Dominic (A0238913Y) and Wen Jun (A0239038B)

0.1 Outline of TCP server

Our TCP server is constructed atop the Tokio runtime and is designed to ensure the reliable, ordered, and fault-tolerant delivery of data bytes. During the server's construction, we assumed that we would be able to effectively differentiate between task types (CPU-bound or IO-bound), as each requires a distinct method of handling to fully leverage the concurrency level. Some non-trivial implementation methods of how we process request multiple clients are listed below.

0.1.1 Spawning of Tokio tasks

Upon accepting a new connection from the user, we will spawn a new tokio task to handle the connection with that single client. This allows our server to handle multiple clients concurrently as long as they are connected to the server.

```
loop {
    let stream = listener.accept().await.unwrap().0;
    tokio::spawn(async move {
        Self::handle_connection(stream).await;
    });
}
```

0.1.2 Output and Input handling

For each connection, we also split the TcpStream into reading and writing ends. We also create an mpsc channel, and spawn a new tokio task to handle the output of results into the TcpStream. This allows us to handle multiple client requests in parallel as the task is not blocked on waiting for the result to be computed, and instead a task is spawned for each new computation, and the results are sent through the channel by these new tasks.

```
let (mut read, mut write) = tokio::io::split(stream);
let (tx, mut rx) = tokio::sync::mpsc::channel(50);
// Spawn a task to write responses to the client
tokio::spawn(async move {
    while let Some(message) = rx.recv().await {
        if write.write_all(&[message]).await.is_err() {
            eprintln!("Failed to write response");
            return;
        }
    }
});
```

0.2 Concurrency Paradigm

We used the **asynchronous programming**, **thread pool** and **channels** to communicate between threads in our implementation. On the client level, a new tokio task has been spawned to handle each incoming connection. Additionally, upon receiving input, a new tokio task is spawned to handle this input, allowing the handler to continue reading incoming input after spawning this task.

For operations that are CPU-bound and inherently more resource-intensive, the server utilizes Tokio's 'spawn_blocking' function. This function offloads the CPU-intensive tasks to a dedicated thread pool that is optimized for blocking operations. The separation of CPU-bound tasks into a distinct thread pool ensures that these operations can proceed in parallel with other asynchronous tasks.

In order to keep to the constraint of 40 concurrent CPU tasks, a counting semaphore of size 40 is used, causing the task to block if there are currently more than 40 CPU-intensive tasks executing.

```
if task_type == TaskType::CpuIntensiveTask as u8 {
    let permit = TASK_LIMIT.acquire().await.expect("Failed to acquire permit");
    let res = task::spawn_blocking(move || Task::execute(task_type, seed))
        .await.unwrap();
    drop(permit);
    let _ = tx.send(res).await;
} else {
    let v = Task::execute_async(task_type, seed).await;
    let _ = tx.send(v).await;
}
```

0.3 Concurrency Level

Our server seeks to achieve **task level concurrency**.

0.3.1 Processing Requests from multiple client

The server can handle multiple clients concurrently because of the tokio::spawn function, which is used to create a new asynchronous task for each incoming connection. This design leverages Tokio's multi-threaded executor to run multiple tasks across available CPU cores. Since each connection is handled in its own task, the server efficiently manages multiple connections without one blocking another, enabling high concurrency and scalability.

0.3.2 Task parallelism

The server is able to run tasks in parallel, inherently due to the its design and the use of the 'Tokio' runtime. Since each connection has its own dedicated thread, the server achieves concurrency by allowing multiple connections to be handled at once.

Additionally, with our use of channels and a separate task for outputting results to the writing end of the TcpStream, multiple tasks (IO or CPU) from the same connection can be handled concurrently.

Worst Case Concurrency

In the case when the 40 CPU-task limit is reached and the task is blocked on waiting for the semaphore, IO tasks can still run concurrently. In all other cases, IO and CPU tasks can run concurrently.