A Verified Thread-Safe Array in Rust

Sasha Pak
Fabian Muehlboeck
Alex Potanin

Abstract

Rust's ownership type system limits the flexibility of safe parallel array manipulation. In particular, safe code is confined to partitioning arrays into disjoint, contiguous slices. We present a statically verified, thread-safe array API that enables arbitrary parallel access patterns. This unlocks more expressive parallelism in safe Rust without relying on run-time checks or unsafe code.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Formal software verification**; *Software performance*.

Keywords: Rust, Verus, program verification, memory safety

ACM Reference Format:
Sasha Pak, Fabian Muehlboeck, and Alex Potanin. 2026. A Verified Thread-Safe Array in Rust. In *Proceedings of International Workshop on Aliasing, Capabilities and Ownership (IWACO) at SPLASH (IWACO'25).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

C and C++ have been among the most popular systems programming languages for a long time. With Rust gaining popularity, software engineers consider starting new projects in Rust or translating existing codebases. In 2020 Linux began to port kernel drivers to Rust, although currently only 0.1% of the repository is Rust code [1]. The advantage of Rust is that its ownership type system makes any software memory safe by design: no invalid pointers, no use after free, and no data races. [1]

However, translating code from C or C++ to Rust remains challenging due to their radically different type systems. Recent attempts include porting the **dav1d** [2] media decoder to Rust [5] to reduce bugs without sacrificing performance. The resulting project, **rav1d**, faced difficulties when sharing mutable arrays across threads. Rust's standard approach – splitting arrays into non-overlapping slices – does not suit **dav1d**'s complex access patterns. Therefore, the team resorted to unsafe buffer wrappers with run-time checks disabled to improve performance. Can we retain both performance and thread safety when manipulating shared buffers in parallel? If standard Rust falls short, we can explore formal verification frameworks to statically prove memory safety in such scenarios.

To investigate this, we examine the capabilities of a modern Rust verification framework in the context of parallel array processing and make the following contributions:

- A statically checked, thread-safe array in verified Rust (subsection 3.1);
- A thread-safe parallel merge sort as a case study demonstrating the applicability of modern Rust verifiers to standard parallel algorithms (subsection 3.2);

Finally, we discuss future directions for Rust verifiers to produce high-performance programs with lower verification overhead (subsection 4.2).

2 Background

Rust is a unique systems programming language that leverages a powerful type system to guarantee memory safety at compile time. Its core is forged from affine types and the concept of borrowing [13]. A variable can either own a value, hold a shared reference to it, or hold a unique reference that allows mutation:

```
fn f(owns: i32, reads: &i32, writes: &mut i32) { ... }
```

These principles protect from unrestricted aliasing and make concurrency fearless:

```
let a = vec![0, 1, 2, 3];
rayon::join( || { read(&a); },
             || { read(&a);
                  // write(&mut a); cannot modify
                });
```

Here we run two closures in parallel using the concurrency library **rayon**. Both closures can read the vector but cannot modify it, as Rust understands that it is being shared across threads. In many cases, this protection mechanism prevents various memory bugs and data races. However, it is inevitably incomplete. In the following example, we try to modify the disjoint elements of a list in parallel using parallel iteration `par_iter()`. Rust does not infer that mutated

[1] With the exception of code explicitly marked `unsafe`, where the programmer takes responsibility for upholding memory safety manually.

IWACO'25, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

locations are disjoint and conservatively rejects the program from compilation.

```
fn f(a: &mut [i32], disjoint_inds: &HashSet<usize>) {
    disjoint_inds.par_iter().for_each(|&i| a[i] += 1);
    // cannot mutate captured variable     ^^
}
```

This program illustrates the mutation of the shared data that does not follow a simple structure. **rav1d** team faced similar patterns. In an attempt to support such a challenging scenario, we can resort to Verus [11], *the only unbounded verifier that targets surface-level Rust and supports concurrency.*

Verus has `PCell`, a wrapper around a value that separates permissions from data similar to an earlier work [16].

```
let (cell, Tracked(mut perm)) = PCell::new(0);
cell.replace(Tracked(&mut perm), 66);
let value = *cell.borrow(Tracked(&perm));
assert(value = 66);
```

You can ignore the `Tracked` wrapper – it just signals to Verus that the inner value is part of the proof and should be erased after compilation. In this example, `PCell::new` takes a value and returns two things: a cell that holds the data and a permission object that controls access to it. Holder of `&mut perm` can mutate `cell`. Holder of `&perm` can read from `cell`. Verus carefully tracks the correspondence between `cell` and `perm` to ensure that no other permission can be used in its place. Compared to standard Rust, the advantage is that `cell` can be safely shared between threads, while `perm` can be passed from thread to thread to transfer unique access. This wrapper is strictly more flexible than a Rust reference.

Now that we have enough instruments, we will build a thread-safe array API in Rust and use it to write a parallel merge sort.

3 Approach

We aim to support arbitrary parallel array access without data races or run-time checks. To achieve this, it suffices to ensure that threads operate on disjoint subsets of the same buffer. These subsets need not be contiguous and can consist of arbitrary sets of indices.

3.1 A Thread-safe array

We create a list of Verus cells together with a list of permissions. Each permission will govern the access to a cell at a certain index. The list of cells is stored inside the struct `Array`. Permissions are stored inside a map from an index to a permission object.

```
struct Array<T> { ptrs: Vec<PCell<T>> }
type Perms<T> = Tracked<Map<usize, PointsTo<T>>>;
```

In all of our Verus examples, we will omit pre- and post-conditions of the defined functions. The secret sauce that holds everything together is types, visible in the function signature. Also, we sometimes remove the `Tracked` wrapper to save space.

```
fn new(data: Vec<T>) -> (res: (Self, Perms<T>));
fn read(&self, i: usize, perms: &Perms<T>) -> &T;
fn replace(&self, i: usize, x: T, perms: &mut Perms<T>)
    -> T;
```

Array API consists of three methods: `new`, `read` and `replace`. `new` takes a vector, transfers its elements into an array, and returns it together with a map of permissions. Importantly, both `read` and `replace` take the array by a shared reference. This makes it possible to work with the same array in different threads. `read` and `replace` borrow permissions immutably and mutably, respectively.

Now that we can read from and write to an array, we will try to work with it in parallel. Normally, one would have to use `split_at_mut` Rust function to get two views over the disjoint halves of the array, but this time we do not need it. Compared to previous examples that used `rayon`, we stick to manual thread spawning and atomic reference counters (`Arc`) because of limited Verus support for concurrency. Yet we are able to modify separate sections of the same array in parallel.

```
let (a, Tracked(mut perms)) = Array::new(vec![0, 1]);
let a = Arc::new(a);
let (a_r1, a_r2) = (Arc::clone(&a), Arc::clone(&a));
let Tracked(mut perms1) = ... // move perms[1] to perm1
let t1 = vstd::thread::spawn(move ||
    (*a_r1).replace(0, 66, Tracked(&mut perms)));
let t2 = vstd::thread::spawn(move ||
    (*a_r2).replace(1, 77, Tracked(&mut perms1)));
```

Such a program is possible because we split the permission map into two parts: `perms` and `perms1`, containing permissions associated with `a[0]` and `a[1]`, respectively. The array itself is passed to threads via shared references (`Arc`s in this case). Unlike a regular write to a vector, which requires exclusive access, the replace function only needs exclusive access to a permission map, while the array itself can be shared.

There are two nice things about this implementation. First, parallel reads and writes do not have extra run-time checks. Because `PCell<T>` is a zero-cost abstraction over `T`, the array has the same memory representation as `Vec<T>`. The Permissions map is a ghost value that lives only during compilation and is erased from the program before its execution. Second, everything is built on top of the `PCell` interface and does not use other unsafe code. Apart from the Verus verifier itself, the array implementation does not rely on any additional trusted code or assumptions.

3.2 Case Study: Parallel Merge Sort

A good way to test the designed array would be to try integrating it into **rav1d** in place of unsafe wrappers. At the same time, it would require studying the large code base and instrumenting the necessary parts of it with Verus. Since Verus, as most other verifiers, does not support full Rust; it would result in many additional challenges, orthogonal to the project goal. Thus, we leave this task for future work and

focus instead on an easier and more isolated program that also uses arrays extensively: a parallel merge sort.

Sequential merge sort is a recursive sorting algorithm that sorts two halves of the list and then merges them to produce a sorted sequence. This algorithm can be made parallel with little effort, as we can independently sort the list's halves.

Using the described thread-safe array, we implemented a parallel merge sort. It is automatically thread-safe, as it operates on our safe array abstraction. It consists of 261 lines of code, 120 lines of which are annotations, proofs, and ghost variables. Below is the signature of the sorting function.

```
fn _merge_sort_parallel(
    arr: Arc<Array<i32>>,
    mut lo: usize, hi: usize,
    Tracked(perms): Tracked<&mut Region<i32>>,
    buf: Arc<Array<i32>>,
    mut buf_lo: usize,
    Tracked(buf_perms): Tracked<&mut Region<i32>>,
    threshold: usize
) -> (ret: Result<(), ()>)
```

`arr` is the input array, and `buf` is the reusable helper array to temporarily store the merged result. `lo`, `hi`, and `buf_lo` are indexes. `perms` and `buf_perms` are permissions to access both the input and the buffer. `threshold` specifies the minimal array length to be sorted in parallel. We introduce the type `Region`, which is a wrapper around a permission set with two additional values: `lo` and `hi`. A region represents a permission map where indices form a contiguous slice within a corresponding array. This simplifies proofs for merge sort and potentially other algorithms that operate on contiguous array ranges. Finally, the return value is of type `Result`, signalling an error if thread spawning or joining failed.

3.3 Observations

The implementation and verification processes were not straightforward and presented several challenges. Before discussing these difficulties and nuances, we highlight one key advantage of Rust and Verus: the memory safety of the program does not depend on the correctness of its specification. Thanks to the extended safe subset of Rust on which Verus operates, any program that verifies is guaranteed to be memory-safe.

Mutable references. Verus currently has limited support for mutable references, a known challenge that can be addressed with techniques such as prophecy variables [6] or backward functions [8]. As a result, we could not use slices (`&mut [i32]`) or functions like `split_at()` that return mutable slices, which would simplify the verification of merge sort and avoid the need for `PCells` entirely.

Specification burden. Many post-conditions are written merely to state that parts of the state or ghost state remain unchanged. While Verus already leverages the immutability of shared references, mutable references assume that all fields may be mutated. It would be beneficial to have either immutable ghost types or other mechanisms that allow specifying only the modified parts of a data structure, without having to explicitly list all unchanged fields.

Thread::scope. Verus currently supports only the standard `thread::spawn` function, which does not allow closures to capture references. As a result, we used `Arc`, which incurs some overhead. Adding support for `thread::scope` in Verus would address this limitation.

Vector indexing. Although Verus statically checks all vector accesses, the implementation still includes Rust's runtime bounds checks. To improve performance, we used the unchecked variant `Vec::get_unchecked`, which must be marked as trusted since Verus does not verify its unsafe implementation.

Specialised pointers and cells. Verus extends Rust with primitives like `PCell` and `PPtr`, which are convenient to use. However, manually wrapping elements into `PCells` incurs linear overhead. A potential improvement would be to allow casting a list of values to a list of `PCells`, avoiding manual data transfer.

4 Evaluation

We managed to maintain memory safety thanks to `PCell` primitive provided by Verus verifier. Our second goal was to maintain performance. We did not rewrite **rav1d** with the new array, so we can not evaluate it within their project. Instead, we compare parallel merge sorts implemented with and without our thread-safe abstraction. To make the comparison fair, we minimised the differences between versions. Implementations mostly diverge in the way they work with the array. Verus version simply shares one array between threads, while the standard safe Rust API requires `split_mut_at`:

```
fn split_at_mut(self: &mut [T], mid: usize)
    -> (&mut [T], &mut [T]);
```

Any efficient implementation of a parallel merge sort that does not use unsafe code has to use this function. It is safe because it not only checks that `mid` is smaller than the array length, but also statically pauses access to the input slice while returned halves are being used. We will call this version *Slices*.

The final alternative we consider is similar to *Slices*, but uses unchecked array indexing and unsafe unchecked versions of `split_at_mut`, which we dub *Slices Unchecked*. In summary, different versions have different levels of safety: *Verus* is checked at compile time, *Slices* is checked at run time, *Slices Unchecked* is not checked at all.

4.1 Setup

The experiments were conducted on a system with 60 x86-64 CPU cores, one thread per core, running at 2.1 GHz. The node was allocated 32 GiB of memory and ran Ubuntu 20.04.6 LTS 5.4.0-208-generic. The benchmarks were compiled with Rust 1.88. Array sizes ranged from 10,000 to 1,000,000 elements in

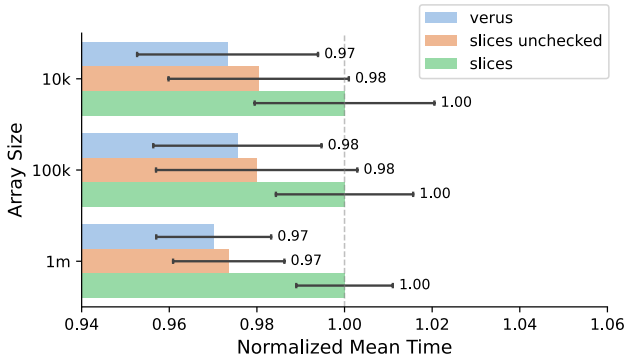

Figure 1. Comparison of sequential merge sorts performance. The results are normalised to the execution time of *Slices* (Smaller is better). *Verus* consistently outperforms both *Slices* and *Slices Unchecked*.

the sequential mode and from 10,000 to 100,000,000 elements in the parallel mode. For each input size, between 700 and 4,000 runs were performed, each using a freshly generated random array. The same array was used across all algorithms to ensure fairness. Input generation time was not included in the performance measurements.

4.2 Results

Figure 1 shows the performance of the three merge sort variants. *Slices* is 2-3% slower than both *Verus* and *Slices Unchecked*, as expected due to its run-time bounds checks. *Verus* is 0.4-0.7% faster than *Slices Unchecked*. While the difference is small, it is statistically significant. The cause of this speedup is unclear and requires further study. Figure 2 presents results for parallel merge sort. *Slices Unchecked* remains 1-7% faster than *Slices*, depending on input length. However, *Verus* shows a surprising 3-14% slowdown, peaking at 14% on 10 million elements. The cause remains unclear. We tested multiple factors, including ghost state, `PCells`, `Arcs` and Verus safety wrappers for thread spawning, but none explained the degradation. We leave this for future investigation.

Bounds checking accounts for only 2–3% slowdown on average in both sequential and parallel merge sorts, suggesting that unsafe vector access may be unnecessary in some cases. Second, supporting primitives like `split_at_mut` in Rust verifiers could eliminate the need for permission cells in simple partitioning cases like merge sort and potentially result in faster implementations similar to *Slices Unchecked*. We expect our thread-safe array to be more beneficial in complex parallel patterns where such splitting is not feasible, as in **rav1d**. Exploring this integration is left for future work.

5 Related Work

Verification of parallel and concurrent programs in Rust is a relatively new area, recently explored by [10] and [12].

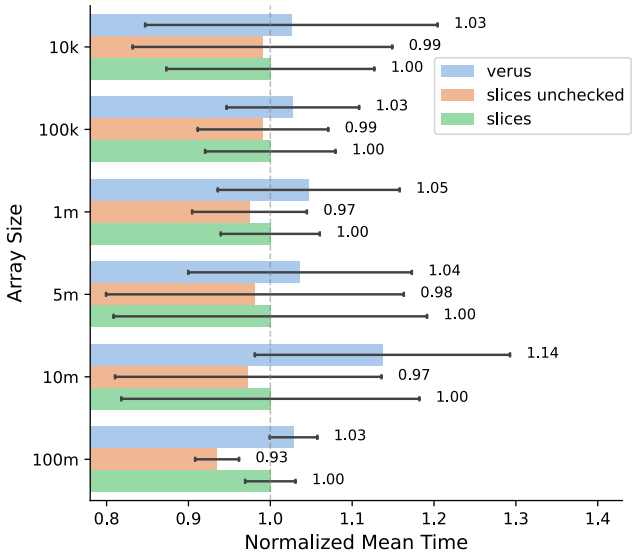

Figure 2. Comparison of parallel merge sorts performance. The results are normalised to the execution time of *Slices* (Smaller is better). *Slices Unchecked* remains the fastest, while *Verus* is now the slowest across all sizes, by no more than 14%, averaging 5.3% slower.

We focus on verifying memory safety in parallel array manipulation, demonstrating our approach on parallel merge sort. Similar work exists: [4] and [15] verify parallel sorting in verification-oriented languages; [7] targets linked lists; and [14] and [9] use VerCors [3], which supports only conventional languages without Rust's ownership type system. Rust differs from standard approaches: while most languages allow unrestricted aliasing and rely on external verifiers to ensure memory safety, Rust guarantees it by default. Verus safely extends Rust to be more expressive, making it impossible to break memory safety even if the specification is expressed incorrectly.

6 Discussion

In this paper, we establish the basis for verifying thread-safe arrays in Rust. Future work includes identifying the cause of the slowdown in parallel merge sort compared to standard slice splitting, applying our array API to other projects such as **rav1d**, and revisiting merge sort with mutable references once Verus supports them to simplify the implementation and assess annotation overhead.

7 Conclusion

In this work, we explored the verification of thread-safe arrays in Rust. We designed a statically checked, thread-safe array abstraction in Verus and demonstrated its applicability with a parallel merge sort. Our evaluation revealed an unexpected result: the verified implementation was slower

than the safe, run-time checked version, highlighting current performance trade-offs in Rust verification. Finally, we outlined future directions for Rust verifiers to enable both faster verified programs and lower annotation overhead.

References

[1] 2011. *Linux kernel source tree.* https://github.com/torvalds/linux Accessed July 17, 2025.

[2] 2018. *dav1d is the fastest AV1 decoder on all platforms :).* https://code.videolan.org/videolan/dav1d/ Accessed July 17, 2025.

[3] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM 2014: Formal Methods*, Cliff Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). Springer International Publishing, Cham, 127–131.

[4] Korbinian Breu. 2014. *Quantified Permissions for Sets and Sequences.* Master's thesis. DER TECHNISCHEN UNIVERSITAT M ¨ UNCHEN.

[5] Stephen Crane. 2024. *Porting C to Rust for a Fast and Safe AV1 Media Decoder.* https://www.memorysafety.org/blog/porting-c-to-rust-for-av1/

[6] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings* (Madrid, Spain). Springer-Verlag, Berlin, Heidelberg, 90–105. doi:10.1007/978-3-031-17244-1_6

[7] Christian Haack, Marieke Huisman, Clément Hurlin, and Afshin Amighi. 2015. Permission-Based Separation Logic for Multithreaded Java Programs. *Logical Methods in Computer Science* Volume 11, Issue 1 (Feb. 2015). doi:10.2168/lmcs-11(1:2)2015

[8] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

[9] Yernar Kumashev. 2020. GPGPU Verification: Correctness of Odd-Even Transposition Sort Algorithm. http://essay.utwente.nl/80585/

[10] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

[11] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (April 2023), 30 pages. doi:10.1145/3586037

[12] Callista Le, Kiran Gopinathan, Koon Wen Lee, Seth Gilbert, and Ilya Sergey. 2024. Concurrent Data Structures Made Easy. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 335 (Oct. 2024), 29 pages. doi:10.1145/3689775

[13] Daniel Marshall and Dominic Orchard. 2024. Functional Ownership through Fractional Uniqueness. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 131 (April 2024), 31 pages. doi:10.1145/3649848

[14] Mohsen Safari and Marieke Huisman. 2020. A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-Based Sorting Algorithms. In *Integrated Formal Methods: 16th International Conference, IFM 2020, Lugano, Switzerland, November 16–20, 2020, Proceedings* (Lugano, Switzerland). Springer-Verlag, Berlin, Heidelberg, 257–275. doi:10.1007/978-3-030-63461-2_14

[15] Thomas Tuerk. 2009. A Formalisation of Smallfoot in HOL. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 469–484. doi:10.1007/978-3-642-03359-9_32

[16] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. GhostCell: separating permissions from data in Rust. *Proc. ACM Program. Lang.* 5, ICFP, Article 92 (Aug. 2021), 30 pages. doi:10.1145/3473597