets` values and cannot be proven independent, neither statically nor with cheap run-time checks. There is no need for synchronization: the (correctly implemented) algorithm guarantees unique offsets.

This pattern requires run-time checks to ensure the uniqueness of `offsets`. Implementation bugs can lead to duplicates, creating data races. `rustc` recognizes this risk and rejects compiling the naive implementation of this program (Listing 6(c)). However, unlike in Sec. 4, there is no dedicated support for this specific pattern.

Rust offers three solutions. Listing 6(d) **`unsafe`**ly dereferences a pointer to write, leaving the programmer scared. Listing 6(e) atomically stores with relaxed ordering to placate `rustc`, but does not guarantee uniqueness, leaving programmers scared with potentially non-deterministic errors. Listing 6(f) introduces `par_ind_iter_mut`, our new interior-unsafe function. It checks `offsets` for uniqueness in parallel then uses a Rayon parallel iterator that **`unsafe`**ly passes `out`'s `offsets[i]`th element to task $T_i$. The programmer rests assured that this interior unsafe approach catches errors at run time, upgrading fear to comfort. Unfortunately, the run time of the check can outweigh the useful work (e.g., 2.8× slowdown in `lrs`). The programmer must choose between fear and overhead.

The `SngInd` pattern generalizes beyond offset arrays. For example, a pure [43] offsets function or even a collection of pointers could similarly be checked for uniqueness with an interior unsafe function. These were rare in our benchmarks.

a) Tasks indirectly access independent chunks of the `vector`.




```
par_for (i=0; i<offs_len-1; i++)
  for(j=offsets[i]; j<offsets[i+1]; j++) out[j] = f(...);
```

b) C-like code

```
out.par_chunks_ind_mut(offsets)
   .for_each( |local_out|
     local_out.iter_mut().for_each(|oi| *oi = f(...))
   );
```

c) Safe implementation using `par_ind_chunks_mut`

**Listing 7: Ranged indirect access pattern (`RngInd`).**

**Ranged indirect writes (`array[B[i]..B[i+1]] = f()`):** `RngInd` is to `SngInd` as `Block` is to `Stride`. In this pattern, `offsets[i]` and `offsets[i+1]` provide the start and end, respectively, of a contiguous chunk passed to task $T_i$. Listing 7(b) shows `C` tasks that write to a contiguous range of `out`. `SngInd` solutions (unsafe, synchronization, interior-unsafe) and their trade-offs apply to `RngInd`.

However, in our observations, the prevailing version of this pattern is when chunk order aligns with task iteration order.[3] We can exploit this order to check that `offsets` monotonically increases to guarantee non-overlapping chunks. Listing 7(c) safely expresses this pattern with our proposed `par_ind_chunk_mut`. It creates a Rayon parallel iterator that dynamically ensures `offsets` monotonically increases when Rayon splits the iterator among tasks. Unlike `SngInd`, this is cheap, making *comfort* an easier trade-off to accept.

This pattern can also extend beyond an array of offsets. The same arguments of `SngInd` apply.

**Observation 4:** *For patterns with mutable access to independent subsets of unstructured data, Rust **abandons zero-cost fearless parallelism**: the programmer must choose between **`unsafe`** performant code or unnecessary synchronization and dynamic checks to prove independence.*

## 5.2 Arbitrary reads&writes of unstructured data

Rust does not eliminate fear when tasks have irregular true data dependences. For patterns discussed so far, task read and write sets are independent per phase. However, arbitrary read-write (`AW`) tasks have occasional data dependences through mutable unstructured data. Such behavior is found in application domains spanning graph analytics, geometry, statistical inference, and others [87]. These dependences necessitate run-time mechanisms like synchronization to enforce correct memory access interleavings.

**Overlapping reads and writes:** When distinct tasks have conflicting accesses to the same location, synchronization is usually required to prevent incorrect access interleavings. Listing 8(a) shows overlapping read-modify-writes to a parallel hash table in PBBS. Each task uses function-based indirection to insert into some table entry. Unlike in Sec. 5.1, hashes are not unique so tasks may be dependent. Similar overlapping conflicting accesses are common in graph algorithms like push-based PageRank [131].

[3] Tasks are independent so are unordered in terms of execution.

a) C++

```
class HashTable {
  T[] table;
  bool insert(T v) {
    if (CAS(&table[hash(v)], EMPTY, v)) return true;
    else ...
  }
};
```

```
struct HashTable { table: Vec<T> }
fn insert(&mut self, v: T) { self.table[hash(v)] = v; }
```

b) Mutating without synchronization is forbidden in Rust.

```
struct HashTable { table: Vec<Mutex<T>> }
fn insert(&mut self, v: T) { *self.table[hash(v)].lock() = v; }
```

c) Even with synchronization, `rustc` rejects mutable borrows.

```
struct HashTable { table: Vec<Mutex<T>> }
fn insert(&self, v: T) { *self.table[hash(v)].lock() = v; }
```

d) `rustc` requires marking the borrow as immutable to compile.

**Listing 8: Hash Table: a data structure with AW.**

Conflicting irregular parallel accesses require synchronization, which Rust correctly enforces. Accessing an unsynchronized hash table across threads triggers compile-time errors in Listing 8(b). Listing 8(c) wraps each hash table entry with a `Mutex` and uses `lock()` to get a synchronized mutable reference to the entry. Interestingly, the compile error persists, even with this synchronized `insert`, because `insert` mutably borrows `self`—Rust does not differentiate synchronized mutable references and unsynchronized ones. Best practice to solve this is making the hash table interior mutable, enabled by `Mutex` itself being interior mutable. We argue that forcing the programmer to mark a reference as immutable, even when they know it will be mutated (e.g., `insert`), is not intuitive and fails to make them fearless. Introducing a third type of reference, such as a mutable *shared* reference[78], would be more intuitive.

Although `rustc` rules out data races by identifying a lack of synchronization, it does not capture synchronization errors. For example, fine-grain locking in Rust remains as susceptible to deadlock and livelock as in its decades of predecessors. Likewise, programmers can incorrectly apply atomic types for lock-free synchronization allowing incorrect interleavings of memory accesses from different tasks. Synchronization with Rust remains as scary as ever.

**Benign races** appear useful for some parallel operations but require extreme care to use correctly if even possible [12]. For example, the following appears to be a truly benign race: a snippet of a parallel suffix array calculation that finds distinct characters in `string`.

```
let present = vec![0u8; 256];
string.par_chars().for_each(|c| present[c as usize] = 1);
```

The writes of many tasks will race on various elements of `present`, but the result remains consistent, regardless of their interleavings, since all tasks write the value 1. However, this is not portable. Compiler transformations can transform benign races into non-benign ones [12]. Rust uses the C++ memory model for atomics [102]. Both languages are strive for portable code. At the language level, these stores would seem to be benign races; one might assume they will map to a single instruction. Compilers generate code across a diversity of ISAs, including different bit widths in the data path. Hypothetically, if `present` were an array of 64 bit integers but the ISA only provided 32 bit stores, the compiler would necessarily break these stores into two, losing atomicity. This issue is one of several reasons why the C++ standard disallows "potentially concurrent conflicting actions" [53]. `rustc` correctly refuses to compile this code, forcing the programmer to use atomic stores. Porting PBBS code to Rust revealed this issue in PBBS to us, which can be fixed by using relaxed atomic stores.

**Observation 5:** *Rust programmers expressing arbitrary read-write tasks are safe from data races, but are otherwise left* ***scared****.*

## 6 TASK SCHEDULING WITH RUST

We have so far assumed that the set of tasks at each parallel region (third dimension of Fig. 1) is statically known or easily describable. However, another dimension of (ir)regularity involves the task dispatching scheme. Tasks may discover new work and dynamically spawn new tasks that execute in the same parallel region. We consider two such cases: *(i)* nested fork-join in the form of *divide-and-conquer* using the existing Rayon task scheduler, and *(ii)* dynamic priority-ordered scheduling using our Rust implementation of the MultiQueue [88, 95]. The former is well-structured, with each parent task waiting for its children to join. In the latter, a parent pushes its children into a queue then finishes its own execution. Worker threads later pop and run the children.

**Fork-join scheduling:** Divide-and-conquer is a popular technique which involves dividing a problem into sub-problems, solving them recursively, and merging the results. Sub-problems are easily solved in parallel because they are independent [9, 87]. Listing 9's merge sort partitions an array in two and spawns a task to sort each part.

Forking and joining tasks in divide-and-conquer is straightforward and fearless with Rust. The `join` function offers a safe and high-level abstraction, and Rayon's well-established reputation gives confidence in a correct implementation. Consequently, the programmer only fears concurrency errors related to shared data accesses (Sec. 4,Sec. 5). With divide-and-conquer, children tasks should not have conflicting concurrent accesses, which `rustc` statically verifies by tracking lifetimes.

We did not observe non-strict fork-join [10] in our benchmarks, where child tasks join any task. However, we expect more challenges with Rust. This pattern could have arbitrary read-write tasks on unstructured data, scaring programmers (Sec. 5.2).

**Dynamic priority scheduling:** Some algorithms require [29] or prefer [80] tasks to execute in some priority order, necessitating manual thread management. We examine the MultiQueue (MQ) [95] concurrent priority scheduler. The MQ wraps a vector of sequential priority queues, each guarded by a lock. Push picks a random queue, locks it, and pushes the task. Pop locks two random queues, and pops from the queue with the higher priority top task. MQ approximates a sequential priority order rank with probabilistic

```
fn sort(input: &[T], output: &mut [T]) {
  if input.len() <= Threshold { ... } // go sequential
  else {
    let mid = output.len() / 2;
    let (l_in, r_in) = input.split_at(mid);
    let (l_out, r_out) = output.split_at_mut(mid);
    rayon::join( || sort(l_in, l_out), || sort(r_in, r_out) );
    ... // merge the two sorted halves
  }
}
```

**Listing 9: Merge sort using the *divide-and-conquer* pattern.**

rank guarantees [95]. Our `bfs` and `sssp` use the MQ with long-running worker threads that pop tasks from the MQ then execute them (potentially pushing new tasks) until the MQ is empty.

Programmer fear hinges on *(i)* the MQ correctness, *(ii)* proper worker threads' interactions with MQ, and *(iii)* tasks' access patterns. The first two primarily fall on the shoulders of the scheduler implementer, whereas the third is relevant to the user of the API.

Implementing the MQ scheduler is nontrivial, and we found Rust to leave us *scared*. MQ requires synchronization when threads access queues, achievable through the standard library `Mutex`. Rust `Mutexes` offer two advantages over `C++ Mutexes`. First, they encapsulate objects which prevents unsynchronized access and rules out atomicity violations on accesses to the internal sequential queues. Second, releasing a lock is done by dropping the `MutexGuard` returned by `lock()`, either manually or automatically when it goes out of scope. This ensures that lock release is never forgotten. However, using these locks leaves the MQ implementer susceptible to deadlocks and livelocks, and therefore scared. Nevertheless, once implemented correctly, callers do not fear *(i)* and *(ii)* when using the scheduler. This was presumably true of Rayon: its implementers would have faced fears when implementing the library, but they provided an interior-unsafe API to users.

Even assuming a correct Rust MQ implementation, using a dynamic priority scheduler via APIs is a mixed bag for programmers. MQ-based applications often operate on unstructured data with arbitrary read-write tasks (`AW`). As shown in Sec. 5.2, these irregular accesses are scary. However, this fear is not caused by tasks' scheduling scheme but instead by their accesses.

**Observation 6:** *Task scheduling **does not directly affect** the level of fear of parallel programming in Rust but instead the data access patterns of tasks.*

## 7 EVALUATION

Our case study so far focuses on the programmer's experience and fear(lessness) in expressing various parallel patterns. This section validates the importance and credibility of our derived conclusions and suggestions by answering three questions:

*(i)* Is irregular parallelism common enough to cause concern? *It's present in all benchmarks (29% of accesses).*

*(ii)* Do our case study and suggested solutions lead to good performance? *Yes. RPB is* 1.09× *faster and* 1.44× *slower than* `C++`*-based equivalents at 1 and 24 threads, respectively.*

*(iii)* What are the costs of avoiding **`unsafe`** code? *Using run-time checks for* `SngInd` *and synchronization for* `AW` *increase execution time by* 1.3× *and* 2.1× *on average at 1 and 24 threads.*

### 7.1 Methodology

**Experimental setup:** We run benchmarks on an AWS `c5.metal` instance with Ubuntu 22.04, disabling Intel Turbo Boost, hyperthreading, and Linux NUMA balancing, consistent with prior work [105]. We evaluate performance both on one and 24 cores (threads) on one socket. We compile RPB with `rustc` v1.69.0, in release mode (equivalent to LLVM's `-O3`). We use Rayon v1.7.0. We compile the original PBBS benchmarks with OpenCilk v2.0 [105] (a fork of LLVM 14) with `-O3`. OpenCilk outperforms the library-based implementation of Cilk [9]. We use wall-clock time to compare execution time.

**Table 1: Ported benchmarks and their parallel access patterns.**

| | | | Tasks' Accesses: Regular | | | | Tasks' Accesses: Irregular | | | Task dispatch | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Abbrv.** | **Benchmark name** | **Inputs** | `RO` | `Stride` | `Block` | `D&C` | `SngInd` | `RngInd` | `AW` | `static` | `dynamic` |
| `bw` | Burrows–Wheeler decode | wiki [2] | ✔ | ✔ | | | ✔ | ✔ | ✔ | ✔ | |
| `lrs` | longest repeated substring | wiki | ✔ | ✔ | ✔ | | ✔ | ✔ | | ✔ | |
| `sa` | suffix array | wiki | ✔ | ✔ | | ✔ | ✔ | ✔ | | ✔ | |
| `dr` | Delaunay refinement | kuzmin [2] | ✔ | ✔ | | | ✔ | ✔ | ✔ | ✔ | |
| `mis` | maximal independent set | link, road | ✔ | | ✔ | | ✔ | ✔ | | ✔ | |
| `mm` | maximal matching | rmat, road | ✔ | | ✔ | | ✔ | ✔ | | ✔ | |
| `sf` | spanning forest | link, road | ✔ | | ✔ | | ✔ | ✔ | | ✔ | |
| `msf` | minimum spanning forest | rmat, road | ✔ | | ✔ | ✔ | ✔ | ✔ | | ✔ | |
| `sort` | comparison sort | exp. [2] | ✔ | | ✔ | ✔ | | ✔ | | ✔ | |
| `dedup` | remove duplicates | exponential | ✔ | ✔ | | | | | ✔ | ✔ | |
| `hist` | histogram | exponential | ✔ | ✔ | | | | ✔ | ✔ | ✔ | |
| `isort` | integer sort | exponential | ✔ | | | | | ✔ | ✔ | ✔ | |
| `bfs` | breadth-first search | link, road | | | | | | | ✔ | | ✔ |
| `sssp` | single-source shortest path | link, road | | | | | | | ✔ | | ✔ |

**Table 2: Input graphs, their progeny, and their characteristics**

| **Name** | **Shorthand** | $\|V\|$ | $\|E\|$ | $\|E\|/\|V\|$ |
|---|---|---|---|---|
| **Hyperlink2012-hosts** [74] | link | 101 M | 2043 M | 20.1 |
| **R-MAT graph** [17] | rmat | 34 M | 200 M | 6.0 |
| **Full USA roads** [1] | road | 24 M | 58 M | 2.4 |

**Benchmarks:** Table 1 lists the 14 benchmarks we ported to RPB, along with their input sets and access patterns. For `bw` through `sort`, we use PBBS's existing `C++` code. We implement the `C++` versions of `bfs` and `sssp` using PBBS's graph data structure and our own MultiQueue. We run each benchmark 10 times at 24 cores and 3 times at 1 core, and report mean execution times. For `sort`, we use sample sort. Table 2 details the input graphs.

**Coverage of Parallel patterns:** We are not aware of any general-purpose taxonomy classifying all parallel patterns programmers use. The patterns in the "Structured Parallel Programming" book [73] serve as a proxy to characterize the breadth of our study. There is not a 1-to-1 mapping between our lower-level code-based patterns and their higher-level algorithmic patterns, so we search for those in our benchmarks. RPB uses 14 of 22 mentioned parallel patterns:

- **Present:** fork-join, map, stencil, reduction, scan, recurrence, pack, geometric decomposition, gather, scatter, search, segmentation, category reduction, and workpile.
- **Absent:** pipeline, superscalar sequences, futures, speculative selection, expand, term graph rewriting, branch and bound, and transactions.

Future work could further explore Rust's handling of the latter.

### 7.2 Coverage of irregular parallelism

Irregular parallelism is abundant in PBBS as a tool for high performance, so leaving it unsupported is unacceptable for a language claiming to be "blazingly fast" [101]. The two following static observations support this claim.[4]

**All RPB benchmarks have irregular parallelism.** Table 1 shows the parallel patterns (summarized in Table 3) that each benchmark

[4]Future work can evaluate the contribution of irregular parallelism at run time. However, static measurements better reflect the programmer's experience and fears.

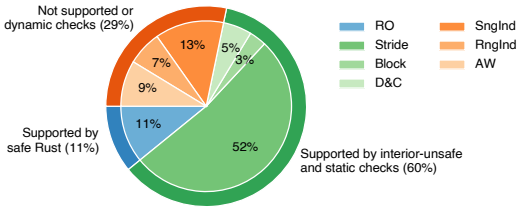



**Figure 3: Distribution of access patterns in RPB.**

**Table 3: Studied patterns and their safety levels.**

| Abbr. | Write pattern | Parallel expression | Fearlessness |
|---|---|---|---|
| **RO** | Read only (AXM) | spawn(Rust) / par_iter(Rayon) | F |
| **Stride** | Striding | par_iter_mut (Rayon) | F |
| **Block** | Blocking | par_chunks_mut (Rayon) | F |
| **D&C** | Divide and Conquer | join (Rayon) | F |
| **SngInd** | Single-valued indirection | par_ind_iter_mut (ours) | C |
| **RngInd** | Ranged indirection | par_ind_chunks_mut (ours) | C |
| **AW** | Arbitrary writes | mix of above | S |

uses. Seven out of 14 have AW, scaring the programmer. Six have SngInd but not AW, forcing the programmer to choose between comfort and performance. sort only has RngInd, so is comfortable to express but not fearless. In short, Rust does not provide fearlessness for any benchmark in RPB.

**29% of accesses in RPB are irregular.** Although irregular parallelism is present, it is necessary to show that it is not a minor part of the algorithm. We statically collect all accesses to shared data structures inside parallel regions, then classify them based on their pattern, which is illustrated in Fig. 3. Although most accesses are instances of regular parallelism, 29% of them are not. Specifically, the programmer is scared when expressing 9% of them (AW), has to choose between comfort and performance for 13% of them (SngInd), and is comfortable when expressing 7% of them (RngInd).

## 7.3 Performance of zero-cost Rust vs. C++

We evaluate the performance of our recommended Rust expressions by comparing a version of RPB with the original PBBS and our C++ MultiQueue. In this version of RPB, we use unsafe operations to implement SngInd and AW to mitigate performance degradation of run-time safety checks. However, we use par_ind_chunks_mut to express RngInd because its overhead is negligible.

**1-thread performance comparison:** PBBS and RPB differ in both language and runtime. We begin evaluation at 1 thread to sidestep differences in parallel runtime implementations such as work stealing. Fig. 4(a) shows execution times of Rust and C++-based benchmarks, normalized to C++[5]. RPB performs close to PBBS in most benchmarks, demonstrating our port's faithfulness. It also suggests that Rayon's static checks of regular parallelism and RngInd's run-time check are indeed static and cheap, respectively. Notably, RPB surprisingly outperforms PBBS in 9 benchmark-input pairs by up to 57% (sa). This may be due to Rust's potential to transfer high-level information to the compiler, such as annotating generated

[5]bw results for C++ are not reliable because they are using *ParlayLib*'s homegrown runtime and not Cilk due to compilation errors. The discrepancy between road and link results for bfs is consistent across additional road network vs power-law graphs.

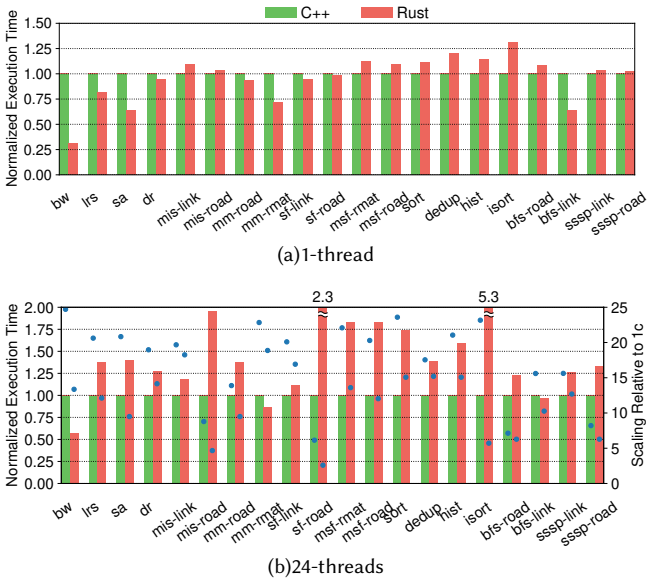



**Figure 4: Execution time of RPB vs. PBBS at 1 and 24 threads. Blue dots (right Y-axis) show scaling relative to 1c.**

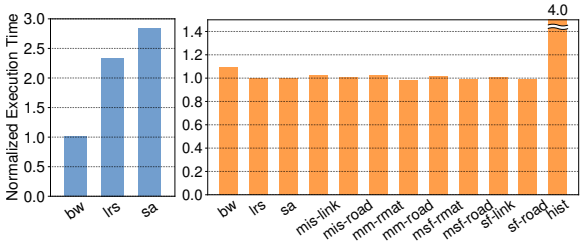
LLVM code with potentially more accurate alias information due to AXM. However, future work is required for detailed characterization.

**24-thread performance comparison:** Fig. 4(b) shows execution time at 24 threads. The blue dots represent scaling relative to the same code at 1 thread, measured by the right Y-axis. Interestingly, RPB scales worse than PBBS on all benchmarks. This can be caused by inefficiencies in our implementations, Rayon's worse runtime management, or OpenCilk's better parallel code optimizations [105, 106]. Our 1-thread evaluation suggests that our implementation is not inefficient. sssp and bfs have worse scalability in Rust but do not use Rayon, suggesting that the language and compiler may be a part of the problem. However, this scaling degradation is worse for the rest of the benchmarks that use Rayon. This suggests that Rayon's runtime management is also worse than Cilk's.

## 7.4 Overheads of replacing unsafe code

In our benchmarks, there are many cases where tasks are algorithmically independent but Rust does not understand that, as shown in Sec. 5.1. We used **unsafe** for those in the previous evaluation because rustc was unconvinced that they are independent. Unsafe code in a code base is the ultimate source of fear for programmers. It can be replaced by pattern-specific interior-unsafe functions with run-time checks or synchronization, reducing the fear to a comfortable level or restricting the set of possible errors, respectively.

**Interior unsafe:** We integrate par_ind_iter_mut into three of seven benchmarks that use SngInd. This function has an expensive run-time check but brings the programmer comfort. Fig. 5(a) shows the execution times of this comfortable version normalized to the previous scary unsafe one. SngInd is a small part of bw, so the overhead is negligible. lrs and sa have not only huge overhead but also worse scaling.




(a) Interior unsafe[6] (b) Unnecessary synchronization

**Figure 5: Overheads of dynamic offset checking for `SngInd` or unnecessary synchronization for `SngInd` and `AW` at 24 threads.**

**Unnecessary Synchronization:** Although unnecessary for tasks with independent access sets, synchronization can also replace unsafe blocks but does not reduce fear. Fig. 5(b) uses synchronization primitives to implement `AW` and `SngInd`. When the benchmarks use atomics (all except `hist`), the overhead is negligible due to not using atomic read-modify-write operations (RMW) and tagging loads and stores with Relaxed ordering, making them almost zero-cost but scary. However, not every type has corresponding atomic support. For example, large structs in `hist` cannot use atomics, requiring `Mutexes` instead and causing a 4× slowdown.

# 8 RELATED WORK

## 8.1 Rust and its type system

There has been a long search for sound and expressive type systems inspired by Milner's observation [77] that "well-typed expressions do not go wrong." Linear type systems restrict resource usage by allowing objects to be used only once [127], which is too restrictive. Ownership-based types annotate each value with its owner and drop values as their owners go out of scope [21, 22]. Region-based memory management deallocates all allocated memory in a region when it ends [124]. These techniques have inspired numerous works that have demonstrated their benefits [16, 40, 91, 128], modified existing popular programming languages [31, 45, 56], and built more powerful type systems [91, 126]. Nevertheless none of these made their way to—nor gained popularity in—the industry.

Rust borrows ideas from these techniques to establish itself as a safe programming language. Prior work has studied the programmer's experience [59, 136, 138]. Crichton conducted a case study to evaluate the usability of `rustc` error messages in guiding programmers to solve their ownership errors [23]. Fulton et al. surveyed developers and drew conclusions about Rust's ease of use and learning curve [41]. While Rust's creators did not formally prove its safety, others formally modeled subsets of Rust to verify the soundness of the type system for those subsets [49, 58, 72, 129]. Rust also allows the use of unsafe Rust that has prompted research into its practical usage [4, 35, 50], vulnerabilities in unsafe blocks [85], mitigation solutions [67, 96], and retaining performance benefits [57].

Prior work also evaluated and improved Rust's interactions with parallelism. Although some referenced proofs consider concurrency and parallelism [58], the use of unsafe Rust, coupled with considering subsets in the proofs, implies that many concurrency bugs remain possible in Rust. Like us, prior work tried to question the fearlessness of Rust but concentrated on evaluating and categorizing bugs found in real-world system software and libraries [89, 135]. However, target applications in these works do not encompass all the types of parallelism we discussed in this work and mainly focus on regular and/or coarse grain thread-level parallelism. Others enhanced Rust's expressiveness to efficiently support specific data structures [133] or increased concurrency safety by adding more guarantees [25], but these do not cover all types of parallelism.

## 8.2 Regular vs irregular parallelism

In regular parallelism, tasks and dependences are known statically, enabling the representation of a safe schedule pre-execution using tools like task dependence graphs [36, 64]. Programming languages and libraries facilitate the expression of parallel tasks in contexts such as data parallelism [8, 26], dataflow parallelism [90, 121], and fork-join parallelism [9, 10]. Compiler analyses identify parallelism in few cases [13, 75, 79, 94, 100], and runtime systems balance workload [10] and coarsen tasks [137]. Given the widespread support for regular parallelism, prior work focuses on its usability [81, 94].

In irregular parallelism, tasks reveal their dependences as they execute, making task dependence graphs unsuitable for static analyses [87]. While launching parallel tasks is straightforward, ensuring a correct ordering between them and their accesses presents challenges. Broadly, three approaches exist for addressing this issue: pessimistic synchronization [117], optimistic speculation [65, 92, 111], and hybrid methods [7, 47, 132]. Among these, prior work on transactional memory (TM) is most related as it appeared to be a silver bullet [84] to ease parallel programming. Like this paper, prior work reported experience in using TM on selected algorithms and real-world applications [42, 62, 69, 108, 139]. Other work in TM usability performed *user* studies in classrooms [84, 97].

Parallel programming pitfalls have been studied extensively, and researchers classified them into categories such as race condition, atomicity violation [39], deadlock, livelock, order violation [68], non-determinism [66], and priority inversion. However, Rust only guarantees to eliminate data races. Prior work aimed to detect data races statically [14, 20, 60, 61, 120] or dynamically [32, 34, 51, 82, 104, 109], or prevent them via formal type systems [5, 16, 37, 38]. Rust takes the last approach; safe Rust code is data-race-free. Yet, freedom from data races does not imply freedom from all errors [39].

# 9 CONCLUSIONS

Easy parallelism should be easy for programmers to express and Rust finally grants this in a mainstream language. Through its ownership-based type system, Rust, coupled with Rayon, detects concurrency bugs at compile time for read-only and regular parallelism. However, the notorious cases of irregular parallelism with indirect and arbitrarily written addresses remain as difficult as ever, requiring synchronization or fragile interior unsafe functions with expensive run-time checks. On one hand, Rust does not detect concurrency errors such as deadlocks, livelocks, atomicity violations, or order violations. On the other hand, the overheads of run-time checks hinders the benefits of parallelism. Consequently, the programmer must choose between fear or overheads. Rust's fearless concurrency is an exciting advance for the easy cases, but remains over hyped when parallel programming gets hard.

## ACKNOWLEDGMENTS

We sincerely thank Isidor R. Brkić, Leo X. Han, Aster Plotnik, Matthew Sweet, and the anonymous reviewers for their helpful feedback, and Guowei Zhang for helpful early discussions. This work was supported in part by the Digital Research Alliance of Canada, the University of Toronto, NSERC, an Ontario QEII-GSST, and an Ontario Bell Graduate Scholarship.

## REFERENCES

[1] 2006. 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/~challenge9, archived at https://perma.cc/5KYT-YM36. (2006).

[2] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), v2. In *Proc. PPoPP*. DOI: 10.1145/3503221.3508422.

[3] Arvind, David August, Keshav Pingali, Derek Chiou, Resit Sendag, and J Yi Joshua. 2010. Programming multicores: do applications programmers need to write explicitly parallel programs? *IEEE Micro*, 3. DOI: 10.1109/MM.2010.54.

[4] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe Rust? In *Proc. OOPSLA*. DOI: 10.1145/3428204.

[5] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. 2000. Guava: a dialect of java without data races. In *Proc. OOPSLA*. DOI: 10.1145/353171.353197.

[6] Jeff Barr. 2018. Firecracker - lightweight virtualization for serverless computing. AWS News Blog. (Nov. 2018). https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

[7] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proc. PPoPP*. DOI: 10.1145/2370036.2145840.

[8] 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*. DOI: 10.1145/173284.155343.

[9] 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*. DOI: 10.1145/209936.209958.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46, 5. DOI: 10.1145/324133.324234.

[11] 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*. DOI: 10.1145/1640089.1640097.

[12] Hans-J Boehm. 2011. How to miscompile programs with" benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*. https://www.usenix.org/legacy/events/hotpar11/tech/final_files/Boehm.pdf.

[13] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral program optimization system. In *Proc. PLDI*. DOI: 10.1145/1375581.1375595.

[14] Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. 2020. Llov: a fast static data-race checker for OpenMP programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17, 4. DOI: 10.1145/3418597.

[15] Mara Bos. 2023. *Rust Atomics and Locks*. O'Reilly Media, Inc.

[16] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. OOPSLA*. DOI: 10.1145/582419.582440.

[17] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proc. SIAM International Conference on Data Mining (SDM)*. DOI: 10.1137/1.9781611972740.43.

[18] Samartha Chandrashekar. 2020. Announcing the general availability of Bottlerocket, an open source Linux distribution built to run containers. AWS Open Source Blog. (Aug. 2020). https://aws.amazon.com/blogs/opensource/announcing-the-general-availability-of-bottlerocket-an-open-source-linux-distribution-purpose-built-to-run-containers/.

[19] 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*. DOI: 10.1145/1094811.1094852.

[20] Prasanth Chatarasi, Jun Shirako, Martin Kong, and Vivek Sarkar. 2016. An extended polyhedral model for spmd programs and its use in static data race detection. In *Proc. of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. DOI: 10.1007/978-3-319-52709-3_10.

[21] David G Clarke, James Noble, and John M Potter. 2001. Simple ownership types for object containment. In *Proc. ECOOP*. DOI: 10.1007/3-540-45337-7_4.

[22] David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proc. OOPSLA*. DOI: 10.1145/286936.286947.

[23] Will Crichton. 2020. The usability of ownership. In *Proc. of the Workshop on Human Aspects of Types and Reasoning Assistants* (HATRA @ SPLASH). DOI: 10.48550/arXiv.2011.06171.

[24] The Crossbeam Team. 2022. Crossbeam - tools for concurrent programming in Rust. https://github.com/crossbeam-rs/crossbeam. (2022).

[25] Zak Cutner, Nobuko Yoshida, and Martin Vassor. 2022. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In *Proc. PPoPP*. DOI: 10.1145/3503221.3508404.

[26] Leonardo Dagum and Ramesh Menon. 1998. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5, 1. DOI: 10.1109/99.660313.

[27] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4, POPL. DOI: 10.1145/3371102.

[28] The Dashmap Team. 2023. Dashmap - blazingly fast concurrent map in Rust. execution. https://github.com/xacrimon/dashmap. (2023).

[29] Laxman Dhulipala, Guy Blelloch, and Julian Shun. 2017. Julienne: a framework for parallel graph algorithms using work-efficient bucketing. In *Proc. SPAA*. DOI: 10.1145/3087556.3087580.

[30] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)*, 8, 1. DOI: 10.1145/3434393.

[31] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. 2003. Memory safety without runtime checks or garbage collection. In *Proc. LCTES*. DOI: 10.1145/780731.780743.

[32] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. In *Proc. ACM/ONR workshop on Parallel and distributed debugging (PADD)*. DOI: 10.1145/122759.122767.

[33] The Dropbox Capture Team. 2021. Why we built a custom Rust library for Capture. Dropbox.tech. (Sept. 2021). https://dropbox.tech/application/why-we-built-a-custom-rust-library-for-capture/.

[34] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *Proc. OOPSLA*. DOI: 10.1145/2384616.2384650.

[35] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust used safely by software developers? In *Proc. ICSE*. DOI: 10.1145/3377811.3380413.

[36] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9, 3. DOI: 10.1145/24039.24041.

[37] Cormac Flanagan and Martin Abadi. 1999. Object types against races. In *Proc. International Conference on Concurrency Theory*. DOI: 10.1007/3-540-48320-9_21.

[38] Cormac Flanagan and Stephen N Freund. 2000. Type-based race detection for java. In *Proc. PLDI*. DOI: 10.1145/349299.349328.

[39] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proc. PLDI*. DOI: 10.1145/781131.781169.

[40] Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear regions are all you need. In *Proc. European Symp. on Programming*. DOI: 10.1007/11693024_2.

[41] Kelsey R Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L Mazurek. 2021. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Proc. Symposium on Usable Privacy and Security (SOUPS)*. https://www.usenix.org/system/files/soups2021-fulton.pdf.

[42] Vladimir Gajinov, Ferad Zyulkyarov, Osman S. Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. 2009. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proc. ICS'09*. DOI: 10.1145/1542275.1542298.

[43] David K. Gifford and John M. Lucassen. 1986. Integrating functional and imperative programming. In *Proc. ACM Conference on LISP and Functional Programming* (LFP). DOI: 10.1145/319838.319848.

[44] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proc. OOPSLA*. DOI: 10.1145/2384616.2384619.

[45] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in cyclone. In *Proc. PLDI*. DOI: 10.1145/512529.512563.

[46] William Hasenplaugh, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2014. Ordering heuristics for parallel graph coloring. In *Proc. SPAA*. DOI: 10.1145/2612669.2612697.

[47] Muhammad Amber Hassaan, Donald Nguyen, and Keshav Pingali. 2015. Kinetic dependence graphs. In *Proc. ASPLOS-XX*. DOI: 10.1145/2694344.2694363.

[48] Dave Herman. 2016. Shipping Rust in Firefox. Mozilla Hacks. (July 2016). https://hacks.mozilla.org/2016/07/shipping-rust-in-firefox/.

[49] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6, ICFP. DOI: 10.1145/3547647.

[50] Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2022. Poster: unsafe Rust–conscious choice or spiky shortcut? In *Proc. IEEE Symposium on Security and Privacy (SP)*. https://www.ieee-security.org/TC/SP2022/downloads/SP22-posters/sp22-posters-52.pdf.

[51] Robert Hood, Ken Kennedy, and John Mellor-Crummey. 1990. Parallel program debugging with on-the-fly anomaly detection. In *Proc. SC90*. DOI: 10.1109/SUPERC.1990.130004.

[52] Jesse Howarth. 2020. Why Discord is switching from Go to Rust. Discord Blog. (Feb. 2020). https://discord.com/blog/why-discord-is-switching-from-go-to-rust.

[53] 2020. Programming Languages—C++. Standard. International Organization for Standardization, Geneva, CH.

[54] Dana Jansens. 2023. Supporting the use of Rust in the Chromium project. Google Security Blog. (Jan. 2023). https://security.googleblog.com/2023/01/supporting-use-of-rust-in-chromium.html.

[55] Ján Jergus. 2019. Hhvm 4.22.0. HHVM Blog. (Sept. 2019). https://hhvm.com/blog/2019/09/09/hhvm-4.22.0.html.

[56] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of C. In *Proc. USENIX ATC*. https://www.usenix.org/legacy/event/usenix02/full_papers/jim/jim_html/.

[57] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, 4, POPL. DOI: 10.1145/3371109.

[58] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2, POPL. DOI: 10.1145/3158154.

[59] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Communications of the ACM*, 64, 4. DOI: 10.1145/3418295.

[60] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static data race detection for concurrent programs with asynchronous calls. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. DOI: 10.1145/1595696.1595701.

[61] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *Proc. of the International Conference on Computer Aided Verification (CAV)*. DOI: 10.1007/978-3-540-73368-3_26.

[62] Seunghwa Kang and David A. Bader. 2009. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In *Proc. PPoPP*. DOI: 10.1145/1504176.1504182.

[63] Steve Klabnik and Carol Nichols. 2022. *The Rust Programming Language*. (2nd ed.). No Starch Press.

[64] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. 1981. Dependence graphs and compiler optimizations. In *Proc. POPL*. DOI: 10.1145/567532.567555.

[65] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proc. PLDI*. DOI: 10.1145/1250734.1250759.

[66] Edward A. Lee. 2006. The problem with threads. *Computer*, 39, 5. DOI: 10.1109/MC.2006.180.

[67] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe Rust programs with xRust. In *Proc. ICSE*. DOI: 10.1145/3377811.3380325.

[68] 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*. DOI: 10.1145/1346281.1346323.

[69] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. 2010. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proc. EuroSys*. DOI: 10.1145/1755913.1755919.

[70] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. In *Proc. ACM SIGAda Annual Conference on High Integrity Language Technology* (HILT). DOI: 10.1145/2663171.2663188.

[71] Niko Matsakis. 2015. Rayon: data parallelism in Rust. (Dec. 2015). https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[72] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. Rusthorn: chc-based verification for Rust programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43, 4. DOI: 10.1145/3462205.

[73] Michael McCool, James Reinders, and Arch Robison. 2012. *Structured parallel programming: patterns for efficient computation*. Elsevier.

[74] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web-analyzed on different aggregation levels. *The Journal of Web Science*, 1, 1. DOI: 10.1561/106.00000003.

[75] Samuel P. Midkiff. 2012. Automatic parallelization: an overview of fundamental compiler techniques. *Synthesis Lectures on Computer Architecture*.

[76] Mae Milano, Joshua Turcotti, and Andrew C Myers. 2022. A flexible type system for fearless concurrency. In *Proc. PLDI*. DOI: 10.1145/3519939.3523443.

[77] Robin Milner. 1978. A theory of type polymorphism in programming. *Journal of computer and system sciences (JCSS)*, 17, 3. DOI: 10.1016/0022-0000(78)90014-4.

[78] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A type system for borrowing permissions. In *Proc. POPL*. DOI: 10.1145/2103656.2103722.

[79] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35, 10. DOI: 10.1109/TCAD.2015.2513673.

[80] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proc. SOSP-24*. DOI: 10.1145/2517349.2522739.

[81] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proc. PLDI*. DOI: 10.1145/3385412.3385974.

[82] Robert O'callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proc. PPoPP*. DOI: 10.1145/781498.781528.

[83] Steven Pack. 2018. Serverless Rust with cloudflare workers. The Cloudflare Blog. (Oct. 2018). https://blog.cloudflare.com/cloudflare-workers-as-a-serverless-rust-platform/.

[84] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A study of transactional memory vs. locks in practice. In *Proc. SPAA*. DOI: 10.1145/1989493.1989500.

[85] Michalis Papaevripides and Elias Athanasopoulos. 2021. Exploiting mixed binaries. *ACM Transactions on Privacy and Security (TOPS)*, 24, 2. DOI: 10.1145/3418898.

[86] The parking_lot team. [n. d.] Parking_lot - compact and efficient synchronization primitives for Rust. (). https://github.com/Amanieu/parking%5C_lot.

[87] 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*. DOI: 10.1145/1993316.1993501.

[88] Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. 2022. Multi-queues can be state-of-the-art priority schedulers. In *Proc. PPoPP*. DOI: 10.1145/3503221.3508432.

[89] 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*. DOI: 10.1145/3385412.3386036.

[90] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proc. PLDI*. DOI: 10.1145/2491956.2462176.

[91] Ohad Rau, Caleb Voss, and Vivek Sarkar. 2021. Linear promises: towards safer concurrent programming. In *Proc. ECOOP*. DOI: 10.4230/LIPIcs.ECOOP.2021.13.

[92] Lawrence Rauchwerger and David Padua. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proc. PLDI*. DOI: 10.1145/207110.207148.

[93] James Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly & Associates.

[94] G. Ren, P. Wu, and D. Padua. 2005. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *Proc. IPDPS*. DOI: 10.1109/IPDPS.2005.94.

[95] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Multiqueues: simple relaxed concurrent priority queues. In *Proc. SPAA*. DOI: 10.1145/2755573.2755616.

[96] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping safe Rust safe with galeed. In *Proc. Annual Computer Security Applications Conference (ACSAC)*. DOI: 10.1145/3485832.3485903.

[97] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier? In *Proc. PPoPP*. DOI: 10.1145/1693453.1693462.

[98] [n. d.] RPB github repository. (). https://github.com/mcj-group/rpb.

[99] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 2014. Transactionalizing legacy code: an experience report using gcc and memcached. In *Proc. ASPLOS-XIX*. DOI: 10.1145/2541940.2541960.

[100] Radu Rugina and Martin Rinard. 1999. Automatic parallelization of divide and conquer algorithms. In *Proc. PPoPP*. DOI: 10.1145/301104.301111.

[101] The Rust Team. 2023. Rust. https://rust-lang.org. (2023). Retrieved Mar. 23, 2023 from.

[102] The Rust Team. 2022. The Rustonomicon: the dark arts of Unsafe Rust. (2022). Retrieved Nov. 8, 2022 from https://doc.rust-lang.org/nomicon/.

[103] Piotr Sarna. 2021. Scylladb developer hackathon: Rust driver. ScyllaDB Blog. (Feb. 2021). https://www.scylladb.com/2021/02/17/scylla-developer-hackathon-rust-driver.

[104] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15, 4. DOI: 10.1145/265924.265927.

[105] Tao B. Schardl and I-Ting Angelina Lee. 2023. Opencilk: a modular and extensible software infrastructure for fast task-parallel code. In *Proc. PPoPP*. DOI: 10.1145/3572848.3577509.

[106] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: embedding fork-join parallelism into LLVM's intermediate representation. In *Proc. PPoPP*.

[107] Michael L Scott. 2013. *Shared-memory synchronization*. Morgan & Claypool Publishers.

[108] Michael L. Scott, Michael F. Spear, Luke Dalessandro, and Virendra J. Marathe. 2007. Delaunay triangulation with transactions and barriers. In *Proc. IISWC*. DOI: 10.1109/IISWC.2007.4362186.

[109] Konstantin Serebryany and Timur Iskhodzhanov. 2009. Threadsanitizer: data race detection in practice. In *Proc. Workshop on Binary Instrumentation and Applications (WBIA)*. DOI: 10.1145/1791194.1791203.

[110] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10, 2. DOI: 10.1145/42190.42277.

[111] Nir Shavit and Dan Touitou. [n. d.] Software transactional memory. In DOI: 10.1145/224964.224987.

[112] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing contention through priority updates. In *Proc. SPAA*. DOI: 10.1145/2486159.2486189.

[113] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *Proc. SPAA*. DOI: 10.1145/2312005.2312018.

[114] The Spin team. [n. d.] Spin - spin-based synchronization primitives. (). https://github.com/mvdnes/spin-rs.

[115] The Stack Overflow Team. 2022. Stack Overflow developer survey. (2022). https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages.

[116] The Stack Overflow Team. 2023. Stack Overflow developer survey. (2023). https://survey.stackoverflow.co/2023/#technology-admired-and-desired.

[117] Herb Sutter. 2008. Lock-free code: a false sense of security. *Dr. Dobb's Journal*, 33, 9.

[118] The System76 Team. 2022. 2022 at System76: a year in review. The Blog of System76. (Dec. 2022). https://blog.system76.com/post/2022-at-system76-a-year-in-review.

[119] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. [n. d.] Prodigy: improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In DOI: 10.1109/HPCA51647.2021.00061.

[120] Tachio Terauchi. 2008. Checking race freedom via linear programming. In *Proc. PLDI*. DOI: 10.1145/1379022.1375583.

[121] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Proc. of the International Conference on Compiler Construction (CC)*. DOI: 10.1007/3-540-45937-5_14.

[122] Neil Thompson. 2017. The economic impact of Moore's law: evidence from when it faltered. *SSRN*, (Jan. 2017). DOI: 10.2139/ssrn.2899115.

[123] The Threadpool Team. 2022. Threadpool - a very simple thread pool for parallel task execution. (2022). https://github.com/rust-threadpool/rust-threadpool.

[124] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and computation*, 132, 2. DOI: 10.1006/inco.1996.2613.

[125] The Tokio Team. 2022. Tokio - an asynchronous Rust runtime. https://tokio.rs/. (2022).

[126] Caleb Voss and Vivek Sarkar. 2021. An ownership policy and deadlock detector for promises. In *Proc. PPoPP*. DOI: 10.1145/3437801.3441616.

[127] Philip Wadler. 1990. Linear types can change the world! In *Programming Concepts and Methods*.

[128] David Walker and Kevin Watkins. 2001. On regions and linear types (extended abstract). In *Proc. ICFP*. DOI: 10.1145/507635.507658.

[129] Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. 2019. Oxide: the essence of Rust. *arXiv preprint arXiv:1903.00982*.

[130] Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. 2022. Parallel block-delayed sequences. In *Proc. PPoPP*. DOI: 10.1145/3503221.3508434.

[131] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable data-driven PageRank: algorithms, system issues, and lessons learned. In *Proc. EuroPar*. DOI: 10.1007/978-3-662-48096-0_34.

[132] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and S Hiranandan. 1995. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44, 6.

[133] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: separating permissions from data in Rust. *Proceedings of the ACM on Programming Languages*, 5, ICFP. DOI: 10.1145/3473597.

[134] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs? In *Proc. of the SIGSOFT-/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13)*. ACM. DOI: 10.1145/2025113.2025121.

[135] Zeming Yu, Linhai Song, and Yiying Zhang. 2019. Fearless concurrency? understanding concurrent programming safety in real-world Rust software. *arXiv preprint arXiv:1902.01906*.

[136] Anna Zeng and Will Crichton. 2018. Identifying barriers to adoption for Rust through online discourse. In *Proc. of the Workshop on the Intersection of HCI and PL* (PLATEAU @ SPLASH). DOI: 10.48550/arXiv.1901.01001.

[137] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. 2010. Reducing task creation and termination overhead in explicitly parallel programs. In *Proc. PACT-19*.

[138] Shuofei Zhu, Ziyi Zhang, Boqin Qin, Aiping Xiong, and Linhai Song. 2022. Learning and programming challenges of Rust: a mixed-methods study. In *Proc. ICSE*. DOI: 10.1145/3510003.3510164.

[139] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. 2009. Atomic quake: using transactional memory in an interactive multiplayer game server. In *Proc. PPoPP*. DOI: 10.1145/1504176.1504183.


# A JUSTIFICATION FOR THE ADOPTION OF RAYON FOR TASK-BASED PARALLELISM

This appendix provides the rationale for using Rayon as the task-parallel framework. We use a simple microbenchmark that updates every element of a vector with the hash of the element's index. We adapt the hash function from PBBS [2, 113], shown in Listing 10. We evaluate different techniques to parallelize this process, and compare those to a sequential version in Listing 11. Listing 12 shows the Rayon version, which changes the sequential code with net zero lines, swapping `iter_mut` with `par_iter_mut`. Sec. A.1 attempts to parallelize using only the Rust standard library.

## A.1 Parallelization with Rust standard library

To imitate the simplicity of the sequential and Rayon versions, Listing 13 naively launches a thread per task. However, due to the small size of the tasks in this example, the overheads of launching tasks outweighs the gains of parallelism. This version is likely slower than the sequential one. Moreover, for large inputs, this code launches a large number of threads, filling the stack and leading to program termination.

```
fn task(e: &mut usize) {
  let mut v = e.overflowing_mul(3_935_559_000_370_003_845).0;
  v = v.overflowing_add(2_691_343_689_449_507_681).0;
  v ^= v >> 21;
  v ^= v << 37;
  v ^= v >> 4;
  v = v.overflowing_mul(4_768_777_513_237_032_717).0;
  v ^= v << 20;
  v ^= v >> 41;
  v ^= v << 5;
  *e = v;
}
```

**Listing 10: A hash function form PBBS as our task.**

```rust
fn serial_hash(v: &mut [usize]) {
  v.iter_mut()
   .for_each(task);
}
```

**Listing 11: Sequentially replacing v[i] with its hash.**

```rust
use rayon::prelude::*;
fn par_hash(v: &mut [usize]) {
  v.par_iter_mut()
   .for_each(task);
}
```

**Listing 12: Element-wise hashing of v via Rayon.**

```rust
use std::thread::scope;
fn par_hash_1(v: &mut [usize]) {
  scope(|s| {
    let mut threads = Vec::new();
    v.iter_mut().for_each(|vi| {
      threads.push(s.spawn(|| task(vi))); });
    threads.into_iter().for_each(|t| t.join().unwrap());
  });
}
```

**Listing 13: Launching a thread per task (iteration).**

```rust
use std::thread::{scope, available_parallelism};
fn par_hash_2(v: &mut [usize]) {
  let num_threads = available_parallelism() as usize;
  let elements_per_thread = ((v.len()+num_threads-1)/num_threads)+1;
  let mut chunks = v.chunks_mut(elements_per_thread);
  scope(|s| {
    let mut threads = Vec::new();
    for _ in 0..num_threads {
      let chunk = chunk.next().unwrap();
      threads.push(s.spawn(|| v.iter_mut()
                               .for_each(task))));
    }
    threads.into_iter().for_each(|t| t.join().unwrap());
  });
}
```

**Listing 14: Launching a thread per core and splitting v.**

To drive down the overheads, Listing 14 coarsens tasks. It slices the vector into n chunks, where n is equal to the number of cores, and launches a thread to process each chunk. This implementation is prone to load imbalance between threads—all threads but the straggler remain idle until the last thread finishes.

To mitigate the load imbalance issue, Listing 15 slices the vector into multiple chunks, much more than the number of available cores, and puts these *jobs* in a queue. Then worker threads get jobs from this queue and execute them until none remain.

As more performance optimizations are added, the complexity and lines of code (LOC) increases. The Rayon version already provides these and other performance techniques, such as dynamic scheduling through work stealing, yet requires the fewest LOC through its elegant interface.

```rust
use std::thread::{scope, available_parallelism};
use std::sync::Mutex;
fn par_hash_3(v: &mut [usize]) {
  let num_threads = available_parallelism() as usize;
  let elements_per_job = 10000;
  let jobs = v.chunks_mut(elements_per_job);
  let jobs = Mutex::new(jobs);
  scope(|s| {
    let mut threads = Vec::new();
    for _ in 0..num_threads {
      threads.push(s.spawn(|| {
        loop {
          let mut jobs = jobs.lock().unwrap();// lock
          let job = jobs.next();              // get a job
          std::mem::drop(jobs);               // unlock
          if let Some(job) = job {            // if job existed,
            v.iter_mut()                      // process serially
             .for_each(task);
          } else { break; }                   // if not, exit
        }));
    }
    threads.into_iter().for_each(|t| t.join().unwrap());
  });
}
```

**Listing 15: A software runtime using a job queue of v's slices.**

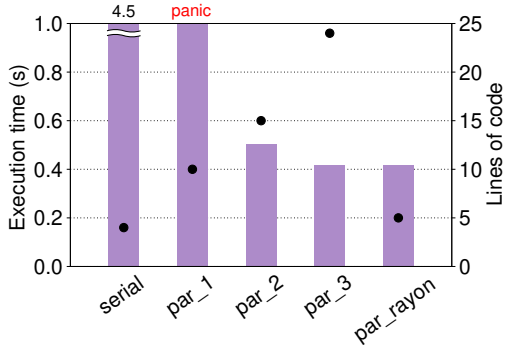



**Figure 6: Run times of different implementations.**

## A.2 Evaluation

Fig. 6 illustrates the run times for each of these implementations. Rayon performs the best while having least amount of code. To measure these run times, we follow the same methodology as Sec. 7.1, except that we target a 16-core machine and execute each benchmark 1000 times. The vector we use in this microbenchmark has $10^9$ elements. In our example, tasks have almost equal lengths; we expect even more favourable results for Rayon when tasks have varying lengths, amplifying the benefits of dynamic scheduling.