Source and Buggy · Follow publication

Data-driven performance optimization with Rust and Miri

Or, how I helped Santa's Elves find their badges 9 times faster

11 min read · Dec 8, 2022

Keaton Brandt Follow

There's something about Rust that makes it uniquely fun to write. It feels like a wooden jigsaw puzzle: sometimes tedious, sometimes frustrating, but always immensely satisfying when the last piece drops into place. Other languages can feel more like soggy cardboard puzzles where the pieces need to be forcibly smooshed together and you're never 100% sure if you got it right.

I chose Rust for Advent of Code this year both because it's fun and because it seems like the language is (finally) gaining momentum in the industry. There are so many great things about Rust: memory-safety, documentation, error messages, its built-in package manager, and so on. It has some shortcomings too, but it's improving all the time. The bigger issues for me are the tools *around* the language. Plenty of tools "work" with Rust by shoehorning it into C/C++ infrastructure (eg. LLDB, Valgrind, Clion). The results are often more kludgy than they're worth.

The most surprising thing for me is how unintuitive it is to optimize Rust code given that it's honestly hard to find a Rust project that doesn't loudly strive to be "blazingly fast". No language is intrinsically fast 100% of the time, at least not when a mortal like me is behind the keyboard. It takes work to optimize code, and too often that work is guess-and-check.

So I'm taking a break from my usual "hot takes" today to focus on something a little more useful: a guide to doing meaningful, data-driven performance analysis with pure Rust tools.

Let's help some elves

For this exercise I'm solving Advent of Code 2022, Day 3, Part 2. You can read the illustriously-contrived Christmas-themed flavor text at the link above. It involves elves and their rucksacks. To summarize the problem more dryly, you are provided input that looks like this:

```
vJrwpWtwJgWrhcsFMMfFFhFp
jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL
PmmdzqPrVvPwwTWBwg
wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn
ttgJtRGJQctTZtZT
CrZsJsPPZsGzwwsLwLmpwMDw
```

The input is meant to be read in groups of 3 lines.

```
vJrwpWtwJgWrhcsFMMfFFhFp
```

```
jqHRNqRjqzjGDLGLrsFMfFZSrLrFZsSL    ┐ Group One
PmmdzqPrVvPwwTWBwg                  ┘
wMqvLMZHhHMvwLHjbvcjnnSBnvTQFn      ┐
ttgJtRGJQctTZtZT                    ┘ Group Two
CrZsJsPPZsGzwwsLwLmpwMDw            ┘
```

Each group has exactly one character that appears in all 3 lines (`r` for Group One and `Z` for Group Two). Each character has a 'priority' score defined as:

- Lowercase item types `a` through `z` have priorities 1 through 26.
- Uppercase item types `A` through `Z` have priorities 27 through 52.

The goal is to parse a file full of 3-line groups and return the sum of the priority scores for every group's matching character. I'm not sure this has any practical applications but it's fun and festive and can be implemented very efficiently, making it a good test case for optimization skills.

So I went all-out on optimizing my solution. I used bit sets to efficiently find the common letter for all 3 lines using a library called FixedBitSet. Then I build a custom parser with nom to read the input file, instead of relying on string splitting and regexes. I made sure avoid heap allocations where possible by using iterators and arrays instead of `Vec` objects. After all that, my solution ran in 1.64 milliseconds — which, in Rust terms, is really not great. Redditors had solutions ~40 times faster with less code. What gives?

The scary part was, I had no idea! All of my code looked perfectly efficient to me. A Rust expert could probably pick out plenty of problems, but I'm still learning — so, I went down a rabbit-hole of performance profiling in search of that holiest of virtues: blazingly-fast speed.

The state of Rust profilers

Profilers are tools that analyze how well a piece of software is performing. This isn't as easy as it sounds, since *measuring* software performance *affects* software performance. The CPU has to do extra work while the software is running to time different operations and save those results to memory.

There's no one universal profiler that does everything — each one makes different tradeoffs and provides different insights, so we'll have to be more

specific.

The tracing library and its various backends are great for providing high-level real-time feedback about how different subcomponents are performing — but can't realistically analyze down to the level of individual lines of code. In other words, it can tell you *that* there's a problem, but not necessarily *where it is*. This is great for production code since it's low impact, but won't help me here.

Valgrind is closer to what I was looking for because it instruments every line of code to check for memory leaks and performance problems, but it's awkward to use with Rust and doesn't work well on macOS. Plus its UI is terrible (I've previously written about how good UI is critically important even for galaxy-brained developers).

Enter Flamegraph

Perf-based libraries like flamegraph work by sampling the stack trace of your software on a regular interval. With a large enough sample size this can reveal which lines of code are most likely to be executing at any given time. In order to get any useful results I had to run my Advent of Code solution 1000 times in a loop, which is certainly not elegant — but the results are pretty useful!

The color scheme takes the name “flame graph” a bit too literally

Ok, at a high level this is pretty inscrutable. The X axis of this graph isn't linear time, it's — err, I'm not sure actually. I think it's time but rearranged such that the graph forms nice shapes. The Y axis represents the call stack. The relative width of each bar indicates how much of the runtime is taken up by a particular stack trace.

We can also see `FixedBitSet::insert` taking a lot of execution time all by itself, which is bizarre given that bit sets are basically the most computationally-efficient data structures ever invented. It turns out FixedBitSet is actually not a zero cost abstraction and instead adds quite a lot of overhead. So I implemented my own extremely minimal bit set (which *is* zero-cost, not to brag).

```
#[derive(Debug)]
struct RucksackBitSet(u64);

impl RucksackBitSet {
    fn add(&mut self, value: u8) {
        // Given that the input will never exceed 57 in practice I can
        // reasonably skip bounds checking here.
        self.0 |= 1 << value;
    }

    fn intersect(&mut self, other: Self) -> &mut Self {
        self.0 &= other.0;
        self
    }

    fn get_first(&self) -> u8 {
        self.0.trailing_zeros() as u8
    }
}
```

With this change we're down to 589 microseconds, a 2.7x speedup from where we started! But, that's really as much as I can glean from flamegraph, and I'm not feeling *blazingly fast* yet.

Enter Miri

Miri is a fascinating project owned by the Rust team that aims to run Rust code in an interpreter. On the surface, that's a deeply silly thing to do. Interpreted Rust

has all the annoying quirks of a compiled language combined with all the terrible performance characteristics of Python. It's the worst of both worlds. Luckily, it's not intended to run any production code — it's intended to find bugs and performance problems during development. Miri is, effectively, Rust's answer to Valgrind.

You may have heard of Miri, but I bet you didn't know it could export detailed profiling information to the Chrome Dev Tools. The docs only refer to that feature offhand and don't really describe how to do it. I get the sense that this is all still very experimental — but it worked for me and it might work for you too!

All you have to do is install it. Here's what I did on my mac, using `rustup`. It's probably the same on other Unix platforms:

```
rustup +nightly component add miri;
cargo install --git https://github.com/rust-lang/measureme --branch stable c
```

You'll also need Google Chrome installed to interact with the output.

Then I run my code like this:

```
MIRIFLAGS="-Zmiri-disable-isolation -Zmiri-measureme=crox" cargo +nightly mi
crox crox.mm_profdata;
```

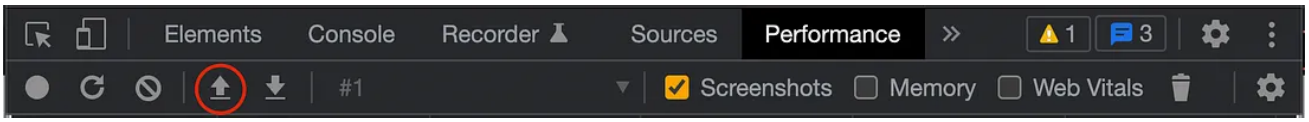
One thing to keep in mind is that Miri is *slow*. My 1.6 millisecond solution took a full 11 seconds to complete with Miri — around 7000 times slower than native performance.

> *Important note: Miri is not intended to accurately replicate optimized Rust runtime code. Optimizing for Miri can sometimes make your real code slower, and vice versa. It's a helpful tool to guide your optimization, but you should always benchmark your changes with release builds, not with Miri.*

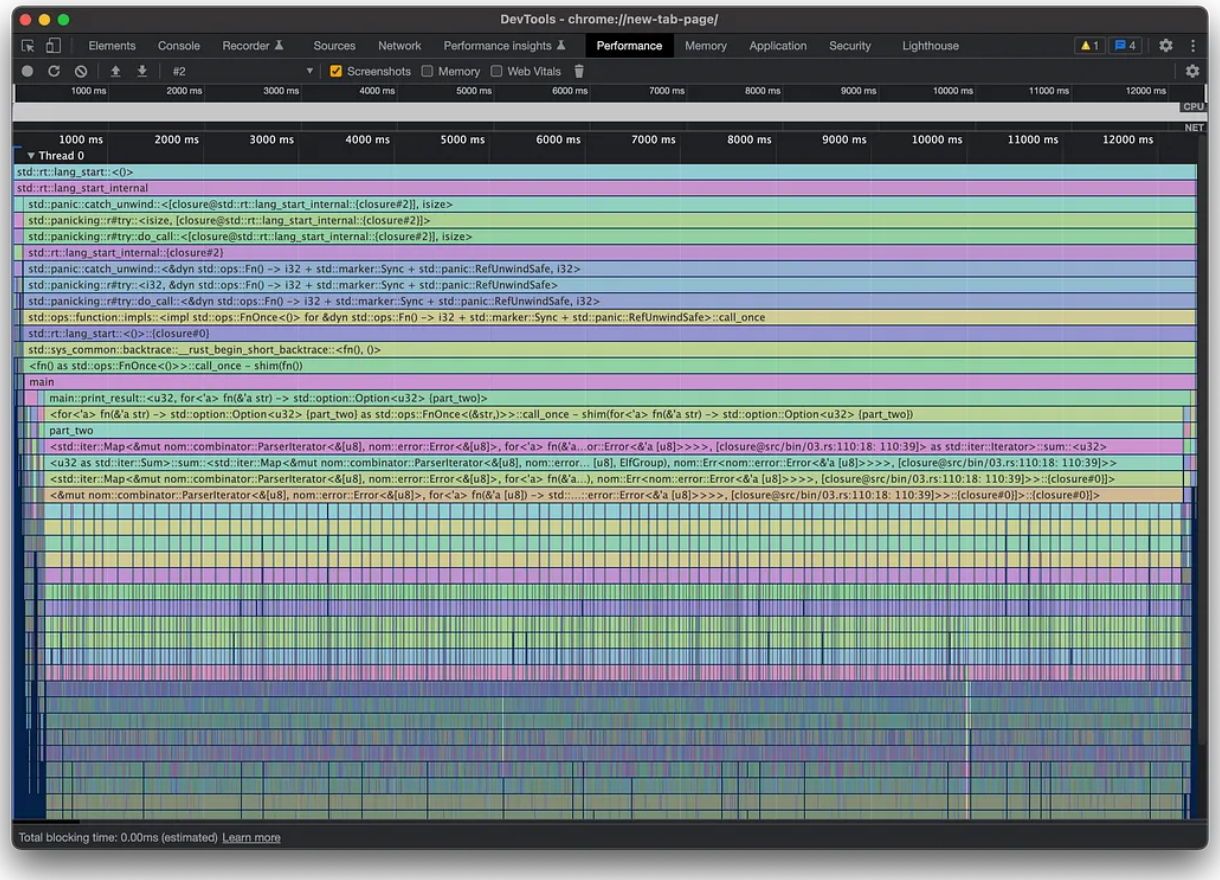
Once execution is finished, the `crox` command creates a file called

`chrome_profiler.json`. I haven't found an automated way of opening this file in Chrome's Performance inspector, so this next part unfortunately involves some clicking. Navigate to any webpage in Google Chrome and open the developer tools (`Option + ⌘ + J` on Mac, `Shift + CTRL + J` on Windows and Linux). I like to pop it out into its own window (from the three-dot menu at the top right) to get some breathing room.

Next, navigate to the Performance tab at the top and click on the little up-arrow button, which is labeled "Load profile".

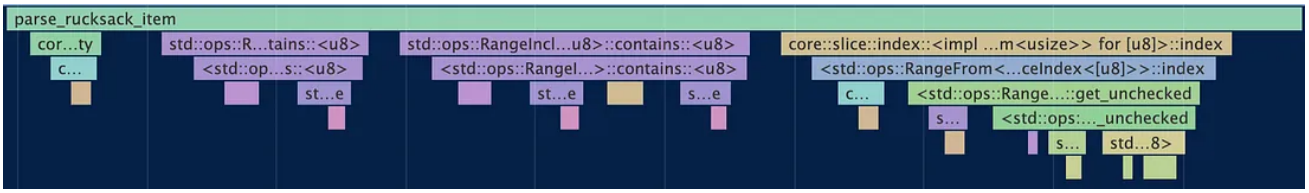


And now comes a little bit more waiting while Chrome parses the JSON file, which in my case clocks in at a whopping 100 megabytes. After a few seconds I'm presented with this surprisingly beautiful timeline chart.

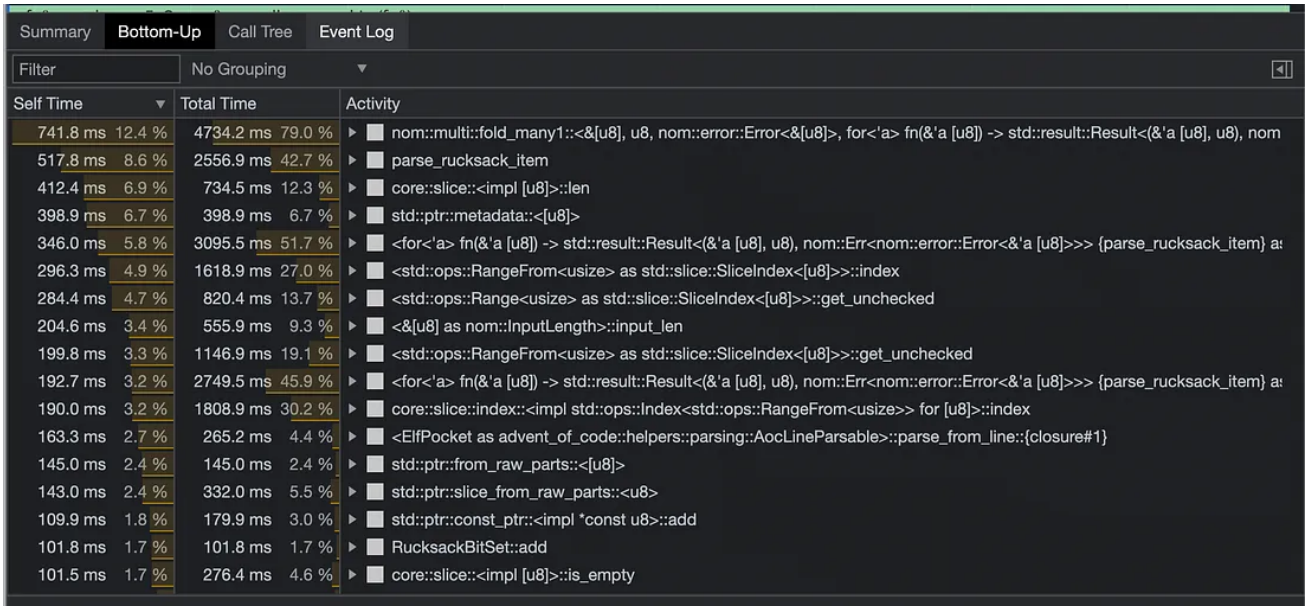


That's a lot of data!

Unlike flamegraph this chart is fully zoom-able — and the X axis is actually just plain old linear time. Every bar is a function call, with the bars underneath it representing the functions *it* calls. So here you can see something curious: the second call from `parse_rucksack_item` to `RangeInclusive::contains::<u8>` is slower than the first.



I have no idea why, but it's interesting! And that's the kind of nuance we missed with flamegraph. Chrome's Dev Tools also provide an interactive bottom-up call tree, which shows which functions take the most 'self-time' — ie. time executing the function's own code, rather than calling into other functions.

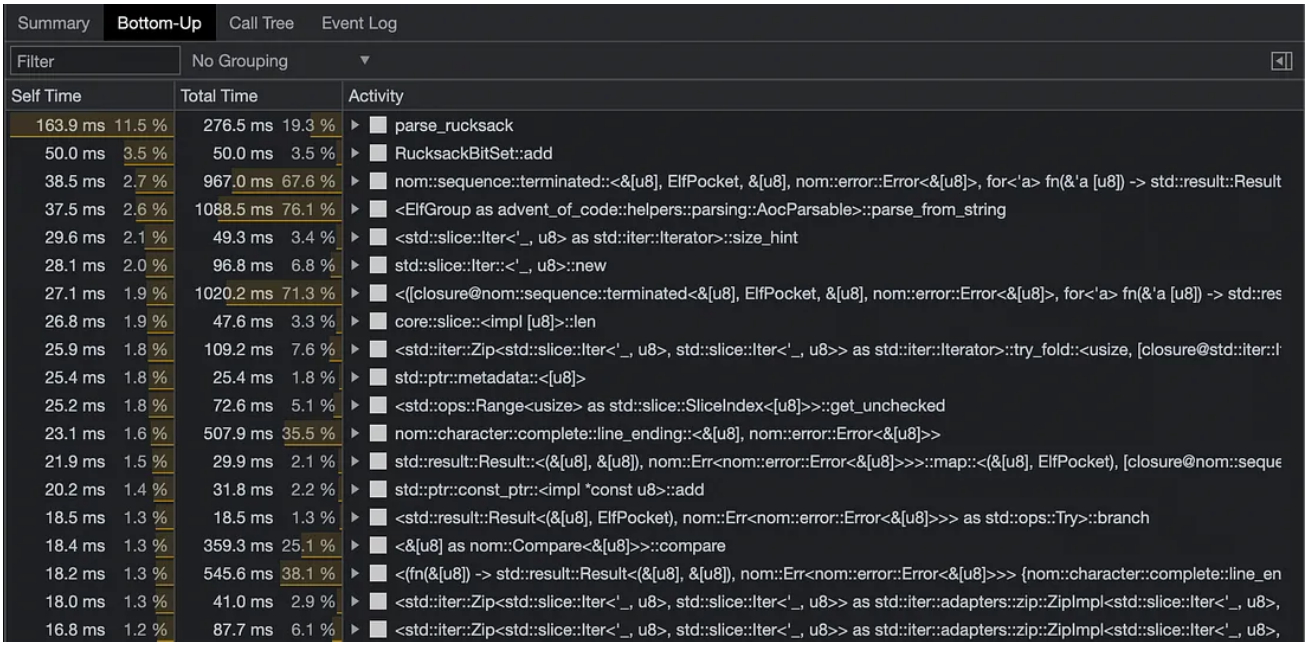


So, back to optimizing our convoluted elf problem! Two things stand out here: nom's `fold_many1` function has a lot of overhead, as does the array slicing operation required for `nom` parsing. These are both consequences of the fact that `parse_rucksack_item` takes a list of input characters and returns a tuple containing the result of parsing the first character and a list of all the remaining characters. I had hoped Rust's compiler could work its inlining, bit-twiddling magic to make this efficient, but alas it couldn't. So, I replaced `parse_rucksack_item` with a slightly-less-elegant function called `parse_rucksack` that parses every character in

the input until it reaches a newline.

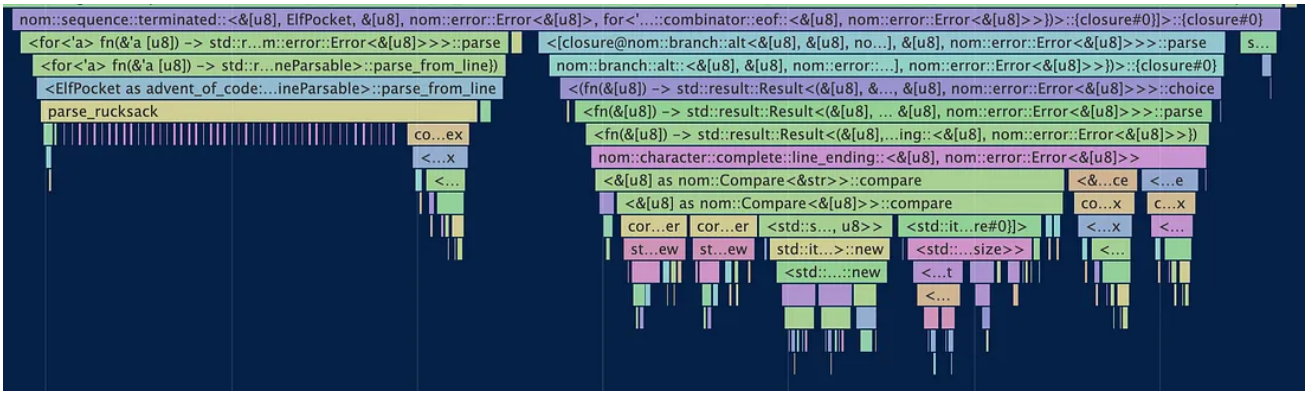
```
fn parse_rucksack(
    input: &[u8],
) -> Result<(&[u8], RucksackBitSet), ParsingError> {
    if input.is_empty() {
        return generic_error_for_input(input);
    }

    let mut rucksack_bit_set = RucksackBitSet::new();
    let mut i = 0;
    while input[i] != b'\n' {
        let c = input[i];
        if c >= b'a' && c <= b'z' {
            rucksack_bit_set.add(c - b'a' + 1);
        } else if c >= b'A' && c <= b'Z' {
            rucksack_bit_set.add(c - b'A' + 27);
        } else {
            return generic_error_for_input(input)
        }
        i += 1;
    }

    return Ok((&input[i..], rucksack_bit_set));
}
```

That trimmed off another 100 microseconds from the optimized build! What's left?

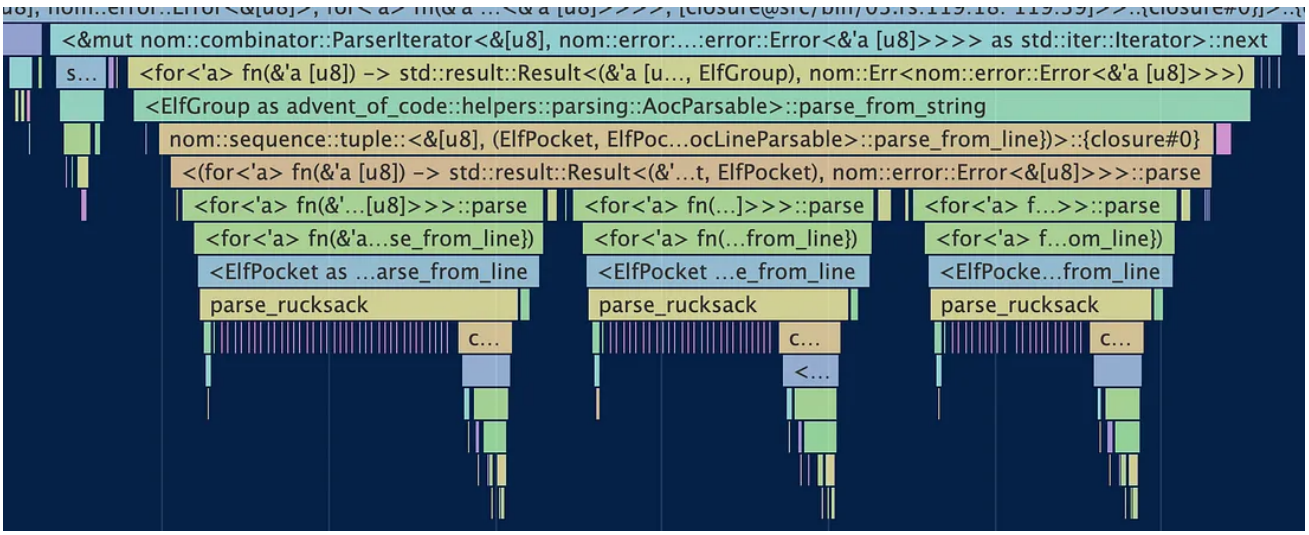


The only particularly-expensive single function in the entire profile is `parse_rucksack`, which is our core business logic. So that's good! But it's not the full story.

"Self Time" is not a perfect indicator of performance impact, which we can clearly see by looking at the timeline chart. `parse_rucksack` stands out not because it's the most time-consuming operation, but because it does so much processing without calling out to another function. `nom::character::complete::line_ending`, which just checks if a character is a line ending, turns out to be even more expensive overall.



Luckily, our new `parse_rucksack` function is already capable of finding those newline characters itself! All we have to do is update it to exclude the newline character from the remainder by returning `&input[i + 1..]` instead of `&input[i..]` and we can eliminate the `line_ending` function entirely. Now our timeline chart is looking pretty excellent:



About half of the total time is spent just doing the core business logic of parsing rucksacks. The full runtime clocks in at 181 microseconds, which is approximately **9 times faster** than what we started with! We can see that `nom` is still adding some overhead, but not an unreasonable amount given that it makes our code more maintainable and debuggable. So yes, my solution is still slower than Reddit's, but at least I understand why and can make informed decisions about which optimizations are worthwhile and which aren't.

This is about as far as Miri can go. I'd love to find a way to profile individual lines of code in the `parse_rucksack` function to eke out even more performance, but I don't know of any tools that can do that, so I'll leave it here. Here's my final code.

Closing thoughts

This wouldn't be a Source and Buggy post if I didn't spend at least some of it complaining about IDEs. As we've seen, performance profiling tools like Miri are extremely helpful — so why did I have to jump through 2 command-line utilities and a *web browser* to use it? Why isn't there a button in my IDE that runs this kind of performance simulation and puts a big red highlight over all the lines that are unusually slow?

I guess this is the hill I'm destined to die on: developers need better user experiences. Every time you click a blog post like this one you're proving that the tools we currently have are not sufficiently intuitive. I shouldn't have to tell you how to profile your code because it should be an easy click away. It should be as easy as running a unit test or renaming a variable. The same goes for debugging code, analyzing data flow, identifying resource contention, etcetera. In my opinion this is lacking in all languages, but it's especially lacking in Rust.

But enough negativity! Miri and Flamegraph are both really handy tools that helped me make big strides in my code's performance. Alas, I've wasted so much time over-optimizing this problem that I'm three days behind on Advent of Code. I must go now — the elves need me.

Disclaimer: I recommend tools related to Google Chrome in this article. I am an

active Google employee, although I do not work on Chrome.

Rust | Profiling | Software Engineering | Optimization | Advent Of Code

Follow

Published in Source and Buggy

85 followers · Last published Jul 23, 2023

A blog about the old-fashioned ideas behind our sci-fi world.

Follow

Written by Keaton Brandt

1.1K followers · 265 following

Senior Software Engineer at Google (but views are my own). Seattlite. Chihuahua chauffeur. Doomscrolls on Wikipedia.

Responses (5)

Write a response

What are your thoughts?

Abdelfattah Sekak
Sep 17, 2023