Trabajo Práctico 7: Comparación de Threads y Async Programming

Programación Concurrente - Primer cuatrimestre 2025

Profesores: Emilio Lopez Gabeiras y Rodrigo Pazos

10 de junio de 2025

1. Introducción

Este trabajo práctico busca comparar dos enfoques de programación concurrente: el modelo tradicional basado en *threads* y el modelo basado en *async programming*. Se implementarán tareas tanto de I/O como de cómputo intensivo, utilizando ambos enfoques. El objetivo es observar y analizar diferencias. **Es necesario utlizar la librería Tokio de Rust.**

2. Objetivos

1. Implementar un sistema concurrente para simular tareas de I/O y cálculo intensivo, usando tanto threads como async (task::spawn en tokio).
2. Medir y comparar tiempos de ejecución y uso de CPU en ambos modelos.
3. Analizar diferencias de performance entre el enfoque orientado a I/O y al cálculo numérico.

3. Implementación

El código debe incluir las siguientes funcionalidades clave, implementadas utilizando tanto el enfoque de `threads` como el de `async/await`:

- **Simulación de tareas I/O:** Cada tarea debe simular una operación de I/O con una espera artificial. En la versión con `threads`, esto se puede lograr usando `std::thread::sleep`. En la versión `async`, debe utilizarse `tokio::time::sleep` para evitar bloquear el thread principal.
- **Cálculo concurrente de Pi con la serie de Leibniz:** Se debe dividir el trabajo del cálculo en subtareas que se ejecuten en paralelo o de forma asincrónica. Cada subtarea debe calcular una parte de la serie (ver función `leibniz_pi_partial` en la sección anterior), y al finalizar se deben combinar los resultados para obtener la aproximación final del valor de π.
    - En la versión con `threads`, se pueden lanzar varios threads y usar `join` para recolectar los resultados.
    - En la versión `async`, se pueden usar `tokio::spawn` o `futures::join` para ejecutar múltiples tareas concurrentes.
    - El número de tareas y la cantidad de términos a calcular deben ser configurables mediante argumentos o constantes.

4. Instrucciones para la experimentación

Ejecutar el programa probando distintas combinaciones de parámetros para observar cómo varían los tiempos y el comportamiento de CPU en función del número de tareas y la cantidad de términos para el cálculo de Pi.

- `tasks`: número de tareas concurrentes que se lanzan (por ejemplo, 10, 100, 1000, 1 000 000).
- `terms`: cantidad de términos usados para la serie de Leibniz en el cálculo de Pi (por ejemplo, 10 000, 1 000 000, 10 000 000).

Para el cálculo de Pi utilizando la serie de Leibniz, se puede usar la siguiente función auxiliar:

```
fn leibniz_pi_partial(start: usize, count: usize) -> f64 {
    (start..start + count)
        .map(|k| {
            let k = k as f64;
            (-1.0f64).powf(k) / (2.0 * k + 1.0)
        })
        .sum::<f64>()
        * 4.0
}
```

Listing 1: Función para cálculo parcial de la serie de Leibniz

Esta función calcula una porción de la serie de Leibniz, útil para distribuir el trabajo entre varias tareas o threads.

5. Análisis sugerido

- Compare los tiempos de ejecución para I/O simulado entre threads y async con diferentes valores de tasks. ¿Cuál es más eficiente en manejar muchas tareas con esperas? Tip: probar con valores altos y explicar por que threads falla y async no.
- Compare los tiempos de ejecución para cálculo de Pi intensivo en CPU entre threads y async. ¿Qué modelo se desempeña mejor?
- Experimente con diferentes cantidades de tareas y términos para Pi para evaluar cómo escala cada enfoque.