INFORME DE LABORATORIO

INFORMACIÓN BÁSICA					
ASIGNATURA:	**Tecnología de Objetos**				
TÍTULO DE LA PRÁCTICA:	*- Hilos en C++* *- Hilos en Java y en GO*				
NÚMERO DE PRÁCTICA:	*04*	**AÑO LECTIVO:**	*2025-B*	**NRO. SEMESTRE:**	*VI*
FECHA DE PRESENTACIÓN	*04/10/2025*	**HORA DE PRESENTACIÓN**	*11:59*		
INTEGRANTE (s): Payehuanca Riquelme Jhastyn Jefferson MICHAEL BENJAMIN SUCLLE SUCA				**NOTA:**	
DOCENTE(s): *CARLO JOSE LUIS CORRALES DELGADO*					

SOLUCIÓN Y RESULTADOS

Código Anexado:

https://github.com/MichaelSucSuc/TO.git

Descripción General

Este proyecto implementa el método del trapecio para calcular numéricamente el área bajo una función continua en un intervalo [a,b][a, b][a,b].
El algoritmo se ejecuta en paralelo, distribuyendo los cálculos parciales entre varios hilos o procesos concurrentes, según el lenguaje:

- Java: Threads
- C++: std::thread
- Go: Goroutines y WaitGroup

La función de prueba utilizada es:

f(x) = 2x^2 + 3x + 0.5

con los límites a = 2 y b = 20

El programa incrementa progresivamente el número de trapecios n hasta que el valor aproximado de la integral se estabiliza, es decir, la diferencia entre dos aproximaciones consecutivas es menor que una tolerancia establecida (1e-9)

Trapecio.java

Características:

- Usa interfaces funcionales (@FunctionalInterface) para representar f(x)f(x)f(x)
- Cada hilo (TrabajadorTrapecio) calcula una parte de la suma total
- El método integrar() combina los resultados de todos los hilos
- El programa aumenta n hasta que el resultado se estabiliza

Tecnologías usadas:

- Java 8+
- Programación Orientada a Objetos
- Clases e hilos (Thread)

`trapecio.cpp`

Características:

- Uso de clases y herencia para simular el comportamiento POO de Java.
- Implementación paralela con `std::thread`.
- Manejo explícito de memoria con `new` y `delete`.
- Estructura modular (clases `FuncionUnivariable`, `TrabajadorTrapecio`, `Trapecio`).

Aspectos técnicos:

- Sincronización por `join()`.
- Dividir las iteraciones entre hilos de forma uniforme.
- Cálculo incremental hasta estabilización.

Archivo: `trapecio.go`

Características:

- Implementación funcionalmente equivalente a las anteriores.
- Utiliza goroutines (ligeras y no bloqueantes) y `sync.WaitGroup` para sincronización.
- Mantiene la misma estructura conceptual que las versiones Java/C++.
- Detección automática de estabilización del resultado.

Ventajas del enfoque Go:

- Mayor eficiencia y simplicidad en la concurrencia.
- Sin necesidad de gestionar memoria manualmente.
- Ejecución rápida gracias al *scheduler* interno de goroutines.

Los tres comparten la misma salida:

```
n = 57751   Área aproximada = 5931.000000582900

El valor de la integral se ha estabilizado.
Valor final aproximado: 5931.000000582900

Valor exacto (analítico): 5931.000000000000
Error absoluto: 0.000000582900
```

Resumen y Análisis de las Implementaciones de Trapecio y TrapecioPool

Los tres lenguajes implementan el método del trapecio para calcular integrales definidas, pero con enfoques notablemente diferentes en cuanto a gestión de concurrencia y modelado de threads. Java utiliza el ExecutorService framework que proporciona una abstracción robusta y bien integrada dentro del ecosistema Java, ofreciendo pools de threads preconfigurados con gestión automática del ciclo de vida. Esta implementación es verbosa pero segura, aprovechando años de evolución en concurrencia dentro de la JVM. El enfoque de Java se centra en la estabilidad y el control granular a través de Futures y Executors, aunque con un overhead de memoria significativo debido a la naturaleza pesada de los threads tradicionales.

C++ presenta una implementación más cercana al metal, donde el desarrollador tiene control total sobre la gestión de memoria y sincronización. La implementación del thread pool en C++ requiere código manual sustancial usando std::thread, std::mutex y std::condition_variable, lo que ofrece máximo rendimiento potencial pero con alto riesgo de errores de concurrencia. La versión con pool en C++ demuestra la complejidad inherente de gestionar recursos manualmente, aunque proporciona flexibilidad sin igual para optimizaciones específicas. Esta aproximación es ideal para aplicaciones donde el rendimiento crudo es crítico, pero requiere expertise significativo en programación concurrente.

Go revoluciona el enfoque con goroutines y channels, donde el concepto de "pool" es más un patrón de diseño que una estructura pesada. Las goroutines, siendo extremadamente ligeras, permiten una concurrencia masiva con overhead mínimo. La implementación en Go destaca por su simplicidad y elegancia, usando channels para comunicación segura entre goroutines y waitgroups para sincronización. El runtime de Go maneja inteligentemente el scheduling de millones de goroutines sobre un número menor de threads del SO, haciendo que la concurrencia sea accesible y eficiente. Esta aproximación es particularmente efectiva para operaciones I/O bound y servicios concurrentes.

En términos de rendimiento práctico, Java ofrece un balance entre rendimiento y facilidad de desarrollo, C++ proporciona el máximo rendimiento potencial a costa de complejidad, y Go excelente eficiencia en escenarios de alta concurrencia. La elección entre estas implementaciones

depende críticamente del contexto de aplicación: sistemas empresariales maduros se benefician de la robustez de Java, aplicaciones de alto rendimiento justifican la complejidad de C++, mientras que servicios cloud y microservicios se inclinan hacia la eficiencia de Go. Cada lenguaje refleja su filosofía de diseño subyacente en cómo aborda el desafío de la programación paralela, demostrando que no existe una solución universal sino diferentes herramientas optimizadas para distintos escenarios de uso.

Preguntas:

1. ¿Cuál de los LP posee ventajas para programar paralelamente?

Go posee las ventajas más significativas para programación paralela entre los tres lenguajes analizados. Su modelo de concurrencia basado en goroutines y channels está integrado de forma nativa en el lenguaje, haciendo que la programación paralela sea más intuitiva y menos propensa a errores. Las goroutines son extremadamente ligeras (consumen solo unos kilobytes de memoria inicial) compared to traditional threads, permitiendo la creación de miles o incluso millones de ellas sin sobrecargar el sistema. El runtime de Go incluye un scheduler inteligente que maneja automáticamente la distribución de goroutines across available CPU cores, optimizando el uso de recursos sin necesidad de configuración manual.

2. ¿Para cada lenguaje elabore una tabla cuando usa Pool de Threads?

Aspecto	Java	C++	Go
Mecanismo Principal	ExecutorService Framework	Implementación manual con std::thread	Goroutines + Channels
Facilidad de Implementación	Alta - APIs bien definidas	Baja - Requiere código manual	Muy Alta - Integrado en el lenguaje
Overhead de Memoria	Alto (~1MB por thread)	Medio (~512KB por thread)	Muy Bajo (~2KB por goroutine)
Gestión de Recursos	Automática por JVM	Manual del desarrollador	Automática por Runtime

Sincronización	synchronized, Locks	mutex, condition_variable	Channels (comunicación en lugar de sincronización)
Escalabilidad	Buena (hasta miles de threads)	Limitada (cientos de threads)	Excelente (millones de goroutines)
Rendimiento	Bueno	Muy Bueno (cercano al hardware)	Excelente en concurrencia masiva
Prevención de Race Conditions	Manual (requires cuidado)	Manual (complejo)	Automática (por diseño de channels)
Ecosistema	Maduro y extenso	Estándar limitado	En crecimiento pero robusto
Curva de Aprendizaje	Moderada	Alta	Baja
Caso Ideal de Uso	Aplicaciones empresariales	Sistemas de alto rendimiento	Servicios concurrentes y APIs

En resumidas cuentas Java ofrece robustez y ecosistema maduro a costa de mayor consumo de recursos. C++ proporciona máximo control y rendimiento pero con alta complejidad. Go destaca en eficiencia y simplicidad para aplicaciones altamente concurrentes, siendo el más adecuado para proyectos que requieren manejar miles de operaciones paralelas simultáneamente.

RETROALIMENTACIÓN GENERAL

REFERENCIAS Y BIBLIOGRAFÍA