A practical analysis of Rust's concurrency story

Aditya Saligrama and Andrew Shen
(and a little bit Jon Gjengset)

PRIMES Computer Science Conference, October 13, 2018

Introduction

- Concurrency is hard.
- High-performance concurrency is harder.
- Fearless concurrency would be nice…

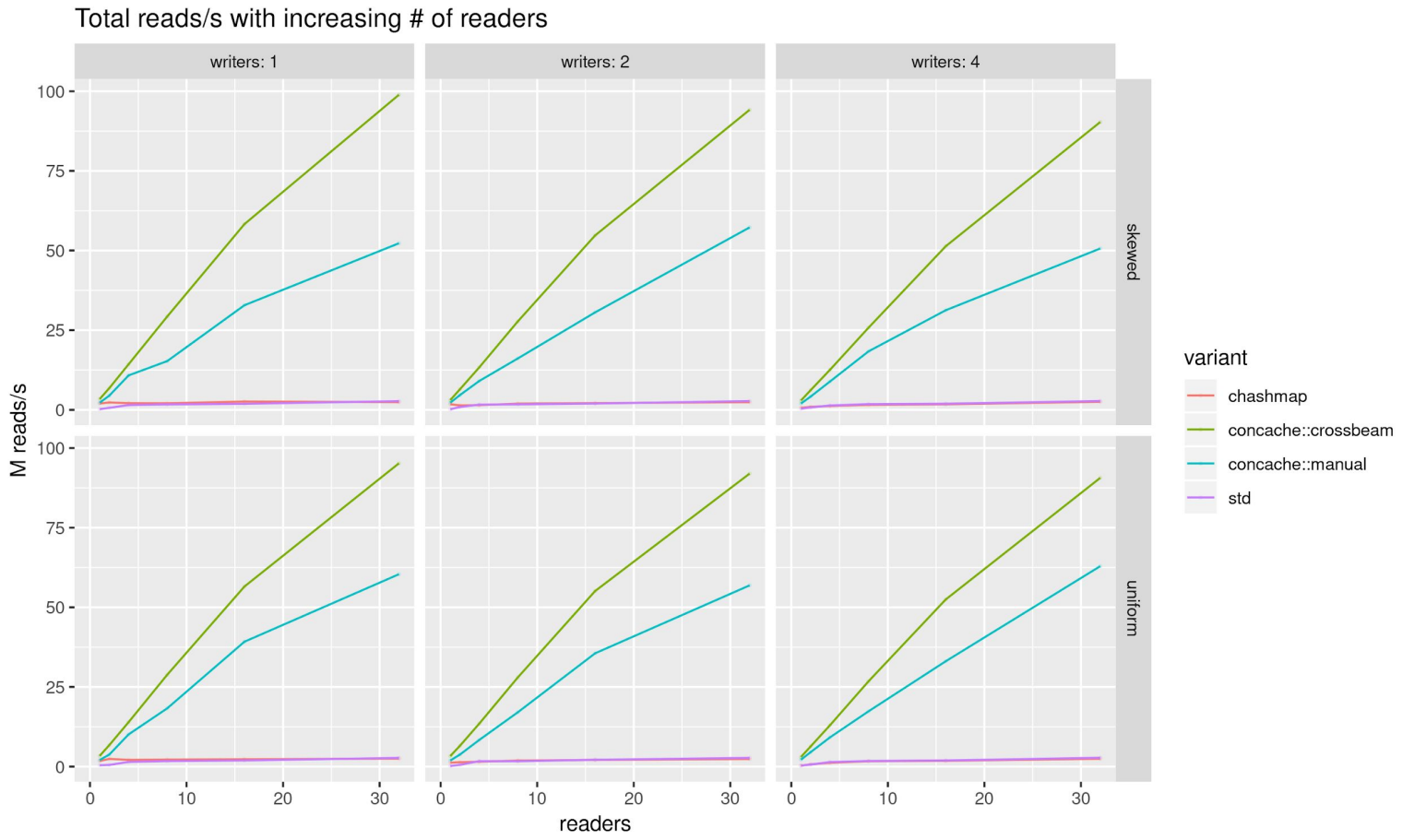
Rust + concurrency = <3?

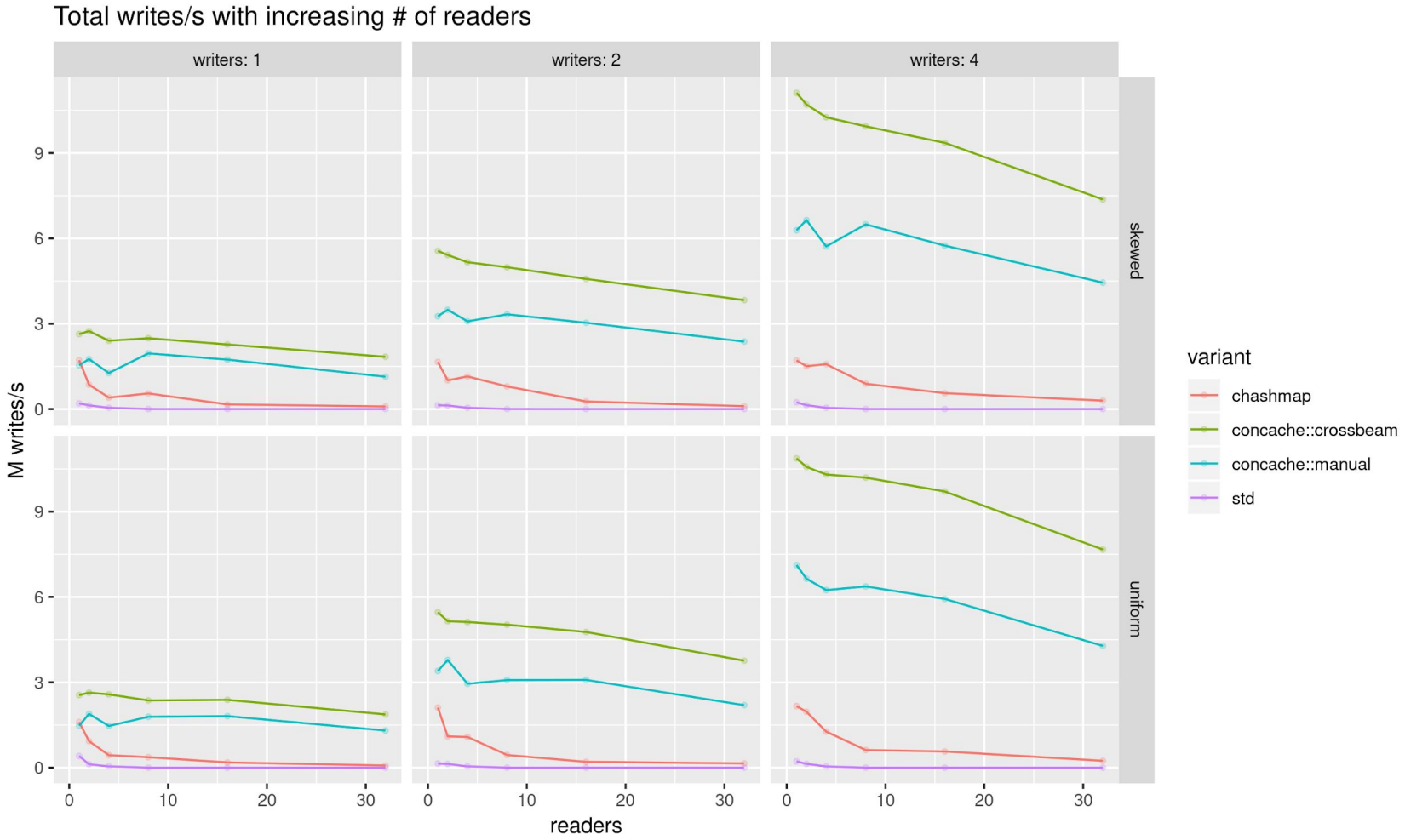
- Rust aims to provide “fearless concurrency”
- For low-level concurrent algorithms too?
- Let’s put that to the test!

Lock-free hashmaps FTW!

- Hashmaps are ubiquitous.
- `Arc<Mutex<HashMap<_, _>>>` anyone?
- *Lock-free* concurrent algorithms fix this…
  …but they're *hard* to get right!

Rust to the rescue?


Total reads/s with increasing # of readers
writers: 1
writers: 2
writers: 4
skewed
uniform
M reads/s
readers
100
75
50
25
0
0
10
20
30
variant
chashmap
concache::crossbeam
concache::manual
std


Total writes/s with increasing # of readers
writers: 1
writers: 2
writers: 4
skewed
uniform
M writes/s
0
3
6
9
0
10
20
30
readers
variant
chashmap
concache::crossbeam
concache::manual
std

```
/// Huh, I guess that *is* a bug…
mod good {
```

Locks are not optional, and that is good!

- Locks wrap the type they protect
- Must go through `Mutex<T>` to access `T`
- Normally locks are like an “honor system"
  - But programmers have no honor…

Look ma', no frees!

```
fn foo() {
    let n = Box::new(5);
    // ...
    // n leaves scope here, the memory is automatically freed
}
```

Borrows Uncover Bugs

- Catches accidental sharing and mutation
- Conditions you to write better code
  - Forces you to think carefully about how your data is accessed.

Pseudocode in C maps well to Rust

- Matches C-style pseudo code closely
- `impl Rust for AcademicPaperAlgo {}`
  - Just copy and paste! (*pfft yeah right...*)

Calling out unsafe code is valuable

- Violate safety restrictions → use `unsafe`
- Marked regions might have bugs!

Safe encapsulation of unsafety is possible!

```
fn get (&self, key : usize) -> Option<usize> {
    let g = epoch::pin(); // open an epoch
    let k = self.head
                .load(Ordering::Relaxed, &g)
                .unwrap();
    // do something with loaded node k
    drop(g); // close the epoch
    // can no longer refer to k
    // the node at self.head can now be freed
}
```

Safe encapsulation of unsafety is possible!

- `crossbeam` library provides *safe* APIs for concurrent operations.
- Let us remove ⅔ of unsafe code + better performance!

```
}

/// No… It *does* live long enough!
mod bad {
```

Auto-Free in an unsafe context

```
fn foo(node: *mut Foo) -> usize {
  unsafe { Box::from_raw(node) }.value

  let x = unsafe { Box::from_raw(node) };
  let v = x.value;
  mem::forget(x); // don't free the Box
  v

  unsafe { &*node }.value
}
```

Tracking pointer modifications

- Encode information in pointers (e.g., low bits)
- Dereferencing == unsafe!
- Who knows if it is intended or accidental?
- Can we solve this with the type system?

```
let x = Box::new([0; 8192]);
let ptr = Box::into_raw(x);
let ptr2: *addr _ = ptr.add_offset(200);
// require specific function for turning *addr -> *mut
// all dereferencing functions take *mut
let z = unsafe { &*ptr2 }; // ERR: ptr2 is *addr!
let w = unsafe { &*ptr }; // OK: ptr is unmodified
let z = unsafe { &*std::mem::declare_valid(ptr2) };

// add to std::mem
fn declare_valid<T>(*addr T) -> *mut T {}
```

Pointers, pointers, oh so many pointers!

- Many choices:
  - `AtomicPtr<T>`, `*mut T`, `&mut T`.
- Differences? Advantages?
- Can combine types too
  - `&mut *mut T`

```
}

/// Why is the compiler yelling at me?
mod ugly {
```

`unwrap()` all the things!

```
fn foo () -> Foo {
    fn_returning_result().unwrap()
}

fn main () {
    do_something(foo());
}
```

```
fn foo () -> Result<Foo> {
    fn_returning_result()?
}

fn main () {
    match foo() {
        Ok(f) => {
            do_something(f);
        },
        Err(e) => { /* … */ }
    }
}
```

Too easy to err on the side of atomics

```
struct Table {
    nbuckets: AtomicUsize, // could just be a usize!
    // ..
}

struct HashMap {
    table: RwLock<Table>,
    // ..
}
```

We're so so tired of `E0597`

```
1 fn main() {
2   struct Foo<'a> {
3     x: Option<&'a u32>,
4   }
5
6   let mut x = Foo { x: None };
7   let y = 0;
8   x.x = Some(&y);
9 }
```

```
error[E0597]: `y` does not live long enough
 --> src/main.rs:8:17
  |
8 |     x.x = Some(&y);
  |                 ^ borrowed value does
not live long enough
9 | }
  | - `y` dropped here while still borrowed
  |
  = note: values in a scope are dropped in
the opposite order they are created
```

Types get 2complicated2fast.

```
HashMap<
  String,
  Arc<Mutex<HashMap<
    String,
    HashMap<usize, usize>
  >>>
>
```

What the heck is Ordering?

- All Atomic Functions require `Ordering`
  - `SeqCst`, `Relaxed`, etc.
- These are poorly explained and confusing
  - But also poorly explained in C!

Compiler likes suggesting adding lifetimes

```
fn search() -> &Node {
    let n = Node::new();
    // ...
    &n
}

fn main() {
    search();
}
```

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:1:16
  |
1 | fn search() -> &Node {
  |                ^ expected lifetime parameter
  |
  = help: this function's return type contains a
borrowed value, but there is no value for it to be
borrowed from
  = help: consider giving it a 'static lifetime
```

```
}

/// Thank you!
/// github.com/saligrama/concache
/// Aditya: saligrama.io
/// Andrew: shenandrew95@gmail.com
mod questions {
```

```
} // <- we didn't forget!
```