Control II: Lenguajes de Programación II

Profesor: Alonso Inostrosa Psijas
Fecha de Entrega: 28/10/2025

Aspectos Importantes:

1. **Lenguaje:** El desarrollo debe realizarse obligatoriamente en lenguaje Go, utilizando *Goroutines* y canales (*channels*) para la comunicación y sincronización.
2. **Entrega:** Debe crear un repositorio Git para respaldar su código. La entrega consiste en un único archivo .zip (que deberá subir a Aula Virtual) que contenga el código fuente (claramente comentado) y un archivo `Readme.md` con el enlace al repositorio, las instrucciones de compilación/ejecución y el análisis de rendimiento.
3. **Uso de IA y/o LLM:** ***No está permitido*** su uso.
4. **Grupos de Desarrollo:** Debe estar conformado por ***2 integrantes***.
5. **Entrevista con el Profesor:** Todos los integrantes del grupo ***deben tener absoluta claridad y entendimiento*** del código (aun cuando ciertas porciones no hayan sido desarrolladas por Ud.).

Enunciado del Problema:

Ejecución Especulativa

La **ejecución especulativa** es una técnica de optimización poderosa donde se aprovecha la concurrencia para realizar múltiples tareas en paralelo antes de saber cuál de ellas será la necesaria.
La idea es que mientras se determina la evaluación de una condición lógica (usualmente costosa en tiempos de ejecución), las posibles ramas de ejecución que se pueden derivar de ella, se ejecutan en paralelo a dicha evaluación. Una vez se ha determinado el resultado de la condición, se hace válida únicamente la rama correspondiente (anulando la otra), lo que mejora el rendimiento del programa.
Este control consiste en implementar directamente el patrón de ejecución especulativa. Para ello, se utilizarán tres funciones de cómputo intensivo proporcionadas en el Anexo. Su programa deberá ejecutar en hilos separados dos funciones, a la vez que se evalúa una condición lógica. Una vez que se ha determinado la condición, deberá descartar (y posiblemente cancelar la ejecución) del hilo que se ejecutó "especulativamente". Por el contrario, se considerarán válidos los resultados retornados por el hilo ejecutado en la rama correcta a la condición.
El grupo deberá analizar los tiempos de cómputo y la lógica de decisión respecto a una ejecución no especulativa para exhibir el beneficio de este patrón concurrente.

Problema a Resolver

El objetivo es que el programa principal lance dos *goroutines* que representen las ramas especulativas

(`rama_A` y `rama_B`) de ejecución. Mientras estas se "ejecutan", el hilo principal determinará la condición que decide cuál de las dos ramas es la correcta (mediante el valor de un umbral, que puede ser mayor o menor que el resultado retornado por una función). Finalmente, el programa deberá usar el resultado de la rama "ganadora" y asegurarse de que la otra rama finalice su ejecución de forma controlada para no desperdiciar recursos.

Requisitos de Implementación

1. **Simulación de Tareas**: El "cómputo" de cada rama se simulará **invocando las funciones -de alto costo computacional-** provistas en el anexo. Por ejemplo, la `rama_A` podría invocar `SimularProofOfWork(data, dificultad)` y la `rama_B` podría invocar `EncontrarPrimos(m)`.
2. **Evaluación de Decisión:** Según el resultado de la ejecución de la función `CalcularTrazaDeProductoDeMatrices(n int)` es que se determina cuál de las dos ramas se considera válida. Recuerde que su ejecución es costosa en términos de tiempo.
3. **Parámetros de Entrada**: El programa debe recibir, al menos, los siguientes parámetros desde la línea de comandos:
    a. `n`: Es la dimensión de dos matrices que se multiplican. Se supone que la evaluación de la condición toma un tiempo considerable, por lo que su tiempo se simula de esta forma.
    b. `umbral`: Valor utilizado para determinar si se ejecuta una u otra rama.
    c. `nombre_archivo`: nombre del archivo donde se registran las métricas de interés.
4. **Lógica de Sincronización**: Deben utilizarse canales (*channels*) para que las *goroutines* comuniquen sus resultados al hilo principal y para que este último pueda señalar la cancelación a la *goroutine* perdedora.
5. **Salida del Programa**: La salida debe ser clara, indicando al menos: los tiempos de inicio y fin de cada cómputo (en milisegundos, etc.), la rama seleccionada, el resultado obtenido, el tiempo total de ejecución, y cualquier valor que le permita realizar análisis de rendimiento para observar la mejora.

Análisis de Rendimiento:

Para cuantificar el beneficio de la ejecución especulativa, deberán comparar su rendimiento contra una ejecución secuencial.

1. **Ejecución Secuencial**: Implemente una función que simule la ejecución secuencial. Esta primero debe decidir la rama ganadora y, solo después, ejecutar el cómputo de dicha rama. Es decir, ejecución no especulativa.
2. **Mediciones**:
    - Ejecute 30 veces la simulación **especulativa** para un conjunto de inputs fijos (ej. input_A=5 para Proof-of-Work, input_B=500000 para primos, rama_ganadora="A").
    - Ejecute 30 veces la simulación **secuencial** con los mismos parámetros (elija la función de la rama correspondiente).
    - Calcule el **tiempo promedio** para ambas estrategias.

3. **Reporte Simple**: En el Readme.md, presente una tabla comparando los tiempos promedio y calcule el **Speedup** (mejora de velocidad) obtenido como:
    - $Speedup = Tpo\_Secuencial \ / \ Tpo\_Especulativo$
4. **Reporte Completo:** Presente un reporte completo de su estudio, utilice gráficas que permitan observar claramente las mejoras de **Speedup** y determinar sus conclusiones.

Sistema de Evaluación

La nota del Control 2 (C2) estará dada por una nota grupal (NG2) resultante de los programas implementados, y un factor de participación (FP2) definido por el profesor a partir de la entrevista con el grupo.

Nota C2 = NotaGrupal2 * FP2

Donde:

Nota Grupal (NG2)

Criterio	Puntos
Calidad del código y Entrega: Código claro, documentado y se ajusta a las condiciones (Git, Readme).	10
Implementación Concurrente: Uso correcto de Goroutines y canales para la lógica especulativa.	30
Lógica de Selección y Cancelación: El hilo principal selecciona y cancela correctamente las ramas.	20
Análisis de Rendimiento: Realiza y reporta correctamente la comparación con el modelo secuencial. Presenta conclusiones relevantes al problema planteado.	30
Manejo de Parámetros por línea de comandos: El programa se configura correctamente.	10
Total	100

Factor de Participación (FP1)

Nivel de Conocimiento Entrega	Factor
Excelente: Responde correctamente todas las preguntas del profesor.	1.1
Bueno: Responde correctamente más del 80% de las preguntas del profesor.	1.0
Insuficiente: Responde correctamente más del 50% de las preguntas del profesor.	0.8
Deficiente: Responde correctamente menos del 50% de las preguntas del profesor.	0.6

Anexos

Funciones de Cómputo Intensivo

Deben incluir y utilizar las siguientes dos funciones en su código para simular las cargas de trabajo, modifíquelas según corresponda y corrija cualquier error.

Función 1: Simulación de Prueba de Trabajo (Proof-of-Work) Blockchain: Esta función simula el proceso de "minado" de un bloque en una blockchain. Busca un valor (llamado `nonce`) que, al ser combinado con los datos del bloque y hasheado (con SHA-256), produce un hash que empieza con un número determinado de ceros. La dificultad define cuántos ceros se requieren. El tiempo de ejecución crece exponencialmente con la dificultad.

```
import (
    "crypto/sha256"
    "fmt"
    "strings"
)

// SimularProofOfWork simula la búsqueda de una prueba de trabajo de blockchain.
// La dificultad determina el número de ceros iniciales que debe tener el hash.
// La complejidad crece exponencialmente con la dificultad.
// Para un computador personal, dificultad 5-6 suele tardar unos segundos.
func SimularProofOfWork(blockData string, dificultad int) (string, int) {
    targetPrefix := strings.Repeat("0", dificultad)
    nonce := 0
    for {
        data := fmt.Sprintf("%s%d", blockData, nonce)
        hashBytes := sha256.Sum256([]byte(data))
        hashString := fmt.Sprintf("%x", hashBytes)

        if strings.HasPrefix(hashString, targetPrefix) {
            return hashString, nonce
        }
        nonce++
    }
}
```

Función 2: Búsqueda de Números Primos (Complejidad Polinomial Alta): Esta función encuentra todos los números primos hasta un número `max`. El algoritmo de prueba por división es simple pero ineficiente para números grandes.

```
// EncontrarPrimos busca todos los números primos hasta un entero max.
// Utiliza un enfoque de prueba por división, cuya complejidad es alta (aprox.
O(n^1.5)).
func EncontrarPrimos(max int) []int {
    var primes []int
    for i := 2; i < max; i++ {
        isPrime := true
        for j := 2; j*j <= i; j++ {
```

```
            if i%j == 0 {
                isPrime = false
                break
            }
        }
        if isPrime {
            primes = append(primes, i)
        }
    }
    return primes
}
```

Función 3: Multiplicación de matrices: Esta función realiza la multiplicación de dos matrices y retorna la suma de los elementos de la diagonal principal.

```
import "math/rand"

// CalcularTrazaDeProductoDeMatrices multiplica dos matrices NxN y devuelve la traza
// de la matriz resultante. La complejidad del cómputo es O(n^3).
func CalcularTrazaDeProductoDeMatrices(n int) int {
    // Se crean dos matrices con valores aleatorios para la multiplicación.
    m1 := make([][]int, n)
    m2 := make([][]int, n)
    for i := 0; i < n; i++ {
        m1[i] = make([]int, n)
        m2[i] = make([]int, n)
        for j := 0; j < n; j++ {
            m1[i][j] = rand.Intn(10)
            m2[i][j] = rand.Intn(10)
        }
    }

    // Se realiza la multiplicación y se calcula la traza en el proceso.
    trace := 0
    for i := 0; i < n; i++ {
        sum := 0
        for k := 0; k < n; k++ {
            sum += m1[i][k] * m2[k][i]
        }
        trace += sum
    }
    return trace
}
```