lace.rs - work-stealing while maintaining Rust safety guarantees

Daan Luth
University of Twente
Enschede The Netherlands

ABSTRACT

In task-based parallelism, work-stealing is an established method for distributing a multithreaded computation. Lace is a concurrent deque design intended for use in work-stealing. The Lace paper comes with an implementation in C, but C does not guard against many common multithreading problems like data races. The Rust programming language aims to resolve these problems, but not much literature exists on the performance of parallel algorithms in safe Rust. We implement the Lace algorithm in safe Rust and compare performance and scaling characteristics with a C reference implementation, the Rayon library, our library using unsafe Rust, and our library using a Chase-Lev deque. We found overhead for the safe version to be around 4 times that of the unsafe version, and 8 times that of the C version. Scaling behaviour was similar to what was reported in the Lace paper, except for the "fibonacci" benchmark. We achieved lower runtimes than Rayon on all benchmarks, but the scalability of Rayon seems better for some benchmarks.

KEYWORDS

Task-based parallelism, Load balancing, Work-stealing, Lace, Rust

1 INTRODUCTION

As hardware manufacturers face a power wall, multi-core systems have become ubiquitous. The challenge of effectively distributing a computational workload across parallel processing units has inspired much research effort over the last decades.

In parallel programming, a computation may be split into fine-grained "tasks" to be distributed across available threads. This paradigm enables the programmer to write multi-threaded programs in a more familiar sequential imperative style using fork-join primitives: subroutines can be tasks, and method call syntax can then be used to instantiate them. The actual distribution of tasks across available threads can then be done by a separate system. Frameworks like Cilk-5 [9] and Wool [8] provide `spawn` and `sync` primitives which create tasks and await their completion respectively.

Tasks may be distributed in various ways. Broadly, one can distinguish systems that map tasks to threads explicitly, implicitly, or based on (expected) work patterns. Explicit mapping yields good performance if the shape of the computation is predictable, but some workloads do not have much predictable structure [20].

42nd Twente Student Conference on IT, January 31st, 2024, Enschede, NL

Even if the structure is irregular, workloads may satisfy the following: If tasks are "spawned" from parent tasks, and must rejoin those, then the computation graph is a tree. If tasks can then also only start once all their arguments are available, the computation is "fully strict", and can be scheduled optimally by a work-stealing algorithm [4]. In work-stealing, idle processors process ("steal") tasks from randomly chosen busy ones to even out the load. Fully strict computations include a broad class of programs, notably those with fork-join semantics, and work-stealing is widely used [20].

For work-stealing, worker threads need to keep track of a pool of tasks to be executed. Task pools may be implemented using a doubly-ended queue ("deque"), which can be used similarly to a function call stack: `spawn`ed and `sync`'ed tasks are pushed and popped at one end, and thieves typically steal from the other end.

While work-stealing is asymptotically optimal, frameworks should endeavour to induce as little overhead on the computation as possible. Since the task deque is used in parallel by workers and thieves, access to it must be coordinated. This can be a major source of overhead, for instance, the "THE" protocol used by Cilk spends about half of its time in a memory fence on architectures without sequentially consistent shared memory [9]. Various deque designs have been considered to decrease synchronization overhead.

Lace is a design by Van Dijk and Van der Pol [6] where the deque is split into a shared and a private section. The "owner" thread operates on the private section unsynchronized, and thieves steal from the shared section. On the x86 architecture, this design only requires one memory fence when shrinking the shared portion, and a compare-and-swap operation when stealing.

The Lace paper includes an implementation as a C library. However, C was designed before the current era of multiprocessor programming, and the language does not help the programmer to prevent multithreading problems such as data races. Rust attempts to solve these problems by restricting the class of programs written in it [14]. For cases where Rust is too restrictive, an "unsafe" mode exists which allows the programmer to locally perform operations which the compiler cannot prove safe [19, Ch. 19.1]. "unsafe" code may violate safety invariants, and excessive use may indicate that the programmer is "writing C in Rust" and disregarding safety. We refer to Rust without use of "unsafe" as "safe". Rust uses an ownership system, which gives the compiler more pointer aliasing information when optimizing. This may lead to improved performance in parallel settings, since the Rust compiler can assume uniqueness of reference in places where a C compiler could not.

There is not yet much literature on the performance of parallel algorithms in safe Rust. In recent work Abdi et al. ported 14 parallel benchmarks from C++ to Rust and explored when the programmer must use "unsafe" code [1], but they do not cover performance characteristics and scaling behaviour in depth.

In this work, we present a case study[1] in implementing a work-stealing system in Rust using minimal "unsafe" code while keeping the API from the C Lace implementation. We compare performance with an implementation using more "unsafe", an implementation using a Chase-Lev deque, the C implementation, and the well-known Rayon library. We pose the following research questions:

(1) How can the Lace algorithm be implemented in Rust with minimal use of "unsafe" Rust?
(2) How does the performance of a Rust work-stealing library using the Lace deque compare to existing Rust multiprocessing solutions and the C implementation?

In section 2 we review work-stealing deque designs and existing work-stealing frameworks. We also review literature on how concurrent shared state may be implemented in Rust. We introduce relevant Rust semantics in section 3, and in section 4 we cover details of the implementation process and setup of the empirical evaluation. In sections 5 and 6 we present our results and analysis.

2 RELATED WORK

2.1 Work-stealing deques

In 2001, Arora et al. [3] published the first non-blocking concurrent deque for use in work-stealing. This "ABP" algorithm requires synchronization for every steal and pop. The design uses a fixed-size buffer, risking overflows. Work by Hendler et al. resolves this using a deque backed by a linked list of small arrays [10]. A linked list with one node per task is not used due to the larger memory and allocation overhead [10], but this choice makes the algorithm more complex. In [5], Chase and Lev present a simpler solution: A deque backed by a circular buffer, which can be resized without the need to update deque indices. This method relies on the indirection introduced by mapping indices onto the circular buffer, and enables reuse of buffer space previously used by stolen tasks.
For some workloads, the number of `push` and `pop` operations is much greater than the number of `steal`s. This may happen if the computation tree has a roughly constant branching factor and little variation in task duration; thieves steal tasks that are about as large as those being executed by the worker they stole from so few steal operations are required to even out the load. Here, one can trade off increased `steal` overhead for reduced `push`/`pop` overhead. At one extreme, Acar et al. [2] present a fully private design, where tasks can only be stolen using a transfer cell. The deque owner then moves tasks into the transfer cell when other threads become idle. In this design `push` and `pop` operations require no synchronization.
In 2014, van Dijk and van der Pol presented Lace, which uses a split deque [6]. The algorithm uses a single buffer, and keeps three indices: `tail`, `split` and `head`, splitting the buffer into a shared (`tail` to `split`) and a private (`split` to `head`) section. Thieves atomically increment `tail` to steal tasks, and the deque owner operates on the private portion of the deque without needing to synchronize with thieves. Thieves request the owner to grow the shared section if it is empty, and the owner may shrink the shared section again if it runs out of work.

[1]Source code and collected traces can be accessed using DOI 10.5281/zenodo.14713348

2.2 Existing Rust work-stealing Frameworks

We review some existing work-stealing frameworks and describe notable design choices and features: In 2014 Linus Färnstrand presented the "ForkJoin" framework [7]. This framework uses an off-the-shelf implementation of a Chase-Lev deque, and work-stealing without leapfrogging. The library uses a separate pool for each input-output task signature. This was done because the task objects are generically typed, and so the pool containing them had to take on the same generic type [7, p. 26]. This issue was also encountered in this work, but resolved differently (4.1.1). To allow tasks to borrow data with lifetimes shorter than the lifetime of the thread pool, ForkJoin uses unsafe code. The API presented is divided into three "algorithm styles" (Reduce, Search, and In-place mutation), and requires tasks to be broken up into a "task" and "join" stage. A notable addition to the stealing code is a linear back-off algorithm, used to limit the number of failed steal attempts and decrease contention.
Another framework created around 2015 is "jobsteal" [18], which takes a more pragmatic approach regarding unsafe code. Task objects are either heap-allocated closures or "inlined" into the task object as raw untyped closure data. While this does allow for a single buffer to store tasks with arbitrary types, typing and lifetime guarantees must then be upheld by the rest of the framework. Task objects are not put into the deque directly, but allocated in a separate arena, and pointers to tasks are kept in the deque. This is necessary because jobsteal uses a Chase-Lev deque which provides no way for the owner to recover stolen task data. As for the API, the library uses scopes, where tasks spawned in a scope are finished before its end. Scopes are an effective way to denote when task input lifetimes end, but their hierarchical structure makes it difficult to use them in loops. For this, jobsteal features parallel iterators.
Rayon is a well-known and mature Rust parallelism library. The library was initially developed by Matsakis, a core Rust developer [1, 7], and has since been adopted into major Rust projects including the standard `rustc` compiler. Other work ([1]) has also used rayon as an assumed performant solution. Like "jobsteal", rayon uses closures for tasks, has a parallel iterator API, and internally uses a Chase-Lev deque from the "crossbeam_deque" library [16]. Besides parallel iterators, Rayon also features scopes, and a "join" primitive to spawn two tasks and wait for their results.

2.3 Shared-state concurrency in Rust

The Lace algorithm uses data accessed by multiple threads at once. To store indices and flag values one can make use of Rust's atomic types, but these cannot be used for the backing buffer. The algorithm guarantees uniqueness of access to buffer slots at any one time, but the compiler cannot verify this. Due to the AXM principle it is not straightforward to create mutable shared state in safe Rust. Some work has been done to find appropriate abstractions.
In 2024, Hong et al. presented case studies in shared mutable state in Rust in an operating systems context [11]. They identify six patterns, among which the "Process-owned value" pattern seems to best match the description of the deque buffer elements, since each element is only accessed by one thread at a time. For this case, they use an `UnsafeCell` and maintain the AXM invariant manually.
Yanovski et al. present a way to allow the compiler to statically prove safety properties for access to an `UnsafeCell` [21]. Their

approach involves creating a type-branded token, which can then be used to access the contents of a correspondingly branded "GhostCell". The impossibility of copying the token ensures data is only accessed by one thread at a time. This abstraction allows the compiler to prove safety, but does not impose any runtime overhead [21]. Using this method to guard access to buffer elements would require thieves to have access to one token for each task in the private section. However, one must then ensure thieves only access one of the tokens at a time - which is another version of the problem we were trying to solve. Thus, this approach does not seem suitable to control access to buffer elements.

Abdi et al. present a case study porting 14 parallel algorithm benchmarks to Rust [1]. In their categorization, the Lace deque algorithm falls into the "SngInd" category, since the backing array is accessed using indices which are unique, but whose relation is too complex for the compiler to prove safety. They suggest using runtime checks and atomic or unsafe accesses to buffer elements.

In summary, existing work suggests using unsafe accesses in the form of `UnsafeCell` or `Cell`, and guarding against bugs using runtime checks or manually enforced invariants.

3 BACKGROUND: RUST

Rust statically guarantees the absence of data races and dangling references [14]. This is done using an ownership and lifetime system, and by prohibiting multiple mutable references to data from existing at once. We discuss relevant details of these systems:

In a program, every value has exactly one "owner" scope. This system allows the compiler to deallocate ("drop") the value's memory when the owning scope ends, thus avoiding the need for garbage collection or manual memory management [19, Ch. 4.1]. For instance, a function (scope) takes ownership of its arguments.

Parts of a program (e.g. functions) may need access to data without deallocating related memory when finished. For this, Rust has the concept of "borrowing". A "borrow" of a value is a reference to it [19, Ch. 4.2] which allows usage without taking ownership.

If data is borrowed and then dropped, the borrow reference would be dangling. To prevent this, every value has an associated lifetime [19, Ch. 10.3]. The compiler then verifies that no borrow "outlives" the data it references. The largest lifetime, corresponding to the runtime of the entire program execution, is called `'static`.

Rust distinguishes between mutable and immutable variables. When borrowing, a reference may also be declared mutable or immutable. To prevent data races, Rust enforces that there is never more than one mutable borrow of any location [19, Ch. 4.2]. Moreover, a mutable reference may not exist while an immutable reference does.This is referred to as the "aliasing XOR mutability" (AXM) principle: The compiler allows either multiple immutable aliases, or one mutable alias, to exist; mutability implies a lack of aliasing.

The AXM principle strictly prohibits mutable shared state, but there are cases where the borrow tracking is too granular and restrictive. For these, Rust has "interior mutable" types [19, Ch. 15.5], whose contents can be changed through a shared reference. These types enforce the borrowing invariants at runtime.

To enable code reuse, Rust features traits [19, Ch. 10.2], which are akin to interfaces in an object oriented language. A **`struct`** may implement an arbitrary combination of traits. Two notable traits are `Send` and `Sync`, which specify that a value may be sent between threads, or accessed by multiple threads at once, respectively.

3.1 Semantics and Undefined Behaviour

When optimizing a program, the compiler must know which semantics it must uphold and which are "Undefined Behaviour" (UB) and can be violated to gain better performance. When using unsafe code to create concurrent shared state, one may want to create multiple **`&mut`** references to a single location. However, having such references at runtime is UB, because it violates the pointer aliasing rules [12, Ch. 16.2]. When using **`&mut`** reference, the compiler assumes that it is unique and so it could, for instance, constant-propagate a value stored to it as the result of a later dereference. This optimization could yield correctness violations if the data at the reference is being mutated by a different thread. Instead, one can use raw **`*mut`** pointers, which can alias arbitrarily with other references. [19, Ch. 19.1] However, **`*mut`** pointer dereferences must be marked "unsafe", and use of them to mutate a single data structure may not be future-proof: Rust does not yet have an established pointer aliasing model [12, Ch. 3.1], so undefined behaviour is defined by elision as anything the official "`rustc`" compiler might violate. A formal aliasing semantics which may become official is "Stacked Borrows" by Jung et al. [13]. In this model, having multiple mutable shared references (including raw pointers) to any location at any point is UB, even if they are unused in the program [13]. To avoid the possibility of our implementation invoking undefined behaviour in the future, we obey these semantics.

4 METHODOLOGY

4.1 Implementation

While implementing the library, there is a trade-off between safety and runtime speed, but to allow for fair comparison with existing systems we had to optimize for speed. To strike a balance, we first created a fast version of the library using an arbitrary amount of "unsafe", and then rewrote it to use as little "unsafe" as possible.

To reduce overhead, we iteratively used the "perf" profiler to find hot code paths, and rewrote the library to decrease the number of instructions used in those paths and add shortcuts for common cases. To this end we also implemented metrics for, among others, the total number of tasks, steals, and split-point movements.

We initially optimized for the "`fibonacci`" benchmark. This is a suitable benchmark to optimize for since tasks are small and so library overhead dominates the runtime. However, the number of steal operations is relatively low, so we later also optimized for benchmarks with more steal attempts like "`matmul`" and "`uts-t3l`".

To avoid invoking undefined behaviour while using unsafe code, we used the MIRI tool (`0.1.0 (917a50a039 2024-11-15)`). This also ensures compliance with the Stacked Borrows semantics. MIRI was run with only "`-Zmiri-backtrace=full`". To further avoid data races we used the ThreadSanitizer tool. Since ThreadSanitizer does not support procedural macros (see 4.1.4), we manually transformed task functions into the expected form. We used Rust 2021 ("`1.83.0 (90b35a623 2024-11-26)`"), with cargo.

To facilitate porting and comparison of the benchmarks, we reused the API from the C implementation. This introduces constraints, most notably when implementing task objects.

4.1.1 Tasks. Implementing task objects using closures is a straightforward and idiomatic solution used by both "jobsteal" and rayon [17, 18]. Closures allow the user to create tasks in a convenient ad-hoc way. However, preliminary testing revealed closures have poor performance compared to static function pointers. Using closures as tasks does not offer any semantically new options over using static methods, so we decided against using closures.
When using the system, a programmer should be able to mix tasks with different signatures. This was implemented using Rust's support for generic types. However, a problem arises when task objects are stored in the deque buffer: The buffer would need to have a generic type to reflect the task type (e.g. `[Task<I,O>]`). This would limit the programmer to one kind of task signature per deque.
The idiomatic way to resolve this would be to use a `Task` trait and store **`dyn`** `Task` objects instead. These keep a virtual method table to specific `Task<I,O>` methods, similarly to a C++ object. To extract task output data one could use a checked downcast to a concrete type using the `Any` trait. However, use of `Any` requires the generic parameters to have a **`'static`** lifetime. So this approach would not allow for task inputs with smaller lifetimes (i.e. borrows). This problem was also found in [7, p. 25]; since our framework aims to use a single thread pool and allow tasks use non-**`'static`** borrows, some unsafe code will be needed [7]. The lifetime problem exists fundamentally because the compiler is unaware of the LIFO semantics of the deque: If a `push()`'ed task borrows a variable, the compiler cannot tell that the borrow is dropped after the next call to `pop()`, and so it requires the lifetime to be over-approximated to at least that of the buffer. We resolved this by converting `Task<I,O>` instances to opaque byte buffers. Lifetime and type semantics must be upheld by provided abstractions and the library user.
This approach does present a different problem: If a type-erased task is stolen, the thief does not know which type it originally had, and so cannot take ownership of the input data and execute the task. This was solved by keeping a pointer to a function that can "unerase" the data to the correct type. This works like a virtual method table, but only incurs the related overhead for steals.
To regain some safety, we could then introduce a runtime type check by comparing the function pointer against what it should be if the type is correct. This check provides as strong of a safety guarantee as the usage of `Any` proposed earlier.

4.1.2 Deque buffer. The buffer backing the deque has to be shared between threads but still mutable. This was achieved using Rust's `UnsafeCell` type, as suggested in [11, p. 8] and [1]. We could lower the amount of **`unsafe`** code by using a `Cell` instead. Under correct operation of the algorithm, no buffer slot can be accessed from different threads at once, so the `Cell` requirements are satisfied. The compiler cannot verify this on its own, so the `Sync` marker trait was used. Implementation of this trait is the only use of "unsafe" code for the buffer. To implement leapfrogging, a separate `thief` buffer is used to communicate about stolen work.

4.1.3 Deque. The Lace deque could be implemented straightforwardly using Rust's support for atomic types. Shared variables and the buffer were accessed through an `Arc` pointer. To prevent false sharing due to overlap in cache lines between owned and shared variables, we used the `CachePadded` type from "crossbeam-utils". The design is depicted in UML-like notation in figure 1.

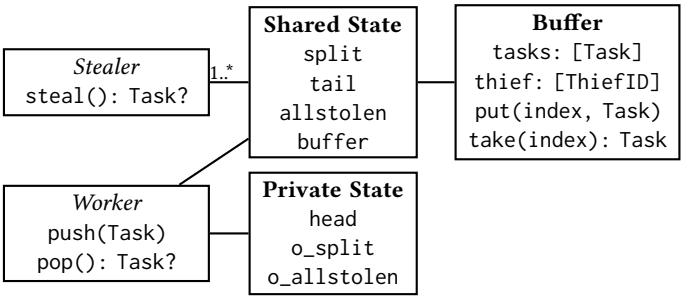

Figure 1: Deque implementation diagram

4.1.4 API. The API exposed by our library mimics that of the C Lace implementation (and, in turn, Wool [8]), and provides "spawn", "sync" and "call" primitives. We added a "join" primitive for the common "spawn-call-sync" idiom with two tasks. To use Rust methods as tasks we wrote a procedural macro ([19, Ch. 19.5]) called `lace_task`, which works as in the following example:

```
#[lace_task]
fn fib(n: usize) -> usize {
  if n < 2 { n } else {
    let (x, y) = join!(fib(n - 1), fib(n - 2));
    x + y
  }
}
```

Gets rewritten to (macro definitions omitted for brevity)

```
fn fib(_worker: &mut Worker, n: usize) -> usize {
  macro_rules! spawn, sync, call, join { ... }
  if n < 2 { n } else {
    let (x, y) = join!(fib(n - 1), fib(n - 2));
    x + y
  }
}
```

For comparison, in Rayon this could be written as

```
fn fib(n: usize) -> usize {
  if n < 2 { n } else {
    let (x, y) = rayon::join(
      || fib(n - 1),
      || fib(n - 2));
    x + y
  }
}
```

To uphold safety guarantees, "spawn" creates a zero-size token of type `SpawnToken<'task, I, O>` to be consumed by "sync". The lifetime parameter on the token encodes that values borrowed by tasks must not be used while the token exists. For example, the following yields a compile-time error:

```
#[lace_task]
fn f(_: &mut usize) {
  let mut y = 4;
  let tkn = spawn!(f(&mut y));
```

```
5    y += 2; // error: use of borrowed value 'y'
6    sync!(tkn);
7  }
```

4.1.5 Rust atomic integer support. The Lace algorithm requires thieves to do a compare-and-swap on the combination of the "tail" and "split" variables and owners to set "split" independently. We used a single `AtomicUsize` to store `(split, tail)`, but Rust does not straightforwardly allow atomic access to only the bits of the `split` index. We used a `fetch_xor((old_s ^ new_s) << 32)` operation to change the split point.

4.2 Empirical evaluation setup

To compare the implementation with existing systems, we reused benchmarks from the original Lace paper. Performance was then compared with the C Lace implementation, Rayon (version `1.10`), and our implementation using a Chase-Lev deque[2]. The C implementation uses the "random stealing" extension by default, where workers steal from random victims if leapfrogging fails. We implemented this extension the same way, and measurements were done with this functionality enabled. While implementing the benchmarks, we aimed to keep conditions the same between different versions, such that the only free variable was the underlying library. We encountered the following complications.

4.2.1 Comparison with the Chase-Lev version. The algorithm by Chase and Lev was not designed with use of the deque as a call stack in mind: There is no way for the owner to retrieve stolen task results. We resolved this by heap- or arena-allocating task **`struct`**s and keeping raw pointers to these in the deque. The owner then keeps another pointer in case the task is stolen. This scheme allowed us to use an off-the-shelf implementation, but does impact the comparison to the Rust Lace version due to overhead related to allocations and pointer dereferencing. We considered augmenting the Chase-Lev algorithm to support a call-stack API by using an age tag with the `top` variable to avoid the ABA problem, as in the ABP algorithm ([3]). However, this would add overhead to update the tag when popping. It was unclear which of these overheads would dominate, and the described algorithm change could be argued to invalidate this comparison, so we did not pursue this option.

4.2.2 Comparison with the C implementation. When comparing the Rust library with the C library, task bodies[3] may run at different speeds in Rust and C. This limits our ability to draw conclusions about performance, since a speedup or slowdown may be caused by the difference in task body code rather than a difference between the libraries. This was mitigated by using C code from the original Lace benchmarks for task bodies. This measure does not impact the comparison between different Rust versions. The switch at runtime between Rust and C does introduce some overhead, but this seems to be negligible compared to the change from Rust to C, so this measure helps to close the comparison gap.

4.2.3 Comparison with Rayon. Rayon supports parallel iterators, but the C implementation does not. For some benchmarks (notably N-Queens) we opted to use these as they represent an idiomatic solution. This does introduce a difference in semantics: rayon parallel iterators may spawn and sync tasks in any order, where the benchmark using our library does not. We did not close this gap, as it represents one in typical use of both libraries.

4.2.4 Differences induced by code placement. We found that code reordering effects could significantly change performance for the `matmul` benchmark: Enabling features only used by the parallel Rust Lace benchmark like "`leapfrog_random`" produced a change in the runtime of the sequential version as well[4]. To control for this effect we recorded sequential execution times with (seemingly) unrelated features enabled, and compute speedup factors relative to the sequential version with the same features enabled.

5 RESULTS

5.1 Use of unsafe code

In the final version of our library, unsafe code is used in the following ways:

- To assign the `Sync` trait to the backing buffer.
- To convert task objects to and from opaque byte buffers.

However, we found that overhead could be reduced significantly by also using "unsafe" code in the following ways:

- To inline small task inputs into the task descriptor rather than heap-allocating them. The inlining threshold used was 6 times a pointer size, like in the C implementation.
- Use of arena allocation to avoid heap-allocating larger task inputs similarly improved performance.
- To read task data from the buffer without erasing it, using `std::ptr::write` and `std::ptr::read`. Task data is only read once so this optimization does not risk correctness.

In our artifact, these changes can be applied using the `unsafe_task`, `unsafe_arena`, and `unsafe_buffer` feature flags respectively.

5.2 Performance characteristics

The test machine has two Intel Xeon Silver 4314 processors, and runs Ubuntu 22.04 LTS with kernel version 5.15.0-124. For each combination of benchmark and worker count we took the median of 20 measurements. Speed-up factors were computed relative to sequential execution (T_S) and execution with one worker (T_1).
The "safe" version is the base library using random leapfrogging. The "unsafe" version also uses all extra features mentioned in 5.1. We obtained the following runtimes.

safe Rust	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.13	35.31	1.97	1.59	17.94
`matmul-2048`	4.89	4.92	0.37	13.14	13.21
`queens-14`	18.46	19.39	1.04	17.69	18.58
`uts-t2l`	21.19	24.74	1.09	19.37	22.61
`uts-t3l`	16.2	19.22	1.08	14.95	17.75

[2]From the "crossbeam-deque" crate, version 0.8.
[3]By "body" we mean any work unrelated to bookkeeping for work-stealing.
[4]This is due to the way we organized the (Rust) benchmarks and library code: One executable handled both sequential and parallel versions, and cargo feature flags were used to enable or disable parts of the library, affecting the whole binary.

unsafe Rust	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.13	9.29	0.5	6.3	18.7
`matmul-2048`	4.89	4.91	0.39	12.63	12.68
`queens-14`	18.55	18.93	1.02	18.21	18.59
`uts-t2l`	21.34	23.84	1.02	20.98	23.43
`uts-t3l`	16.74	17.75	0.94	17.79	18.86

Between these, we can see that library overhead (as measured on `fib`) is higher for the safe version. This seems to be caused mainly by the use of heap allocation of small task inputs; inlining task data into the task descriptor largely eliminates the overhead:

safe +inlining	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.13	11.19	0.53	5.91	21.11
`matmul-2048`	5.53	5.59	0.38	14.52	14.68
`queens-14`	18.66	19.45	1.03	18.05	18.81
`uts-t2l`	20.95	23.5	0.98	21.46	24.07
`uts-t3l`	16.06	17.73	0.95	16.91	18.66

Comparing with the C implementation, we see that absolute overhead for the Rust version is larger, but scaling behaviour is similar between the C and "unsafe" Rust versions.

C Lace	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	2.17	4.38	0.25	8.65	17.41
`matmul-2048`	4.88	4.96	0.41	11.88	12.09
`queens-14`	15.46	15.26	0.82	18.96	18.72
`uts-t2l`	20.82	21.39	0.92	22.58	23.2
`uts-t3l`	15.7	15.47	0.83	18.81	18.54

The Rayon and Chase-Lev versions have greater overhead but seem to scale better on the test hardware, especially for the `fib` benchmark. This is discussed further in section 5.3.

Rayon	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.01	52.09	1.81	1.66	28.73
`matmul-2048`	5.53	5.59	0.55	10.03	10.15
`queens-14`	19.15	32.89	1.78	10.76	18.47
`uts-t2l`	21.24	25.4	1.18	17.99	21.52
`uts-t3l`	16.06	20.02	1.3	12.31	15.34

Chase-Lev	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.12	54.04	1.81	1.72	29.85
`matmul-2048`	5.53	5.59	0.51	10.85	10.98
`queens-14`	19.02	20.05	1.1	17.26	18.2
`uts-t2l`	21.11	26.39	1.06	19.86	24.82
`uts-t3l`	16.53	20.84	1.13	14.68	18.5

Steal operations can fail in two ways: Either the deque was empty, or the compare-and-swap ("busy") failed due to steal contention. We recorded steal outcomes[5] for each benchmark at 48 workers. Using median values over 50 runs to get the following table.

[5]The Lace paper counts random and leap-frogging steals separately, we combine these.

benchmark	#tasks	#steals		
		successful	busy	empty
safe Rust				
`fib-45`	1836311902	8788	120	2255302
`matmul-2048`	449389	185886	2332	108877330
`queens-14`	27358552	3990	36	2021738
`uts-t2l`	96793509	34286	231	2969814
`uts-t3l`	111345630	3610010	40908	103433202
C Lace				
`fib-45`	1836311902	7722	133	852365
`matmul-2048`	449389	199361	12432	670047466
`queens-14`	27358552	2541	14	249034
`uts-t2l`	96793509	27374	178	1616104
`uts-t3l`	111345630	3526804	30873	154021447

We found the number of "empty" steals to have large outliers for the `fib-45`, `matmul-2048`, `queens-14` and `uts-t2l` benchmarks. We suspect this to be due to thread starvation.

5.3 The fibonacci benchmark

Because tasks are small, the "`fib`" benchmark is relatively sensitive to library overhead. Using data for 1, 2, 4, 8, 16, 24, 32, 40 and 48 workers, we obtain the speedup and runtime graphs in figures 2 and 3. Although the overhead of the Rayon version is larger, its scaling behaviour seems better. Unfortunately we did not have access to more processors to find where this speedup curve flattens off.

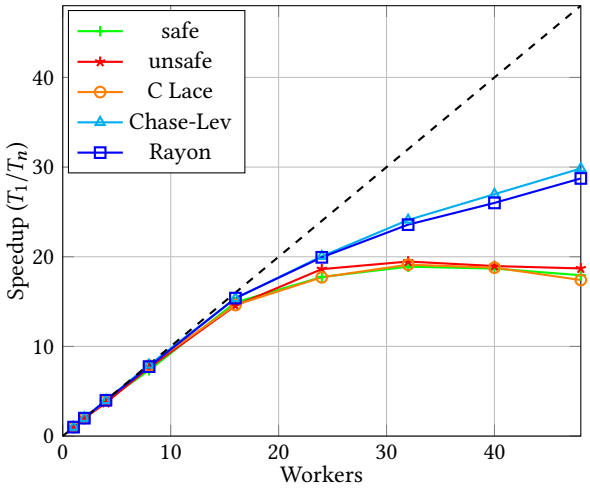

Figure 2: Speedup graph for `fib-45`.

The poor scaling behaviour of the Lace versions is contrary to what was found in [6]. We discuss our investigation into this behaviour. For the C implementation using 48 workers, we found the number of successful leapfrogging operations for `fib 50` to be much lower than reported in the Lace paper. Most of the attempts to leapfrog thieves failed due to the deque being empty. This, along with a similarly low number of grow operations, suggests that workers are slow to process split-point move requests. This could suggest contention due to hyper-threading: The test platform has 32 physical cores, and uses hyper-threading to present 64 logical cores. We tested the Rust Lace and Chase-Lev versions using 32 worker

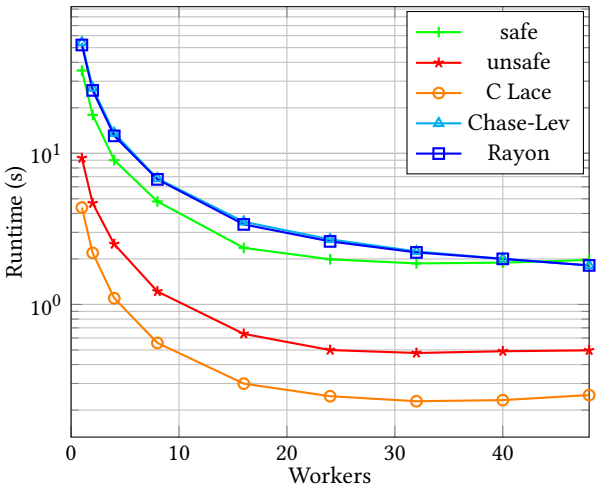

Figure 3: Median runtime (T_n) for `fib-45`

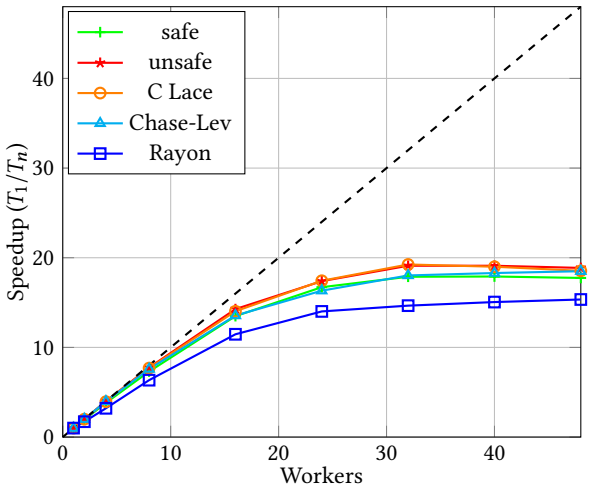

Figure 4: Speedup graph for `uts-t3l`.

threads pinned in a way so as to produce either minimal or maximal hyper-threading contention. We found that the Lace version was more adversely affected by hyper-threading with a slowdown factor of 2.2 compared to the Chase-Lev versions' 1.4. This could mean that the Chase-Lev version (and Rayon) can scale further with hyper-threading. These findings partially explain the unexpectedly poor scalability past 32 workers. We suspect caching effects may also play a role: At 32 workers, we found the cache miss rate for `fib 50` to be ±60%. Use of the receiver-initiated deque by Acar et al. implemented in [6] yielded cache miss rate of ±30% at 32 workers and a speedup of 34.12 at 48 workers. The cores on the test system are spread across two sockets, which may exacerbate the detrimental effects of a cache miss. Although the speedup at 48 workers using the receiver-initiated deque is larger, the absolute runtime was greater than that of the C Lace version. This may imply that all available parallelism is exhausted by the Lace versions, and that the Chase-Lev version can scale further because it simply has more overhead left to get rid of. Due to time constraints we could not find conclusive reasons for the observed behaviour.

5.4 The UTS benchmark

We measured runtimes for the UTS ([15]) benchmark using tree `t3l`, resulting in the speedup graph in figure 4. On this benchmark the absolute runtimes are ordered the same way as for `fib`. Scaling behaviour up to 48 is better using our library than with Rayon. These findings corroborate what was reported in [6].

5.5 The matmul benchmark

The runtime distribution (see figure 6) for the `matmul` benchmark is somewhat unexpected: The safe Rust version is faster than the C version. As noted in 4.2.4 this benchmark was especially sensitive to spurious optimization, so it is unclear if this result is significant. We found that enabling random leapfrogging increased runtime for the `matmul` benchmark by around 1.4 times at 32 workers. The number of non-leapfrogging steals increased to around a quarter of all tasks. For this benchmark in particular, a random steal may have the worker thread access part of the matrix that is not in its cache. We found that enabling random leapfrogging lead to more cache misses (±26% rather than ±18%), and suspect this to be the cause for the performance difference. Turning off random leapfrogging yielded the following runtimes:

safe w/o `leap-random`	T_S (s)	T_1 (s)	T_{48} (s)	T_S/T_{48}	T_1/T_{48}
`fib-45`	3.01	35.43	2.02	1.49	17.52
`matmul-2048`	5.53	5.63	0.29	19.32	19.7
`queens-14`	19.15	19.92	1.05	18.32	19.05
`uts-t2l`	21.24	24.66	1.1	19.37	22.49
`uts-t3l`	16.06	19	2.17	7.41	8.77

5.6 General remarks

Using the average speedup across each of the benchmarks we get the graph in figure 5. On average, the libraries using the Lace deque

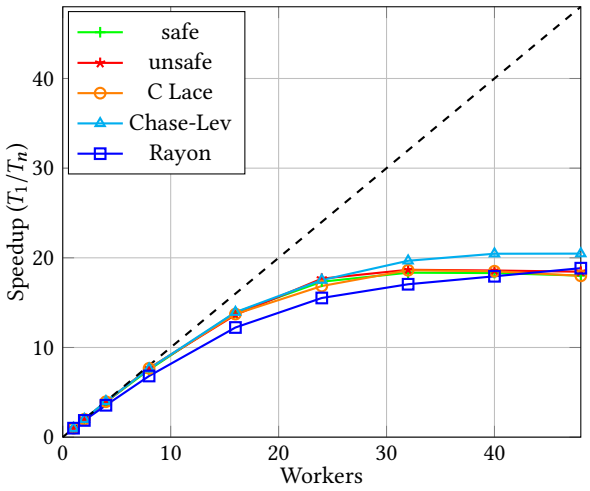

Figure 5: Speedup graph for `average`.

scale better below 40 workers, but the Rayon and the Chase-Lev versions overtake past this point. For the Lace versions we see diminishing returns from 32 workers on. We suspect this to be in part due to hyper-threading, as discussed in 5.3. Normalizing

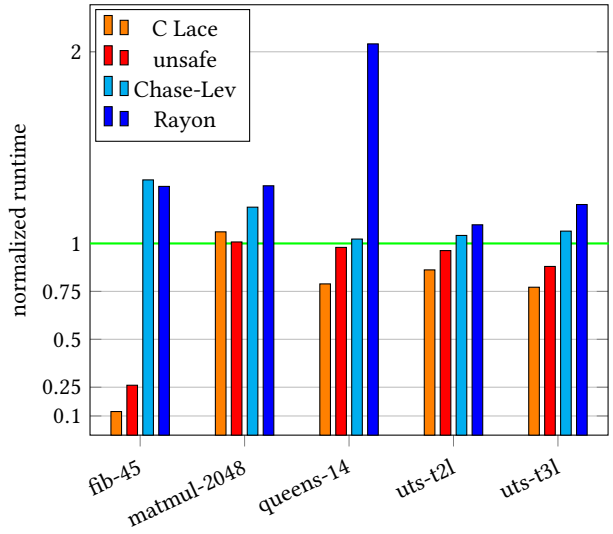

Figure 6: Averages of median runtimes relative to those of the "safe" Rust version, over all worker counts.

runtimes of the different library versions against our base library, we get the bar plot in figure 6. As measured on `fib`, overhead for the "safe" version is on average 3.8 times that of the "unsafe" version, and 2.1 times larger for the unsafe version relative to the C version. In earlier work the disuse of unsafe Rust increased runtimes by an average factor of only 2.1 at 24 threads [1], suggesting the Lace algorithm is particularly affected by this limitation. The runtime difference between the sequential versions of safe Rust and C `fib` is a factor of around 1.4, suggesting our library may be inducing more overhead than necessitated by the change of language.

6 CONCLUSION

In this paper, we presented a case study implementing a work-stealing algorithm in safe Rust, and compared performance with existing multiprocessing solutions. We answered two questions:
RQ1 - How can the Lace algorithm be implemented in Rust with minimal use of "unsafe" Rust? We could implement the algorithm using five lines of "unsafe" Rust, conceptually handling two cases:

- (1 line) To implement `Sync` for the task buffer. This is unavoidable since the used `Cell` type does not implement `Sync`.
- (4 lines) To convert task objects to and from opaque byte buffers. As discussed in 4.1.1 this could not be avoided except by changing the library API, and we did not do so because it would complicate performance comparison for RQ2.

We found that library overhead could be reduced significantly by introducing "unsafe" code to avoid heap allocation of task inputs and erasure of values only read once. We conclude that Rust's restrictions on data lifetimes and initialization status hamper both performance and API expressivity for our case. The latter could be improved with fairly little use of "unsafe".
RQ2 - how does the performance of a Rust work-stealing library using the Lace deque compare to existing Rust multiprocessing solutions and the C implementation? Our experiments show that the safe Rust library achieves similar scaling behaviour to the C Lace implementation, but has around 8 times the overhead. We found that use of "unsafe" code could reduce overhead down to around twice that of the C implementation. The Rust library achieved lower absolute runtime than Rayon on all benchmarks, and scaling behaviour up to 48 workers was also better than that of Rayon for all benchmarks with the exception of "`fibonacci`". Scaling behaviour between the safe and unsafe Rust and C Lace versions was largely similar.

7 FUTURE WORK

7.1 NUMA-awareness in victim selection

We found that performance for some benchmarks is impacted by hyper-threading and cache contention. NUMA-aware victim selection when stealing could help to reduce these effects. We did not implement this functionality due to time constraints.

7.2 Backoff algorithm to reduce steal contention

For the `matmul` benchmark performance was impacted by the relatively large number of failed steal attempts. Preliminary testing revealed that scaling behaviour could be improved using a backoff algorithm to reduce steal contention (as done in [7]). However, such an algorithm caused performance to decrease for other benchmarks. Future work could investigate this further.

7.3 Augmenting the Chase-Lev algorithm

As described in 4.2.1 the option of augmenting the Chase-Lev algorithm to support a call-stack API was not used. Future work could explore this option and related performance aspects.

7.4 Difference between Chase-Lev and Rayon

For the `queens`, `matmul` and `uts-t3l` benchmarks there seems to be a significant difference in scaling behaviour between our library with the Chase-Lev deque and Rayon. This is unexpected since Rayon internally uses the same Chase-Lev deque. Future work could investigate this difference.

7.5 Optimal deque algorithm selection

In this work we found significant differences in scaling behaviour between deque algorithms for some benchmarks. We did some experiments to find the causes of these discrepancies, but future work could investigate more deeply which deque algorithm can be used to achieve optimal performance on a given class of workloads. Similarly, one could investigate why some of these algorithms perform better or worse in the presence of hyper-threading.

8 AI DISCLOSURE

The author used ChatGPT ("GPT-4o mini") to learn about Rust features useful for implementation. The author has not used any suggested code verbatim, reviewed suggestions as needed, and takes full responsibility for the content of this work.

REFERENCES

[1] Javad Abdi et al. "When Is Parallelism Fearless and Zero-Cost with Rust?" en. In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. Nantes France: ACM, June 2024, pp. 27–40. ISBN: 9798400704161. DOI: 10.1145/3626183.3659966. URL: https://dl.acm.org/doi/10.1145/3626183.3659966 (visited on 11/12/2024).

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. "Scheduling parallel programs by work stealing with private deques". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 219–228. ISSN: 0362-1340. DOI: 10.1145/2517327.2442538. URL: https://doi.org/10.1145/2517327.2442538 (visited on 01/25/2025).

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. "Thread scheduling for multiprogrammed multiprocessors". In: *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. SPAA '98. New York, NY, USA: Association for Computing Machinery, June 1998, pp. 119–129. ISBN: 978-0-89791-989-0. DOI: 10.1145/277651.277678. URL: https://dl.acm.org/doi/10.1145/277651.277678 (visited on 01/25/2025).

[4] Robert D. Blumofe and Charles E. Leiserson. "Scheduling multithreaded computations by work stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: https://dl.acm.org/doi/10.1145/324133.324234 (visited on 01/25/2025).

[5] David Chase and Yossi Lev. "Dynamic circular work-stealing deque". en. In: *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. Las Vegas Nevada USA: ACM, July 2005, pp. 21–28. ISBN: 978-1-58113-986-0. DOI: 10.1145/1073970.1073974. URL: https://dl.acm.org/doi/10.1145/1073970.1073974 (visited on 10/31/2024).

[6] Tom van Dijk and Jaco C. van de Pol. "Lace: Non-blocking Split Deque for Work-Stealing". en. In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by Luís Lopes et al. Cham: Springer International Publishing, 2014, pp. 206–217. ISBN: 978-3-319-14313-2. DOI: 10.1007/978-3-319-14313-2_18.

[7] Linus Färnstrand. "Parallelization in Rust with fork-join and friends: Creating the fork-join framework". In: 2015. URL: https://www.semanticscholar.org/paper/Parallelization-in-Rust-with-fork-join-and-friends%3A-F%C3%A4rnstrand/df5a56d44a008530d66c9993040c83837829fcc8 (visited on 11/19/2024).

[8] Karl-Filip Faxén. "Wool-A work stealing library". In: *SIGARCH Comput. Archit. News* 36.5 (June 2009), pp. 93–100. ISSN: 0163-5964. DOI: 10.1145/1556444.1556457. URL: https://dl.acm.org/doi/10.1145/1556444.1556457 (visited on 11/19/2024).

[9] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. "The implementation of the Cilk-5 multithreaded language". In: *SIGPLAN Not.* 33.5 (May 1998), pp. 212–223. ISSN: 0362-1340. DOI: 10.1145/277652.277725. URL: https://dl.acm.org/doi/10.1145/277652.277725 (visited on 11/20/2024).

[10] Danny Hendler et al. "A dynamic-sized nonblocking work stealing deque". en. In: *Distributed Computing* 18.3 (Feb. 2006), pp. 189–207. ISSN: 1432-0452. DOI: 10.1007/s00446-005-0144-5. URL: https://doi.org/10.1007/s00446-005-0144-5 (visited on 11/20/2024).

[11] Jaemin Hong et al. "Taming shared mutable states of operating systems in Rust". In: *Science of Computer Programming* 238 (Dec. 2024), p. 103152. ISSN: 0167-6423. DOI: 10.1016/j.scico.2024.103152. URL: https://www.sciencedirect.com/science/article/pii/S0167642324000753 (visited on 11/14/2024).

[12] *Introduction - The Rustonomicon*. URL: https://doc.rust-lang.org/nomicon/ (visited on 11/19/2024).

[13] Ralf Jung et al. "Stacked borrows: an aliasing model for Rust". en. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3371109. URL: https://dl.acm.org/doi/10.1145/3371109 (visited on 11/14/2024).

[14] Nicholas D. Matsakis and Felix S. Klock. "The rust language". en. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. Portland Oregon USA: ACM, Oct. 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: https://dl.acm.org/doi/10.1145/2663171.2663188 (visited on 11/04/2024).

[15] Stephen Olivier et al. "UTS: An Unbalanced Tree Search Benchmark". en. In: *Languages and Compilers for Parallel Computing*. Ed. by George Almási, Călin Caşcaval, and Peng Wu. Berlin, Heidelberg: Springer, 2007, pp. 235–250. ISBN: 978-3-540-72521-3. DOI: 10.1007/978-3-540-72521-3_18.

[16] *rayon-rs/rayon*. original-date: 2014-10-02T15:38:05Z. Dec. 2024. URL: https://github.com/rayon-rs/rayon (visited on 12/20/2024).

[17] *Rayon: data parallelism in Rust · baby steps*. URL: https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/ (visited on 11/18/2024).

[18] robert. *rphmeier/jobsteal*. original-date: 2015-11-15T05:40:15Z. 2016. URL: https://github.com/rphmeier/jobsteal (visited on 11/19/2024).

[19] *The Rust Programming Language - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ (visited on 11/19/2024).

[20] Peter Thoman et al. "A taxonomy of task-based parallel programming technologies for high-performance computing". en. In: *The Journal of Supercomputing* 74.4 (Apr. 2018), pp. 1422–1434. ISSN: 1573-0484. DOI: 10.1007/s11227-018-2238-4. URL: https://doi.org/10.1007/s11227-018-2238-4 (visited on 11/04/2024).

[21] Joshua Yanovski et al. "GhostCell: separating permissions from data in Rust". In: *GhostCell: Separating Permissions from Data in Rust (Artifact)* 5.ICFP (Aug. 2021), 92:1–92:30. DOI: 10.1145/3473597. URL: https://dl.acm.org/doi/10.1145/3473597 (visited on 11/18/2024).