Outline and explanation of our TCP server + non-trivial implementation details

We used tokio tasks in all parts of our TCP server. We have:
1 tokio task per connection which further spawns
   a. 1 tokio task that reads results from a channel and sends it back to the client
   b. 1 tokio task per client task that executes said task and writes the result to the channel

We heavily rely on the tokio scheduler to enable concurrency. Using tokio makes our server extremely flexible, as the scheduler can arbitrarily schedule any tasks such as {read from client, execute IO task, execute CPU task, write to client}. One of the biggest gains this enables is one thread being able to execute n IO tasks concurrently.

A key implementation detail is that we place IO tasks in core threads (through tokio::spawn) and CPU-intensive tasks in blocking threads (through tokio::task::spawn_blocking). Tokio handles concurrently by swapping at .await points, so code that spends a long time without reaching an .await will prevent other tasks from running. In the execute_async() for the CPU-intensive task, there are no .await points, as compared to the IO task which calls .await() immediately. As such, we placed CPU-intensive tasks in a dedicated thread pool by using tokio::task::spawn_blocking. These blocking threads are used to run blocking code that would otherwise block other tasks from running, if placed in the same core thread. Through the OS scheduler, execution of core and blocking threads will be preemptively scheduled and switched on the same CPU.

Level of concurrency + explanation of design

Our implementation fulfills a task-level concurrency:

Multiple clients can run concurrently: We spawn a new independent task to handle each client connection, meaning many clients can be handled concurrently.

I/O and CPU tasks in the same CPU are executed concurrently:

1. We spawn separate tokio tasks for each message
2. These tasks are then scheduled by the Tokio runtime onto available worker threads
   a. IO tasks are placed in core threads (tokio::spawn)
   b. CPU tasks are placed in blocking threads (tokio::task::spawn_blocking)
3. Core and blocking threads can run in the same CPU.
   a. The OS scheduler can move CPU execution from one thread to another, running CPU-intensive tasks in blocking threads and IO tasks in core threads
4. Tokio can also place multiple tasks like handling a connection, reading the client request, doing the IO task, writing back to the client on the same core thread
   a. These tasks will be cooperatively multitasked via .await points (Tokio handles concurrently running many tasks on a few threads by swapping on .await points)

Brief explanation of cases when the concurrency level decreases

The concurrency level would decrease to a client-level concurrency if async tasks placed in a core thread (through tokio::spawn) run too much blocking code AND/OR do not yield often (through .await), which will hog the thread and reduce the number of tasks that can interleave. This might happen if MysteryTask, which will be placed in core threads in our code, are CPU bound tasks.

Concurrency paradigm

Thread-based: We rely on OS threads and the OS scheduler: each task runs in its own thread, and the OS scheduler does preemptive scheduling to run different threads of different tasks

Event-driven / async: Tasks yield control at .await points; a runtime (like Tokio) schedules them efficiently.

Shared-memory parallelism: Tasks synchronize access via semaphores

Will your server run tasks in parallel? Briefly explain.

Our server will run tasks in parallel. For example, Multiple threads can run CPU tasks while another thread can read/write from a TCP stream. This is because tokio can schedule these tasks to run on different hardware threads. Some threads (such as those reading/writing client data) can't truly parallelize because they rely on the same underlying TCP stream (at least in some implementations), but when that is not the case, our program enables true parallelism.

Other Implementations

The first implementation that we tried was to have one kernel thread for each client and a thread pool which is in charge of executing the tasks and sending the results back to the clients. We realised that this would be rather complex to optimise, since we weren't sure which ordering of tasks would produce efficient runtimes. This made us realise that we were trying to reinvent Tokio's task scheduler, so we proceeded to just lean into Tokio heavily.

Secondly, we ran CPU-intensive and IO tasks asynchronously with Task::execute_async() at first. Before this optimization, it was possible that all of our core threads were busy with CPU intensive tasks while IO tasks waited in queue. Now with this change, blocking is expected and contained within blocking threads and the OS uses preemptive scheduling to switch thread execution between blocking and core threads.