Practical `async` Rust over lunch

cpg

November 2023

Goal

Run tasks in parallel so that they complete faster.

Two categories of tasks

- Compute-bound (CPU, memory)
  $\rightarrow$ Fully utilize the hardware.
- I/O-bound: reading/writing data, locally or over network.
  - Read data from disk (network or local).
  - Make a web request and retrieve the response.

  $\rightarrow$ Do things while waiting.

Tasks can also combine the two over their lifetime, e.g. retrieve data, then perform a computation, then write results.

Send each task to a thread?

In `std::sync`:

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

So we can do:

```
let task1 = std::thread::spawn(|| { ... });
let task2 = std::thread::spawn(|| { ... });
// Wait until the tasks have completed.
// Returns an error if the thread panicked.
task1.join()?;
task2.join()?;
```

Issues

- Thread overhead (e.g. context switches), in particular if there are many tasks.
- Ergonomics (e.g. synchronization between threads).

Use a thread pool to limit the overhead.

rayon crate

One thread per CPU available (default), each with a work queue.

```
use rayon::prelude::*;
// Each task needs to be Send.
tasks.into_par_iter().for_each(|task| ...);
```

With work-stealing, rayon can efficiently handle tasks spawning other tasks (e.g. `flatten`).

Issues

Still not adapted for I/O-intensive tasks, e.g. sending and waiting on 100 HTTP queries.

Driving principles

- Large number of operations (*futures*), often cheap and I/O-bound.
- Runtime (single- or multi-threaded) cheaply switches between tasks as they are able to make progress, until they complete.
- Similar ergonomics to sequential code:

```
async fn f(x: u64) { ... }

// Run sequentially
f(42).await?;
f(41).await?;

// Run concurrently, by creating a new future that will
// start both calls and wait on them.
join!(f(42), f(41));
```

Async Rust basics

Async Rust basics

Native support

- `async fn`, `await`
- `Future` trait in the standard library.
- Async traits: should be stabilized for Rust 1.75. In the meantime, use the `async_trait` crate.

External resources

- Some utilities in the futures crate: joining, selecting, streams. . .
- Bring your own runtime; most widely used is Tokio.
- Tokio brings asynchronous I/O APIs for network, filesystem, signals, processes.

The `Future` trait for asynchronous computations

```
pub trait Future {
  type Output;
  // Ready? If not, make progress, without blocking.
  // `Context` provides a callback for the runtime to call
  // poll again when the future is ready to make more
  // progress.
  fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
      -> Poll<Self::Output>;
}
pub enum Poll<T> {
  Ready(T),
  Pending,
}
```

Role of the runtime

Polling futures until they complete, using one or more threads.

Async Rust basics

Async functions and tokio entrypoint

`async fn` and `async` block

```
async fn f(...) -> T { ... }
// is syntactic sugar for
fn f(...) -> impl Future<Output=T> {
  // State machine generated with all the futures
  // awaited in `f`.
  ...
}

async { ... } // async block
```

Tokio entrypoint

```
// This simply runs the future on the main thread
// until completion.
#[tokio::main]
async fn main() { ... }
```

Async Rust basics

Examples of futures

- `tokio::time::sleep`: deeply integrated into the runtime (see (this post)).
- Network I/O: the `poll` function can use the `epoll` notification mechanism to notify the waker when more data is available, so that the runtime can poll the future again. See this page.
- `tokio::task::spawn_blocking`: runs a blocking (non-async) function on a separate threadpool (default size 512).

```
pub fn spawn_blocking<F, R>(f: F) -> JoinHandle<R>
where
  F: FnOnce() -> R + Send + 'static,
  R: Send + 'static;

spawn_blocking(|| { ...  }).await?;
```

- File I/O: Uses `spawn_blocking`, as `epoll` is not available. (`io_uring` is, but support is still experimental).

Synchronization primitives

Running multiple futures at the same time

Sharing data

Limiting concurrency

Channels

Synchronization primitives

Running multiple futures at the same time

```
// Runs sequentially
f1(...).await;
f2(...).await;

// That too (unlike javascript)
let fut1 = f1(...);
let fut2 = f2(...);
fut1.await;
fut2.await;
```

How do we actually run futures concurrently?

For example, process 100 HTTP queries, making progress on some while others are waiting on I/O.

Synchronization primitives

Running multiple futures at the same time

Joining futures

```
// Runs in parallel, but in the same task (=> thread).
use futures::{future::join_all, join};
join!(f1(...), f2(...));
join_all(vec![f1(...), f2(...)]).await;

// If the order does not matter:
use futures::stream::{StreamExt, FuturesUnordered};
let fut: FuturesUnordered = it.collect();
fut.collect().await?;
```

The `join` methods create a new future that polls all the futures to completion.

Technical anecdote: an earlier version of the `join_all` method had quadratic complexity because every poll would poll all the futures.

Since then, the task uses a more clever (but slightly more expensive) implementation when there are more than 30 futures.

Warning

Do not try to join an ungodly amount of futures without limiting concurrency.

Synchronization primitives

Running multiple futures at the same time

Tasks are the scheduling units in tokio.

- Cheap alternative to OS threads.
- Calling `await` in a task *yields* to other tasks.
- Tasks can move between threads (work stealing), not futures.

Spawning tasks

```
pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
where
  F: Future + Send + 'static, F::Output: Send + 'static;
```

Example:

```
// Start running both tasks.
// Each task will be scheduled on a member of the thread pool.
let task1 = tokio::task::spawn(async { f1(...).await });
let task2 = tokio::task::spawn(async { f2(...).await });
// Error if the tasks panicked or have been cancelled.
task1.await?;
task2.await?;
```

Synchronization primitives

Running multiple futures at the same time

Spawning vs joining

- Drawback: Unlike `join`, `spawn` requires `Send + 'static` (naturally).
- Advantage: Allows parallelism of compute-bound segments (when using a multithreaded runtime).

Warning

This compiles, but likely does not do what you want:

```
async fn f() { ... }
tokio::task::spawn(async { f() }).await?.await;
```

Synchronization primitives

Running multiple futures at the same time

Comparison

```
// Sequential
f(0).await;
f(1).await;
// Concurrent, same thread (same task)
join_all([f(0), f(1)]).await;
// Concurrent, possibly multi-threaded (different tasks)
let task1 = tokio::task::spawn(f(0));
let task2 = tokio::task::spawn(f(1));
task1.await?;
task2.await?;
```

rayon	threadpool for compute-bound sync tasks.
`futures::join`	await multiple futures.
`task::spawn`	run a future as a separate task.
`task::spawn_blocking`	run a sync/blocking function on a large threadpool.

Bounds: `spawn` and `spawn_blocking` require `Send + 'static`.

Synchronization primitives

Running multiple futures at the same time

Sharing data

Limiting concurrency

Channels

How to make data readeable/writeable from different futures/tasks (possibly on different threads)?

The borrow checker forces us to think about

- Lifetimes
- Mutability

but therefore enables fearless concurrency
(although this does not cover deadlocks!).

Non-mutable references without spawning

No special attention required:

```
async fn f(x: &T) { ... }
futures::join_all([f(&x), f(&x)]).await;
```

Synchronization primitives

Sharing data

Meeting a 'static lifetime when spawning

```
// 'static not met, will not compile:
tokio::spawn( async move {f(&s)}.await );

// Shared ownership of x with Arc (atomic reference count)
let x: Arc<T> = std::sync::Arc::new(x);
tokio::spawn({
    let s = s.clone(); // cheap
    async move { f(s.clone()).await }
}).await;
```

Notes:

- Network clients (`reqwest`, `tonic`...) are usually hiding an `Arc` and are therefore already cheaply `Clone`-able.
- An `Arc`'ed variable cannot be mutated unless interior mutability is used (e.g. a `Mutex`).

Mutating data: (async) Mutexes

```rust
use tokio::sync::Mutex;

let x = Mutex::new(x);

async fn f(x: &Mutex<T>) {
    // This will block any other call from locking
    // until the guard is dropped.
    let guard = x.lock().await;
    // The guard can be used transparently
    // as a &T of a &mut T.
}
```

Can be combined with `Arc` when spawning: `Arc<Mutex<T>>`.

Warning

Do not use `std::sync::Mutex` unless you are sure of what you are doing (risk of deadlocks). See the documentation.

Mutating data: Read-write locks

Allow an arbitrary number of readers OR a single writer.

```
use tokio::sync::RwLock;
let x = RwLock::new(x);
// Read with multiple readers
let f = || async { let x = x.read().await; ... };
futures::future::join_all([f(), f()]).await;
// Write. Blocks any call to .read()
let w = x.write().await;
*w = Default::default();
```

Can be combined with `Arc` when spawning: `Arc<RwLock<T>>`.

Deadlock warning

```
let r = x.read().await;
let x = x.write().await;
```

Same with "write then read". Call `drop` or `downgrade`.

Async allows us to create a very large amount of tasks, but it is still often desirable to put limit on the concurrency:

- Limits of the network resources we are accessing (e.g. APIs).
- I/O limits.
- CPU-bound tasks, whether they execute in the runtime thread or on the large blocking threadpool.

Semaphores

```
use tokio::sync::Semaphore;

let sem = Semaphore::new(10);

// Blocks until a permit is available.
let permit = sem.acquire().await?;
```

Semaphores and spawning

```
let sem = std::sync::Arc::new(tokio::sync::Semaphore::new(2));

let mut tasks = vec![];
for item in items {
    let permit = sem.clone().acquire_owned().await;
    // Permit is moved to the task.
    tasks.push(tokio::task::spawn(async move {
        // do things, permit gets dropped at the end
    }));
}
for task in tasks {
    task.await?;
}
```

Alternative: move a clone of the `Arc`'ed semaphore into the task and acquire a permit inside it. The difference is that we will not block during the `for` loop as permits are released.

Channels in tokio

In `tokio::sync`:

	Producers	Consumers	Remarks
`oneshot`	1	1	Single value
`mpsc`	∞	1	Send work or receive results.
`broadcast`	∞	∞	Each consumer receives each value.
`watch`	1	∞	

`mpsc` comes as bounded or unbounded, `broadcast` is always bounded.

Usage generically looks like:

```
// Depending on the channel rx and/or tx can be Clone'd.
let (rx, tx) = channel::new();

rx.send(value).await;
let value = tx.recv().await;
```

Streams

Definition

Streams are essentially "async iterators".

The `Stream` trait

```
pub trait Iterator {
  type Item;
  fn next(&mut self) -> Option<Self::Item>;
}

pub trait Stream {
  type Item;
  async fn poll_next(&mut self) -> Option<Self::Item>;
}
```

They provide for example more flexible ways of processing a list of futures than `join`:

filtering, mapping, flattening, controlling concurrency. . .

The `StreamExt` and `TryStreamExt` traits provide useful methods to work on streams (resp. of streams of results).

```
let results: Vec<_> = stream::iter(tasks)
        .enumerate()
        .map(|(i, task)| {
            let client = client.clone();
            async move {
                ...
            }
        })
        .buffer_unordered(8)
        .collect()
        .await;
```

When using these, make sure to understand the `Item` types of your streams before and after functions are applied, by reading the *Trait implementation* documentation section on the return type of the combinator.

Mixing compute-bound code

Blocking the runtime

A compute-bound blocking call in an async function will prevent the corresponding runtime thread from polling its futures.

```
async fn f(x: T) {
  let y = g(x).await;
  cpu_heavy(y); // blocks the runtime thread
}
```

This could for example prevent a server from serving requests, or result in a deadlock.

Rule of thumb

Do not spend a long time without `await`'ing.

Seen so far:

rayon	small threadpool for compute-bound tasks.
`task::spawn`	run a future on a separate task.
`task::spawn_blocking`	run a sync/blocking function on a large threadpool (512 threads).

Mixing compute-bound code

Options

- Use `tokio::task::spawn`. Even on a multi-threaded runtime, this does <u>not</u> guarantee that the task will run on a separate worker thread!
  (task spawned on the worker's queue + infrequent work stealing.)
- Use the `spawn_blocking` threadpool, making sure to limit concurrency (e.g. with a `Semaphore`).
- Use a separate threadpool, e.g. `rayon`: `rayon::spawn` and use a `tokio::sync::oneshot` to `await` the result from tokio.
- Use a separate tokio executor (see this post).

Even when using a small threadpool for compute-bound segments, make sure to control concurrency, for example to avoid queued tasks to consume all memory.

Interesting reads:

- https://ryhl.io/blog/async-what-is-blocking/
- https://github.com/tokio-rs/doc-push/issues/77
- https://github.com/tokio-rs/tokio/pull/4105

Async I/O

Async IO traits

Filesystem

Tokio async IO traits

Non-blocking/async analogues of the standard library traits.

std::io::	tokio::io::
Read	AsyncRead
BufRead	AsyncBufRead
Seek	AsyncSeek
Write	AsyncWrite

- Most high-level functions (e.g. read/write buffers) are available in the `[Trait]Ext` extension traits.
- The above traits are implemented on tokio analogues of `std` structs:

std::fs	tokio::fs
std::net	tokio::net
std::io	tokio::io
std::process	tokio::process

Common operations:

- Using higher-level libraries (e.g. `reqwest`, `tonic`) and reading/writing buffers at once.
- Reading or writing buffers from/to implementors of async read/write traits.
- Copying data between an `AsyncRead` and an `AsyncWrite`, using `tokio::io::copy`.

Caveat

The `tokio::fs` operations can be significantly slower than the sync (`std::fs`) ones.

We trade-off performance for non-blocking operations.

The `tokio::fs` operations rely on `spawn_blocking` (in absence of a useful `epoll`). A large part of the overhead then comes from moving data across threads. But also polling, etc. See this issue.

In some cases (very low latency filesystem I/O), it might make sense to directly use blocking calls. See for example this post.

When things go wrong

General issues

Deadlocks

```
error: future cannot be sent between threads safely
  --> src/main.rs:18:5
   |
18 |     require_send(send_fut);
   |     ^^^^^^^^^^^^ future created by async block is not `Send`
   |
   = help: the trait `Sync` is not implemented for `RefCell<i32>`
   = note: if you want to do aliasing and mutation between multiple
           threads, use `std::sync::RwLock` instead
note: future is not `Send` as it awaits another future which
      is not `Send`
```

When things go wrong

General issues

- Lifetime issues:
  - `higher-ranked lifetime error` is very common with streams. A solution is usually either to ensure that `stream::iter` is passed a `'static` object, or to call `boxed()` on your stream. See this.
  - Annotate lifetimes on functions that take multiple references on arguments/outputs.
- `Cannot infer type` in async blocks:

```
async {
  ..
  Ok::<T,E> // Annotate the type, e.g. anyhow::Ok(x)
}
```

- Object needs to be `'static`: Wrap into an `Arc`.
- Object needs to be `Send` (e.g. RNG): Put the non-`Send` code into a scope (see this page).
- Recursion: Use the `async_recursion` crate.
- Traits: Use the `async_trait` crate (until Rust 1.75).

Deadlock

- Task 1 waits on task 2.
- Task 2 can progress only when task 1 does.

```
// Say `fut` can only complete when cleanup is called
fut.await;
cleanup.await; // this is never reached
```

Unfortunately, Rust's memory safety features do not help with deadlocks.

Most common deadlock reasons

- Sync `Mutex` used in `async` context.
- Attempting to acquire a lock twice in the same task.
  - Call `lock` twice on `Mutex`. Use `drop`.
  - Call `read` then `write` or vice-versa on a `RwLock`. Use `drop` or `downgrade`.
- Using multiple locks.
- Use a bounded queue without reading the results.
- More complex circularities.

Debugging tools

- Careful documentation of the locking paths.
- tokio console
- `timed_lock` crate

References

- Official async Rust book
- Tokio tutorial
- Tokio documentation