Rust: allow multiple threads to modify an image (wrapper of a vector)?

Asked 1 year, 2 months ago

Active 1 year, 2 months ago

Viewed 309 times

3

Suppose I have a "image" struct that wraps a vector:

```
type Color = [f64; 3];

pub struct RawImage
{
    data: Vec<Color>,
    width: u32,
    height: u32,
}

impl RawImage
{
    pub fn new(width: u32, height: u32) -> Self
    {
        Self {
            data: vec![[0.0, 0.0, 0.0]; (width * height) as usize],
            width: width,
            height: height
        }
    }

    fn xy2index(&self, x: u32, y: u32) -> usize
    {
        (y * self.width + x) as usize
    }
}
```

It is accessible through a "view" struct, which abstracts an inner block of the image. Let's assume that I only want to write to the image (`set_pixel()`).

```
pub struct RawImageView<'a>
{
    img: &'a mut RawImage,
    offset_x: u32,
    offset_y: u32,
    width: u32,
    height: u32,
}

impl<'a> RawImageView<'a>
{
    pub fn new(img: &'a mut RawImage, x0: u32, y0: u32, width: u32, height: u32) -> Self
    {
        Self{ img: img,
              offset_x: x0, offset_y: y0,
              width: width, height: height, }
    }

    pub fn set_pixel(&mut self, x: u32, y: u32, color: Color)
    {
        let index = self.img.xy2index(x + self.offset_x, y + self.offset_y);
        self.img.data[index] = color;
    }
}
```

Now suppose I have an image, and I want to have 2 threads modifying it at the same time. Here I use rayon's scoped thread pool:

```
fn modify(img: &mut RawImageView)
{
    // Do some heavy calculation and write to the image.
    img.set_pixel(0, 0, [0.1, 0.2, 0.3]);
}

fn main()
{
    let mut img = RawImage::new(20, 10);
    let pool = rayon::ThreadPoolBuilder::new().num_threads(2).build().unwrap();
    pool.scope(|s| {
        let mut v1 = RawImageView::new(&mut img, 0, 0, 10, 10);
        let mut v2 = RawImageView::new(&mut img, 10, 0, 10, 10);
        s.spawn(|_| {
            modify(&mut v1);
        });
        s.spawn(|_| {
            modify(&mut v2);
        });
    });
}
```

This doesn't work, because

1. I have 2 `&mut img` at the same time, which is not allowed
2. "closure may outlive the current function, but it borrows `v1` , which is owned by the current function"

So my questions are

1. How can I modify `RawImageView`, so that I can have 2 threads modifying my image?
2. Why does it still complain about life time of the closure, even though the threads are scoped? And how do I overcome that?

Playground link

One approach that I tried (and it worked) was to have `modify()` just create and return a `RawImage`, and let the thread push it into a vector. After all the threads were done, I constructed the full image from that vector. I'm trying to avoid this approach due to its RAM usage.

multithreading rust lifetime interior-mutability

Share
Improve this question
Follow
edited Oct 9 '20 at 7:10

asked Oct 9 '20 at 6:29

MetroWind
467 ● 3 ● 16

Add a comment

2 Answers

Active Oldest Votes

1

✓

Your two questions are actually unrelated.

First the **#2** that is easier:

The idea of the Rayon scoped threads is that the threads created inside cannot outlive the scope, so any variable created *outside* the scope can be safely borrowed and its references sent into the threads. But your variables are created *inside* the scope, and that buys you nothing.

The solution is easy: move the variables out of the scope:

```
    let mut v1 = RawImageView::new(&mut img, 0, 0, 10, 10);
    let mut v2 = RawImageView::new(&mut img, 10, 0, 10, 10);
    pool.scope(|s| {
        s.spawn(|_| {
            modify(&mut v1);
        });
        s.spawn(|_| {
            modify(&mut v2);
        });
    });
```

The **#1** is trickier, and you have to go unsafe (or find a crate that does it for you but I found none). My idea is to store a raw pointer instead of a vector and then use `std::ptr::write` to write the pixels. If you do it carefully and add your own bounds checks it should be perfectly safe.

I'll add an additional level of indirection, probably you could do it with just two but this will keep more of your original code.

The `RawImage` could be something like:

```
pub struct RawImage<'a>
{
    _pd: PhantomData<&'a mut Color>,
    data: *mut Color,
    width: u32,
    height: u32,
}
impl<'a> RawImage<'a>
{
    pub fn new(data: &'a mut [Color], width: u32, height: u32) -> Self
    {
        Self {
            _pd: PhantomData,
            data: data.as_mut_ptr(),
            width: width,
            height: height
        }
    }
}
```

And then build the image keeping the pixels outside:

```
    let mut pixels = vec![[0.0, 0.0, 0.0]; (20 * 10) as usize];
    let mut img = RawImage::new(&mut pixels, 20, 10);
```

Now the `RawImageView` can keep a non-mutable reference to the `RawImage`:

```
pub struct RawImageView<'a>
{
    img: &'a RawImage<'a>,
    offset_x: u32,
    offset_y: u32,
    width: u32,
    height: u32,
}
```

And use `ptr::write` to write the pixels:

```
    pub fn set_pixel(&mut self, x: u32, y: u32, color: Color)
    {
        let index = self.img.xy2index(x + self.offset_x, y + self.offset_y);
        //TODO! missing check bounds
        unsafe { self.img.data.add(index).write(color) };
    }
```

But do not forget to either do check bounds here or mark this function as unsafe, sending the responsibility to the user.

Naturally, since your function keeps a reference to a mutable pointer, it cannot be send between threads. But we know better:

```
unsafe impl Send for RawImageView<'_> {}
```

And that's it! Playground. I think this solution is memory-safe, as long as you add code to enforce that your views do not overlap and that you do not go out of bounds of each view.

Share
Improve this answer
Follow
edited Oct 10 '20 at 10:29

answered Oct 9 '20 at 7:37
rodrigo
83.3k ● 10 ● 129 ● 170

Thanks you for the solution! Very interesting! I tried to use raw pointers, but I put it in the `RawImageView`. It didn't work because raw pointers are not Sync and Send. But I never thought of letting `RawImage` having the pointer.
– MetroWind
Oct 9 '20 at 16:49

@MetroWind: Actually I think you could add the raw pointer to `RawImageView` and make it work, I had to manually implement `Send` anyway. I did this way because your `xy2index` is in `RawImage`, but that can be easily moved.
– rodrigo
Oct 9 '20 at 19:23

« I think this solution is memory-safe, as long as your views do not overlap » So this is absolutely **not** safe, or it is just as safe as C, C++... The idea behind safety in Rust is that **the language itself** guaranties that you cannot have simultaneous write or read-write access to the same piece of data (most of the time it's checked at compilation and sometimes at runtime). In this suggested solution, this is the responsibility of its user (not even its designer) to take care of that; it looks to me like an exact illustration of an anti-pattern in Rust concerns.
– prog-fh
Oct 10 '20 at 7:53

@prog-fh: You are right, of course. What I meant is that this solution is memory-safe... as long as you *add the necessary code to ensure that the views do not overlap*, and the code to ensure that the pixels do not go out of bounds... This code is just a sketch of the working solution, not the final code.
– rodrigo
Oct 10 '20 at 10:28

@prog-fh Just saw your comment. And you are absolutely right in that this is only as safe as C/C++. However it is absolutely necessary in numerical calculations to be able to logically break a piece of continuous memory in arbitrary ways, without copying, and to simultaneously write to them. One could hide it in a library and pretend it's "safe", but the keyword is "pretend". If this is an anti-pattern, at least it's a necessary anti-pattern.
– MetroWind
Dec 2 '20 at 4:17

Show **1** more comment

▲
0
▼

This does not exactly match your image problem but this might give you some clues.

The idea is that chunks_mut() considers a whole mutable slice as many independent (non-overlapping) mutable sub-slices. Thus, each mutable sub-slice can be used by a thread without considering that the whole slice is mutably borrowed by many threads (it is actually, but in a non-overlapping manner, so it is sound).

Of course this example is trivial, and it should be trickier to divide an image in many arbitrary non-overlapping areas.

```
fn modify(
    id: usize,
    values: &mut [usize],
) {
    for v in values.iter_mut() {
        *v += 1000 * (id + 1);
    }
}

fn main() {
    let mut values: Vec<_> = (0..8_usize).map(|i| i + 1).collect();

    let pool = rayon::ThreadPoolBuilder::new()
        .num_threads(2)
        .build()
        .unwrap();
    pool.scope(|s| {
        for (id, ch) in values.chunks_mut(4).enumerate() {
            s.spawn(move |_| {
                modify(id, ch);
            });
        }
    });

    println!("{:?}", values);
}
```

Edit

I don't know the context in which this parallel work on some parts of an image is needed but I can imagine two situations.

If the intent is to work on some arbitrary parts of the image and allow this to take place in several threads that compute many other things, then a simple mutex to regulate the access to the global image is certainly enough. Indeed, if the precise shape of each part of the image is very important, then it is very unlikely that there exist so many of them that parallelisation can be beneficial.

On the other hand, if the intent is to parallelize the image processing in order to achieve high performance, then the specific shape of each part is probably not so relevant, since the only important guaranty to ensure is that the whole image is processed when all the threads are done. In this case a simple 1-D splitting (along y) is enough.

As an example, below is a minimal adaptation of the original code in order to make the image split itself into several mutable parts that can be safely handled by many threads. No unsafe code is needed, no expensive runtime checks nor copies are performed, the parts are contiguous so highly optimisable.

```
type Color = [f64; 3];

pub struct RawImage {
    data: Vec<Color>,
    width: u32,
    height: u32,
}

impl RawImage {
    pub fn new(
        width: u32,
        height: u32,
    ) -> Self {
        Self {
            data: vec![[0.0, 0.0, 0.0]; (width * height) as usize],
            width: width,
            height: height,
        }
    }

    fn xy2index(
        &self,
        x: u32,
        y: u32,
    ) -> usize {
        (y * self.width + x) as usize
    }

    pub fn mut_parts(
        &mut self,
        count: u32,
    ) -> impl Iterator<Item = RawImagePart> {
        let part_height = (self.height + count - 1) / count;
        let sz = part_height * self.width;
        let width = self.width;
```

Share
Improve this answer
Follow

edited Oct 9 '20 at 20:10

answered Oct 9 '20 at 7:37

prog-fh
8,988 ● 1 ● 5 ● 24

Thanks for the answer! Yeah. That's what I'd do if my data can be splitted in a 1-D manner. I guess the question is how to implement a 2D version of this.
– MetroWind
Oct 9 '20 at 17:18

Add a comment

Your Answer

Post Your Answer

By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged multithreading rust lifetime interior-mutability or ask your own question.

The Overflow Blog

- Sequencing your DNA with a USB dongle and open source code
- Don't push that button: Exploring the software that flies SpaceX rockets and...

Featured on Meta

- Providing a JavaScript API for userscripts
- Congratulations to the 59 sites that just left Beta

Related

3
Rust lifetimes - returning value from vector in RWLock in Arc

How to sort a vector in Rust?

Rust mpsc::Sender cannot be shared between threads?

Sharing String between threads in Rust

Store data that implements a trait in a vector

Rust - writing to indices of a vector across multiple threads

Rust lifetime issue with manually dropped reference wrapper

Hot Network Questions

Who (or what) created the atropal?

How to force Mathematica to do infinite-precision calculations?

Dataframe from a character vector where variable name and its data were stored jointly

Why are nerves blocked even though potassium chanels are not blocked?

which one of these paths has the priority: /usr or /usr/local

more hot questions

Question feed