COMPUTACIÓN CONCURRENTE
PRÁCTICA 1

Prof. Manuel Alcántara Juárez
manuelalcantara52@ciencias.unam.mx

Alejandro Tonatiuh Valderrama Silva
at.valderrama@ciencias.unam.mx

José de Jesús Barajas Figueroa
jebarfig21@ciencias.unam.mx

Luis Fernando Yang Fong Baeza
fernandofong@ciencias.unam.mx

Ricchy Alain Pérez Chevanier
alain.chevanier@ciencias.unam.mx

Fecha de Entrega: 11 de Octubre de 2021 a las 23:59

Objetivo

En esta práctica se revisan algunos ejemplos para mostrar la ventaja de usar programas concurrentes. Para medir la mejora obtenida al usar operaciones de forma concurrente, se compara con la mejora obtenida a partir de la Ley de Amdahl.

Indicaciones generales

El primer ejercicio sirve para ejemplificar una condición de carrera. Los siguientes tres ejercicios tratan sobre resolver un problema mediante un programa que puede implementarse de forma secuencial o concurrente. En cada ejercicio se hará una comparación entre los tiempos de ejecución de la forma secuencial y la forma concurrente. Además, la aceleración obtenida en la solución concurrente será comparada con lo que nos dice la ley de Amdahl.

Para hacer las comparaciones, en cada ejercicio (2-3) realizarás una tabla de la siguiente forma

# hilos	Aceleración teórica	Aceleración obtenida	% Código paralelo

donde anotarás la aceleración calculada con la ley de Amdahl, la aceleración que obtuvo tu programa concurrente y el porcentaje de código paralelo de tu programa, también calculado con la ley de Amdahl. Finalmente, en una gráfica mostrarás la ventaja de la versión concurrente (si es que la hubo).

1. Contador Compartido

Abre el archivo *Contador.java* que viene junto con este documento. En el método *main* se crea un objeto del tipo Contador y se ejecutan dos hilos actualizando el contador compartido. Después de que cada hilo actualizó el contador 100000 veces termina su ejecución y el valor del contador se imprime, cuya salida esperada es 200000. Compila y ejecuta el programa.

Como podrás ver, al ejecutar el programa el resultado que se obtiene puede no ser el que uno esperaba. La razón de esto es que el programa tiene una **condición de carrera**, debido a que la operación `contador++` no es una operación básica, sino que después de compilarla se verá algo de la siguiente forma:

```
int tmp = contador;
contador = tmp + 1;
```

Debido a ello puede darse el caso de que un hilo escriba un nuevo valor en el contador cuando el otro ya lo ha actualizado varias veces en ese tiempo. Tu trabajo en este ejercicio, es encontrar cuál es el menor valor posible que puede imprimir el programa mostrando una ejecución en donde suceda eso. Para lograrlo podrás hacer algunas pausas al programa en momentos específicos para tener la oportunidad de conseguir el resultado correcto.

Así que lo primero que debes hacer es remplazar la instrucción `contador++` por el código de la caja gris anterior y añadir la siguiente instrucción el número de veces que creas conveniente dentro del for:

```
if(id == # && i == #) Thread.sleep(#)
```

Cada línea retrasará un hilo en una iteración específica un número de milisegundos. Para ser capaz de decidir qué hilo se esta ejecutando, se tiene que añadir el siguiente código al inicio del método **run()**

```
String nombre = Thread.currentThread().getName();
int id = (nombre.equals("hilo1")) ? 1 : 2;
```

El método Thread.sleep() puede ser interrumpido, por lo que puede lanzar la excepción `InterruptedException`, la cual debes de capturar. Esta excepción se utiliza para cancelar de manera segura los hilos pero no trataremos este tema en esta práctica, solo la capturaremos descomentando el bloque **try-catch** en el método `run()`.

OPCIONAL: Para resolver la condición de carrera se puede restringir la forma en que las operaciones de los dos hilos operan sobre la variable compartida. Esta propiedad se le conoce como ***Exclusión mútua*** y una manera de lograrlo en Java es usando la palabra reservada synchronized. Remplaza la línea `contador++` con este código:

```
synchronized (this) {
    contador++;
}
```

Vuelve a compilar y ejecuta el ejemplo y verás que ahora cualquier ejecución del programa siempre imprimirá el valor de 200000. Como puedes darte cuenta no hemos alterado las operaciones que realiza el programa, sino más bien el orden en que estas son ejecutadas, con el fin de garantizar un resultado correcto.

2. Multiplicación de matrices

Es posible realizar el producto de dos matrices usando varios hilos de forma independiente. Primero recordemos cómo se obtiene el producto de dos matrices:

Sean $A = (a_{ij})$ una matriz de $n \times r$ y $B = (b_{ij})$ una matriz de $r \times m$. La matriz producto $C = AB$ es una matriz de tamaño $n \times m$ que está definida por las entradas

$$c_{ij} = \sum_{k=1}^{r} a_{ik} \cdot b_{kj},$$

es decir, el elemento c_{ij} es igual al producto punto entre la i-ésima fila de A y la j-ésima columna de B. Directamente de esta definición obtenemos un algoritmo secuencial para calcular AB, donde el cálculo principal es el producto punto, que se vería similar al siguiente código.

```
for(k = 0; k < r; k++)
    C[i][j] += A[i][k] * B[k][j]
```

Para calcular el producto de forma concurrente usando varios hilos de ejecución, podemos asignar parejas de la forma (i, j) a cada hilo y este se encargará de obtener la entrada c_{ij}. Por ejemplo, para obtener

$$C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} 2 & 6 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 5 & 3 \end{bmatrix}$$

podemos usar dos hilos: el primero puede obtener las entradas c_{11}, c_{12} y el segundo las entradas c_{21}, c_{22}, o podríamos usar cuatro hilos y que cada uno obtenga una entrada de C.

Para el caso de una matriz de $n \times n$ tenemos la opción de partirla en bloques (submatrices) y luego hacer que cada hilo se encargue de calcular las entradas de un bloque. Para simplificar el ejercicio usaremos únicamente matrices de $n \times n$ donde n es par, y cuatro hilos de ejecución.

Especificación del programa

Crea un programa que obtenga el cuadrado de una matriz A de tamaño $n \times n$ (puedes suponer que n es par).

1. En la forma secuencial se calculan las entradas c_{ij} de A^2 usando tres ciclos anidados.

2. En la forma concurrente usarás **n** hilos de ejecución. Para esto hay que dividir la matriz en **n** bloques del mismo tamaño, y cada hilo obtendrá los coeficientes de un bloque. Cada hilo puede realizar su cálculo de forma independiente, sin tener que esperar alguna operación de otro hilo.

3. Para comparar los tiempos de ejecución deberas usar 1, 2, 4 y 8 hilos, graficar resultados.

Entrada. Tu programa se debe ejecutar así

```
$ java Matriz hilos archivo
```

Donde `archivo` es el nombre del archivo que contiene los coeficientes de A, y el valor de `hilos` es el numero de hilos que se van a ejecutar(1 hilo es la forma secuencial) .

El archivo de entrada incluye los coeficientes de A, con las columnas separadas por un espacio en blanco y las filas separadas por un salto de línea, como el siguiente ejemplo:

```
82 12 3
65 100 3
0 78 21
```

Salida. Tu programa debe guardar el resultado de A^2 en un archivo con nombre `producto.txt`. Antes del resultado deberá aparecer una línea indicando el tiempo que tardó en realizarse el producto, medido en milisegundos.

Compara tus soluciones

Usa los archivos de prueba incluidos en la carpeta `matrices` para probar tu programa. Se incluyen matrices de tamaños 10×10, 100×100 y 1000×1000. Para cada prueba repite el cálculo 20 veces y obtén el promedio del tiempo tardado, tanto en la opción secuencial como la opción concurrente. Con estos datos elabora una gráfica que compare tamaño de matriz frente a tiempo promedio. La gráfica debe incluir tanto los datos de la opción secuencial como los de la opción concurrente. En la figura 1 se muestra un ejemplo.

Anexa tu comparación con la ley de Amdahl.

3. Sopa de letras

En este ejercicio resolverás una sopa de letras. A partir de un texto como el siguiente

```
O I U Q W E J A A D C
A S D A S D C W E E H
O I M O I O D P F I O
```

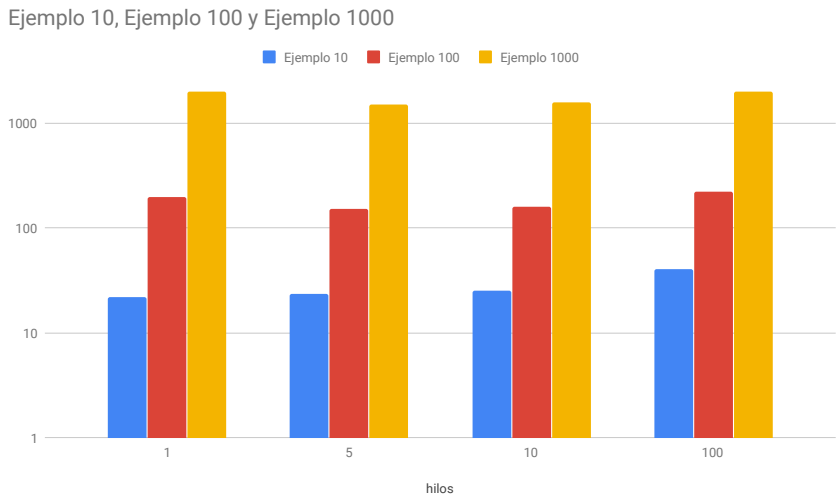

Figura 1: Ejemplo de comparación de tiempos. La opción secuencial es la de 1 hilo.

```
I E P E R R O V N C L
C N J D K D M C L D A
A R O D A T U P M O C
```

y una lista de palabras como esta

```
PERRO
COMPUTADORA
HOLA
ROCA
```

deberás poder encontrar la posición donde empieza la palabra encontrada junto con la dirección que siguen las letras para formar la palabra.

Podemos considerar a la sopa de letras como un arreglo bidimensional donde cada entrada guarda una letra. Para indicar las direcciones usaremos las letras N, S, E, O, NE, NO, SE, SO (de Norte, Sur, Este, etc.). Con estas convenciones, la solución del ejemplo anterior se vería así:

```
PERRO (3,2) E
COMPUTADORA (5,10) O
HOLA (1,10) S
ROCA (3,4) NE
```

Para encontrar una palabra completa primero buscamos en el arreglo hasta encontrar la primera letra de la palabra. Si la encontramos en la posición (i, j), ahora revisamos cada dirección posible para verificar si se encuentra la palabra completa. Si no se encontró en (i, j), seguimos recorriendo el arreglo.

El paso de buscar en una dirección dada se puede hacer de forma recursiva. Por ejemplo, si estamos buscando `PERRO`, estamos en la posición (i, j) (donde hay una P) y queremos revisar hacia el Norte, entonces podemos buscar la palabra `ERRO` empezando en la posición $(i-1, j)$; si no coincide la `E` terminamos la búsqueda en esta dirección, si coincide la `E` ahora buscaremos `RRO` desde la posición $(i-2, j)$. El caso base es cuando la palabra buscada es solo una letra.

Programa

Para la implementación se usará una función recursiva con la firma

```
boolean busca(String palabra, int x, int y, int direccion)
```

que sirve para verificar si en la sopa de letras aparece la palabra empezando en la posición (x, y) en la dirección indicada. Desde luego, hay que revisar en las ocho posibles direcciones por cada posición en la que inicia la búsqueda.

1. En la opción secuencial tu programa irá revisando letra por letra en el arreglo, buscando en las ocho direcciones posibles.
2. La versión concurrente usará n hilos, deberas dividir el conjuntos de palabras en n subconjuntos, de preferencia de la misma cardinalidad, cada hilo va a tener asignado un subconjunto de palabras y va a tener que buscar en toda la matriz estas palabras, recuerden, una misma palabra no puede exisitir en dos subconuntos diferentes
3. Esta implementación es solo una idea de como pueden resolver el ejercicio, la implementación es libre, recuerden que debe funcionar con n hilos y ser concurrente
4. Deben de hacer una grafica de comparación de tiempos para 1, 2, 4 y 8 hilos
    - Hilo 0. Direcciones N, S.
    - Hilo 1. Direcciones E, O.
    - Hilo 2. Direcciones NE, SO.
    - Hilo 3. Direcciones NO, SE.

Entrada. Se ejecutará el programa de la siguiente forma

```
$ java Sopa hilos archivo_sopa archivo_palabras
```

donde `hilos` es como en los otros ejercicios, `archivo_sopa` es el archivo que contiene la sopa de letras y `archivo_palabras` es el archivo con las palabras a buscar.

Salida. El programa mostrará en pantalla la solución, que es una lista con las palabras junto con la ubicación y la dirección correspondientes. Al final se mostrará el tiempo de ejecución de la búsqueda.

Compara tus soluciones

Para hacer la comparación usa los ejemplos que se incluyen dentro de la carpeta `sopas`. De la misma manera que en el primer ejercicio, grafica los tiempos de ejecución y haz la comparación con la ley de Amdahl.