YOSHUA WUYTS ←

TASKS ARE THE WRONG ABSTRACTION — 2024-04-27

Okay, so you've probably read the title and are thinking: *"Hey is this clickbait?"* - and no dear reader, it is not. I'm fairly convinced at this point that tasks, as an abstraction, are not right for Rust and we would do better to achieve parallel execution through composition. However when it comes to *task scheduling* things seem less clear, and I would like to have more data available. Because task scheduling strategies determine which APIs we can provide, and as we're looking to standardize the async Rust ecosystem, this will end up being important.

So okay, big claim - *"Tasks are not the wrong abstraction"*. Tasks are a staple in async Rust (and I should know), and task spawning has been around pretty much since the beginning. However, two relatively recent developments have made me start to question whether tasks themselves actually carry their weight, both for single-threaded concurrent workloads and parallel multi-threaded workloads:

1. Both Bytedance's (TikTok's) `monoio` crate and the `glommio` crate use a thread-per-core architecture and appear to scale significantly better than Tokio's work-stealing architecture on

networking benchmarks. **This raises questions about tasks as primitives for parallel async execution.** (edit: I mean "questions" literally here - these are interesting claims which we should research more closely. We get into that more later on in this post).

2. Single-threaded executors introduce `'static` lifetimes, often do not correctly propagate errors and cancellation, and in recent testing have shown to perform up to 2-3x worse than their structured counterparts. We have alternative APIs available for concurrency which do not have these issues. **This raises questions about tasks as primitives for concurrent async execution.**

In this post I want to discuss task spawning for just concurrent but also parallel execution. I want to show some of the issues both of these approaches run into, show how we can do better, and talk about what we need more data on. Finally I want to speculate a little about where we could go with async parallelism and concurrency in Rust. But to save everyone some reading, here's some code roughly summarizing the first half of this post [1]:

```
// concurrent execution of two futures today (structured)
let a = async { 1 };              // ← does nothing until .
let b = async { 2 };              // ← does nothing until .
let (a, b) = (a, b).join().await; // ← concurrent `.await`

// parallel execution of two futures today (unstructured)
let a = async_std::spawn(async { 1 });  // ← begins paralle
let b = async_std::spawn(async { 2 });  // ← begins paralle
let a = a.await;                        // ← await order do
let b = b.await;                        // ← await order do

// parallel execution of two futures as proposed (structure
let a = async { 1 }.par();         // ← does nothing until
let b = async { 2 }.par();         // ← does nothing until
let (a, b) = (a, b).join().await;  // ← parallel `.await`
```

PREFACE: CONCURRENT AND PARALLEL EXECUTION

Note (2024-04-27): this section was added after repeated questions about the difference between parallelism and

concurrency.

In order to make sense of this post, it's important to understand the differences between parallelism and concurrency, as well as parallel execution and concurrent execution. These are related but distinct terms, and it can take some time to internalize. My favorite definition of the differences between these comes from Aaron Turon's PhD thesis:

> *Concurrency is a system-structuring mechanism, parallelism is a resource*.

Put concretely: "concurrency" refers to the way we schedule work. While "parallelism" refers to e.g. the amount of cores a computer has. If we want to perform *parallel execution of work*, we have to schedule work concurrently using the system's resources for parallelism. We can plot the relationship of parallelism and concurrency in a 2x2 table:

	no parallelism	**has parallelism**
sequential scheduling	sequential execution	sequential execution
concurrent scheduling	concurrent execution	parallel execution

This table is probably going to surprise some folks. What we're seeing here is that even if we use multiple threads, it's still possible to achieve *sequential* execution. How can that be? Well dear reader, imagine some exclusive resource shared between N threads. In order for any thread to make progress they must take an exclusive lock out on that resource. That would certainly make use of multiple threads; but execution would be entirely sequential - only one thread can make progress at any given moment. In order to achieve *parallel execution* it's not enough to just make use of *parallelism*; we also have to schedule concurrently.

Conversely it's also possible to schedule work concurrently despite not having access to any parallelism. An example of this is for example: race two timers with each other. We're waiting on both at the same time, despite not having multiple threads. This is an example of *concurrent execution* without any access to resources for parallelism.

TASKS ARE THE WRONG ABSTRACTION FOR PARALLEL ASYNC EXECUTION

Cq. 2019 I helped popularize the reasoning that: *"Tasks should function like the `async/.await` equivalent of threads"*. Among other things that meant that in the `runtime` crate and subsequently `async-std` we made sure that tasks always returned `JoinHandle` s, and that those handles could be awaited to obtain values. Prior to that it was common to manually create async one-shot channels to obtain values from tasks (src [2]):

```
//! What using tasks was like cq. December 2018

use std::thread;
use futures::{executor, channel::oneshot};

fn main() {
    let mut handles = vec![];

    for _ in 0..10 {
        let (sender, receiver) = oneshot::channel();
        handles.push(receiver);
        juliex::spawn(async move {
            let id = thread::current().id();
            sender.send(id);
        })
    }

    for handler in handles {
        let id = handler.await; // this was actually `await
        println!("handler returned from thread: {id:?}");
    }
}
```

That rationale that *"tasks are like async threads"* has stuck around, and I think it is wrong. See, concurrency and parallelism in async Rust are different than in non-async Rust. The `Thread` abstraction packages both *concurrency* and *parallelism* into a single abstraction. Whereas in async Rust the two can be decoupled: we can execute any number of futures concurrently, and we don't need to also make us of parallelism for it. Let's walk through some

examples, starting with parallel execution using unstructured thread handles:

```
use std::thread;

let mut handles = vec![];
handles.push(thread::spawn(|| {
    1 // ← the result of some computation
}));
handles.push(thread::spawn(|| {
    2 // ← the result of another computation
}));

let output = handles.into_iter().map(|h| h.join().unwrap())
assert_eq!(output, 3);
```

Rust does not provide us with a way to say that no, we don't actually want to leverage parallelism here - we just want concurrency. That's why `thread::spawn` always takes a `+ Send` bound on the closure. In async Rust however, we can just choose to execute work concurrently via the `Join` family of APIs. Here's an example using `futures-concurrency`:

```
use futures_concurrency::prelude::*;

let mut handles = vec![];
handles.push(async {
    1 // ← the result of some computation
}));
handles.push(async {
    2 // ← the result of another computation
}));

let output = handles.join().await.into_iter().sum();
assert_eq!(output, 3);
```

Structurally this is very similar to the unstructured threads example; however because futures are lazy and automatically propagate cancellation, they can be considered structured [3]. Though typically we'd probably write this example like this instead:

```
use futures_concurrency::prelude::*;
```

```
let a = async { 1 };
let b = async { 2 };

let output = (a, b).join().await.into_iter().sum(); // ← ex
assert_eq!(output, 3);
```

Now what about parallelism? Well, the point I'm trying to make is that we can achieve parallel execution through *composition* rather than defining new APIs. It's common practice today to resort to `task::spawn` APIs for this, mirroring the `thread::spawn` APIs:

```
use async_std::task;

let mut handles = vec![];
handles.push(task::spawn(async {
    1 // ← the result of some computation
}));
handles.push(task::spawn(async {
    2 // ← the result of another computation
}));

let output = handles.into_iter().map(|h| h.await).sum();
assert_eq!(output, 3);
```

There's a pretty noticeable difference between the previous two examples: one family of async APIs for concurrency, and another family of APIs for both concurrency and parallelism. My pitch here is different: I believe the right way to achieve parallel execution is through *composition*. What we need is not another way to schedule async work; what we need is a way to define a *parallelizable future*. And that's something I've prototyped in my `tasky` *crate:

```
use futures_concurrency::prelude::*;
use tasky::prelude::*;

let a = async { 1 }.par(); // ← added `.par` to create a `P
let b = async { 2 }.par(); // ← added `.par` to create a `P

let output = (a, b).join().await.into_iter().sum(); // ← ex
assert_eq!(output, 3);
```

This approach makes it so we have one way of scheduling concurrent execution, and resources themselves are responsible for deciding whether they should be parallelizable or not. Again: **async Rust allows us to decouple parallelism from concurrency in ways not possible in non-async Rust; and so we should design our APIs in ways which leverage that decoupling.**

With `ParallelizableFuture` work doesn't start until it is first `.await` ed. This makes it behave just like any other future. Unlike task handles you can't just fire and forget it; you have to be actively `.await` ing it to make forward progress. That means a value is always returned, and cancellation will always propagate. And once we have async destructors, those should be able to naturally propagate through the `.await` points too. This is an API which should be familiar to use, but hard to misuse. It's setting people up for success when it comes to things like propagating cancellation and learning about async concurrency.

TASKS ARE THE WRONG ABSTRACTION FOR CONCURRENT ASYNC EXECUTION

This point is probably easier to argue than the previous one: using `spawn` APIs for just concurrency without also leveraging parallelism is generally not a great experience. Consider the following example, using `task::spawn_local`:

```
use async_std::task;

let mut handles = vec![];
handles.push(task::spawn_local(async {
    1 // ← the result of some computation
}));
handles.push(task::spawn_local(async {
    2 // ← the result of another computation
}));

let output = handles.into_iter().map(|h| h.await).sum();
assert_eq!(output, 3);
```

This now does the exact same thing as our earlier `Join` example, except it needs to allocate space on the heap to store each

individual future. That's a flat performance tax each task needs to pay; and in this case we've already shown it's avoidable in ever scenario. But that's just performance; there are additional restrictions with regards to ergonomics. The signature of `spawn_local` is as follows:

```
pub fn spawn_local<F, T>(future: F) -> JoinHandle<T>
where
    F: Future<Output = T> + 'static,
    T: 'static,
```

The `'static` lifetime ensures that the future cannot contain any borrows, and resolving it takes a lot of effort because it isn't natural to the language. An example of this is the `moro` crate, which provides an API for "scoped single-threaded tasks" via an `async_scope!` macro. The macro is necessary because the lifetimes required for this currently can't be expressed in the language. Here is an adaptation of the `thread::scope` example converted to use `moro` :

```
let mut container = vec![1, 2, 3];
let mut num = 0;

moro::async_scope!(|s| {
    s.spawn(async {
        println!("hello from the first scoped thread");
        dbg!(&container);
    });
    s.spawn(async {
        println!("hello from the second scoped thread");
        num += container[0] + container[2];
    });
    println!("hello from the main thread");
}).await;

container.push(4);
assert_eq!(num, container.len());
```

Let's rewrite this using the `futures-concurrency` , which doesn't rely on tasks, doesn't enforce `'static` lifetimes, and so in turn can freely express what is being expressed here:

```rust
use futures_concurrency::prelude::*;

let mut container = vec![1, 2, 3];
let mut num = 0;

let a = async {
    println!("hello from the first future");
    dbg!(&container);
};

let b = async {
    println!("hello from the second future");
    num += container[0] + container[2];
};

println!("hello from the main future");
let _ = (a, b).join().await;
container.push(4);
assert_eq!(num, container.len());
```

There are more complex cases possible where we have dynamically updating sets of futures or streams we want to append to, which we want to manage as a group. Getting into that here would mean we'd run long, but for an example of the problem I'd like to point to Niko's mini-redis post, and for an example of how to solve this without tasks or `select!`, see the second example of the `StreamGroup` type.

I realize that by this point we've veered pretty far off the original point of this section. But it's pretty trivial it probably bears repeating: **`Tasks` with their `'static` bounds and performance overhead seem like a poor fit when used solely for concurrency.** And while crates like `moro` can help overcome some of those challenges, they don't do it fully and don't appear to provide additional expressivity.

STEALING AND SENDING

Baked into Rust's async design is the assumption that work-stealing schedulers represent the pinnacle of performance, and therefor it for example makes sense that `Waker` must always be `Send`. Work-stealing allows threads to "steal" work from other

threads when they're idle. In case one thread has a lot of work scheduled, and another thread is free, this is supposed to enable lower latencies and more throughput.

This not without downsides though: in order to facilitate this, it requires that every future contained within a task is Send . **The premise of work-stealing is that the performance gains it provides are more than the performance penalties we incur from requiring all futures are Send .** Because making futures Send not only carries a degree of complexity for the language, it also comes with inherent performance penalties because it requires synchronization. You know how you can't use `Rc` with `async/.await` - that's a direct artifact of work-stealing designs.

The Glommio and Monoio runtimes put this premise into question. Neither of them provide a work-stealing runtime, preferring to use a "thread-per-core" design instead. But by doing this, they do not require to use additional synchronization primitives, and seem to perform better on networked benchmarks than work-stealing runtimes. **Monoio claims 2x the throughput of Tokio with 4 cores, and 3x the throughput of Tokio with 16 cores.** This is possible because of their thread-per-core design, but likely also usage of `io_uring` . I believe we should get updated numbers on this, at least comparing Monoio to the `tokio-uring` project.

SHARDED SEND BOUNDS

Tmandry raised an interesting idea a while back: using `Rc` and other `!Send` types inside of `Send` futures should fine, as long as we can guarantee that all references are moved together as a group. Over in Swift conversations have recently begun about a feature called Region Based Isolation, describing a very similar idea based on the ideas from: "A Flexible Type System for Fearless Concurrency" from PLDI 2022. The Swift SE describes it as follows:

> *Through the usage of isolation regions, the language can prove that transferring a non-Sendable value over an isolation boundary cannot result in races because the value (and any other value that might reference it) is not used in the caller after the point of transfer.*

Translating this to Rust, I believe it would allow us to do the following:

```
let rc = Rc::new(12usize); // ← `!Send` type
task::spawn(async move {   // ← crossing a `Send` boundary
    dbg!(rc);              // ← all references moved: compi
}).await;
```

It makes sense that this is all fine: all references to `Rc` are moved to a new thread, so that's not an issue. But I don't know whether this can hold for all `!Send` types. I don't think it would be, because occasionally threads will be "special" and so moving a `!Send` type to another thread even with all references might end up with trouble. This likely would require an additional modifier to Send , integrated through the libraries and possibly language. It's an interesting idea that needs more research before we can consider it viable, but I wanted to make sure to mention it because it does hold promise.

PARALLEL FUTURES AND SEND BOUNDS

Both Monoio and Glommio provide a local executor as part of their API. Earlier on in this post I've explained why "local executors" are not a great abstraction. That's why for `wasi-async-runtime` , which also features a single-threaded runtime, I've chosen not to provide a `spawn` API at all. Instead people are encouraged to use libraries such as `futures-concurrency` instead.

However, even if work-stealing might not carry its weight in Rust right now, I think we might still be able to provide a `spawn` API. That could potentially even make thread-per-core architectures nicer to express in certain cases where we want an `async fn main` . **The choice of whether we believe work-stealing carries its weight will end up deciding what the right API will be to go with.**

The `task::spawn` API for both Tokio and async-std is quite similar and looks something like this:

```
pub fn spawn<F, T>(future: F) -> JoinHandle<T>
where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
```

It takes a `Future`, and that `Future` must be Send. Easy, right? Now what if we compare that with the signature of `thread::spawn`, which looks like this:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T + Send + 'static,
    T: Send + 'static,
```

Here we don't have a future, but instead we have a closure. The closure itself is Send, and the value `T` is Send. At a glance this might look equivalent to the `task::spawn` APIs, but it's not. This becomes clearer if we encode the original API not as taking `Future`, but instead as taking `IntoFuture`:

```
pub fn spawn<F, T>(into_future: F) -> JoinHandle<T>
where
    F: IntoFuture<Output = T> + Send + 'static,
    <F as IntoFuture>:IntoFuture: Send + 'static,
    T: Send + 'static,
```

Not only is the thing we pass into the other thread Send; because we want to allow the ongoing computation itself to be movable between threads the future itself must also be Send. `thread::spawn` can use `!Send` types freely because once the computation has started, it will not be moved. If we wanted to encode that, we could do that by changing it to the following signature:

```
pub fn spawn<F, T>(into_future: F) -> JoinHandle<T>
where
    F: IntoFuture<Output = T> + Send + 'static,
    T: Send + 'static,
    // `F:IntoFuture` is no longer required to be `Send` he
```

Whether `task::spawn` should take `Send` bounds is a decision we're going to have to make if we want to encode a standardized spawn API in the stdlib. It's not necessarily zero-sum though; we could probably encode both. But so far evidence seems to indicate

that if we want maximum performance thread-per-core is a better approach; while if we want maximum convenience enabling `!Send` types to work inside of parallelizable futures may actually be simpler to use. More data here would certainly be helpful.

A VISION FOR CONCURRENCY AND PARALLELISM IN RUST

So far I've asked a number of questions, and probably observed a little too much. I imagine the average async Rust user will be somewhere between confusion and morbid curiosity of where I'm going with all of this. I want to put a rough sketch forward of where I, Yosh, would like concurrency and parallelism in Rust to eventually get to. I believe we could get pretty far if we made concurrency and parallelism first-class concepts in Rust via two new keywords, which we'll call `par` and `co`.

Take the venerable Rayon `ParallelIterator` trait. It allows us to iterate over items in parallel rather than in sequence. While it works great using combinator APIs; it does not allow us to use `for`-loops the way we'd expect. What if we could do that by introducing `for par..in` loops:

```
fn square(input: impl Iterator<Item = i32>) -> impl Iterato
    gen move {                  // ← current unstable `gen`
        for par num in input { // ← parallel iteration synt
            yield num * num;
        }
    }
}
```

The body of the loop here would be roughly equivalent to Rayon's `ParallelIterator::for_each` - with the exception that it doesn't just loop but returns an iterator. In async Rust we don't yet have async iteration, but we're currently looking at something like this:

```
fn square(input: impl async Iterator<Item = i32>) -> impl a
    async gen move {              // ← current unstable `as
        for await num in input {  // ← sequential async ite
            yield num * num;
```

```
        }
    }
}
```

This is all unstable and unconfirmed, but it seems likely things will end up along these lines. The least certain part is the exact shapes and names of traits, but that's not the important bit here. Now as we've said we can decouple concurrency and parallelism in async Rust. So what would a concurrent version of this loop look like? Well, one of the main benefits of async Rust is that we can run things concurrently - so what if we made that a first-class part of the language? That's what a hypothetical `co` keyword could be for:

```
fn square(input: impl async Iterator<Item = i32>) -> impl a
    async gen move {
        for co await num in input {  // ← concurrent async
            yield num * num;
        }
    }
}
```

Whether we'd spell this `co.await`, `co_await` or `co await` doesn't particularly matter. Making concurrency easier to leverage seems like a nice thing. In terms of implementation we could leverage my recent work on `ConcurrentStream` for this. If we then wanted to extend this to parallelism too, we could instead use `par await`:

```
fn square(input: impl async Iterator<Item = i32>) -> impl a
    async gen move {
        for par await num in input {  // ← parallel async i
            yield num * num;
        }
    }
}
```

This doesn't just need to stop at iterators either; we could integrate this into futures and `async/.await` too. Not too different from how Swift's `async let` notation works. Remember our earlier example using `futures-concurrency` to evaluate futures concurrently?

```
use futures_concurrency::prelude::*;

let a = async { 1 };
let b = async { 2 };

let output = (a, b).join().await.into_iter().sum(); // ← ex
assert_eq!(output, 3);
```

It would be nice if the compiler could perform control-flow analysis directly, and for automatically schedule concurrent execution of futures where permitted, as long as they were called with `.co.await`:

```
let a = async { 1 }.co.await; // ← concurrent await syntax
let b = async { 2 }.co.await; // ← concurrent await syntax
assert_eq!(a + b, 3);
```

And what about parallelism? Well, we should also be able to compose `.par` with `.await` to achieve parallel async execution:

```
let a = async { 1 }.par.await; // ← parallel await syntax (
let b = async { 2 }.par.await; // ← parallel await syntax (
assert_eq!(a + b, 3);
```

One of the main appeals of representing async operations as types is that we can then arbitrarily combine them with other futures to achieve concurrent execution. Neither future here needs to be `'static` to work with `par`, and borrows should just work as expected. **If we are able to bake concurrent and parallel execution directly into the language, we no longer have to represent the computation as a type. By making `.par` a modifier to `.await`, parallel futures would not be represented in the type system and we would be able to solve the scoped parallel async execution problem directly from the language.**

Oh and the other nice bonus of this: it would work perfectly with `#[maybe(async)]` function bodies, as we can always fall back from concurrent semantics in async contexts to sequential semantics in non-async contexts. There is probably also something

interesting to be said about bare `par {}` blocks and scoped threads, but that's out of scope (ง ˙ω˙)ง for now.

CONCLUSION

In this post we've jumped around a fair bit. You would be forgiven for having some trouble following all threads and ideas. But let me try and summarize the arguments I've attempted to make:

1. **Tasks are a poor fit for non-parallel concurrent execution.** They come with additional performance overhead and impose `'static` lifetime restrictions, creating knock-on problems.
2. **Tasks are a poor fit for parallel concurrent execution.** They are presently designed to function as "async/.await" versions of threads. And as a result combine both concurrency and parallelism into a single API. Instead we would do better to provide a `ParallelFuture` type which provides parallel execution through *composition* with concurrency primitives.
3. **The success of the Monoio and Glommio runtimes are putting into question whether work-stealing executors are the right fit.** They show significant performance improvements over work-stealing by leveraging a thread-per-core architecture. We need more data to understand the differences before we can make decisions.
4. **Deciding between thread-per-core and work-stealing executors has ramifications for what `spawn` APIs should look like.** Work-stealing executors can't capture `!Send` types, while non-work-stealing executors can operate on `!Send` types.
5. **There might be benefits to elevating concurrency and parallelism primitives from the libraries into the language.** Fast, safe, concurrent execution is Rust's flagship feature, and making that more accessible would likely pay dividends. This would also provide an alternative way of expressing "scoped tasks".

I hope that if you take anything away from this post it's some of the ground truths of async Rust may be less set in stone than you expected them to be. There definitely appear to be tradeoffs between the various task scheduling approaches and designs, and I don't think we should just assume that work-stealing is the better approach. What we really need is a better understanding of the

tradeoffs involved, and for that we're going to need data. Deciding without measuring is not going to work.

Oh also: **we cannot provide standardized async APIs for parallelism without first stabilizing async destructors.** I find myself repeating this so often I feel like a walking meme at this point. Without async destructors so much of async Rust is broken, and we cannot stabilize interfaces before we know how async destructors will interact with them. Async parallelism specifically is also extra-broken. Async Rust is the flagship feature of priority for Rust as a language, and async closures + async destructors is where we should be spending our time as they're fundamental building blocks.

And on a closing note: I just want to put out there that we should dare to dream beyond the mere ossification of the status quo. Better things are possible, as long as we take care of each other and are willing to put in the work. I regularly tell myself this; and now I'm it to you too.

NOTES

1. Note that the third example here says: "structured". What I actually mean is: as structured as is possible when working within the current limitations of the language. But this API would be entirely structured if we didn't have those limitations. ←
2. Romio and Juliex were the reactor and executor Boats and I worked on between 2018-2019. It's a pretty good snapshot of what the state of the art of APIs looked like back then. ←
3. I've gone into details about the limitations of structured concurrency in Rust today in other posts. TLDR: we need an async version of `Drop` to actually solve all cases. ←

REFERENCES

▸ View all references