	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): 12 – 13

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 26

Semestre: 2023-1

Fecha de entrega: 14 de diciembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 12 – 13.

Algoritmos paralelos

OBJETIVO

- El estudiante conocerá y aprenderá a utilizar algunos de los patrones paralelos para el acceso a datos y distribución de tareas que le servirán en el diseño e implementación de un algoritmo paralelo en computadoras con arquitectura de memoria compartida.

ACTIVIDADES

- Realizar ejemplos del funcionamiento de directivas de OpenMP en el lenguaje C, tales como los constructores *critical*, *section*, *single*, *master*, *barrier* y las cláusulas *reduction* y *nowait* que permitirán implementar algún patrón paralelo.
- Realizar programas que resuelvan un problema de forma paralela utilizando distintas directivas de OpenMP.

DESARROLLO

Parte 1. Ejercicios de la guía

Practica 12

Ejemplo *critical*

Se tiene la siguiente función para encontrar el valor máximo entero de entre los elementos de un arreglo unidimensional de *n* elementos.

```
int buscaMaximoSerial(int *a, int n)
{
    int max, i;
    max = a[0];
    for (i = 1; i < n; i++)
    {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
```

Para realizar una versión paralela se puede usar utilizar el constructor *for* para dividir las iteraciones del ciclo entre los hilos por defecto. La función queda:

```
int buscaMaximoFor(int *a, int n)
{
    int max, i;
    max = a[0];
#pragma omp parallel
    {
#pragma omp for
        for (i = 0; i < n; i++)
        {
            if (a[i] > max)
                max = a[i];
        }
    }
    return max;
}
```

Además, es posible anidar los constructores *parallel* y *for* de la siguiente manera (cuando lo que está dentro de la región paralela solo es una estructura *for* y esta es posible paralelizarla):

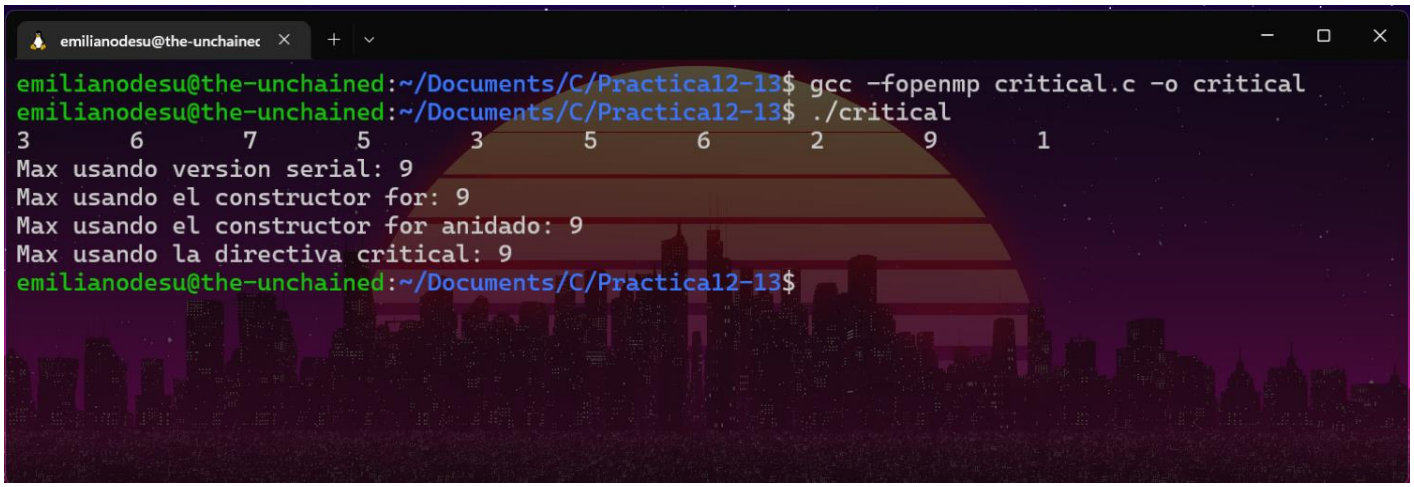
```
int buscaMaximoParallelFor(int *a, int n)
{
    int max, i;
    max = a[0];
#pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        if (a[i] > max)
            max = a[i];
    }
    return max;
}
```

Así, cada hilo trabaja con diferentes partes del arreglo, pero puede suceder el caso donde dos de los hilos encuentren la proposición *a[i]>max* verdadera, e intentarán actualizar la variable *max* donde se encuentra almacenado el valor máximo encontrado, ocasionando una condición de carrera. Una forma de arreglar este problema es indicar a los hilos que deben modificar uno a la vez la variable *max*. Esto se consigue con el constructor *critical*:

```
int buscaMaximoCritical(int *a, int n)
{
    int max, i;
    max = a[0];
#pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        if (a[i] > max)
        {
#pragma omp critical
            {
                if (a[i] > max)
                    max = a[i];
            }
        }
    }
    return max;
}
```

Es importante señalar que, aunque cada hilo espera su turno, se debe revisar nuevamente si *a[i]>max* cuando cada hilo entra a actualizar a la variable *max*. De no hacerse, el valor de *max* que utiliza uno de los hilos en espera puede que sea menor al valor de *a[i]* que analiza en ese momento debido a que fue actualizado por uno de los hilos que anteriormente entró a modificarlo, dando un resultado incorrecto.

Ejecución del ejemplo de uso de *critical*



Se creó un arreglo unidimensional de 10 elementos, con el cual se llamó a cada una de las funciones antes mencionadas, para obtener su valor máximo entero.

Ejemplo *reduction*

Para explicar el funcionamiento de esta clausula se usa como ejemplo una función que realiza el producto punto entre dos vectores de *n* elementos, que se realiza como en la siguiente figura:

$$\mathbf{A}\cdot\mathbf{B} = \begin{bmatrix} A_1 & A_2 & \dots & A_n \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{bmatrix} = A_1B_1 + A_2B_2 + \dots + A_nB_n$$

La función para obtenerlo (de forma serial) es:

```
double prodpuntoSerial(double *a, double *b, int n)
{
    double res = 0;
    int i;

    for (i = 0; i < n; i++)
    {
        res += a[i] * b[i];
    }
    return res;
}
```

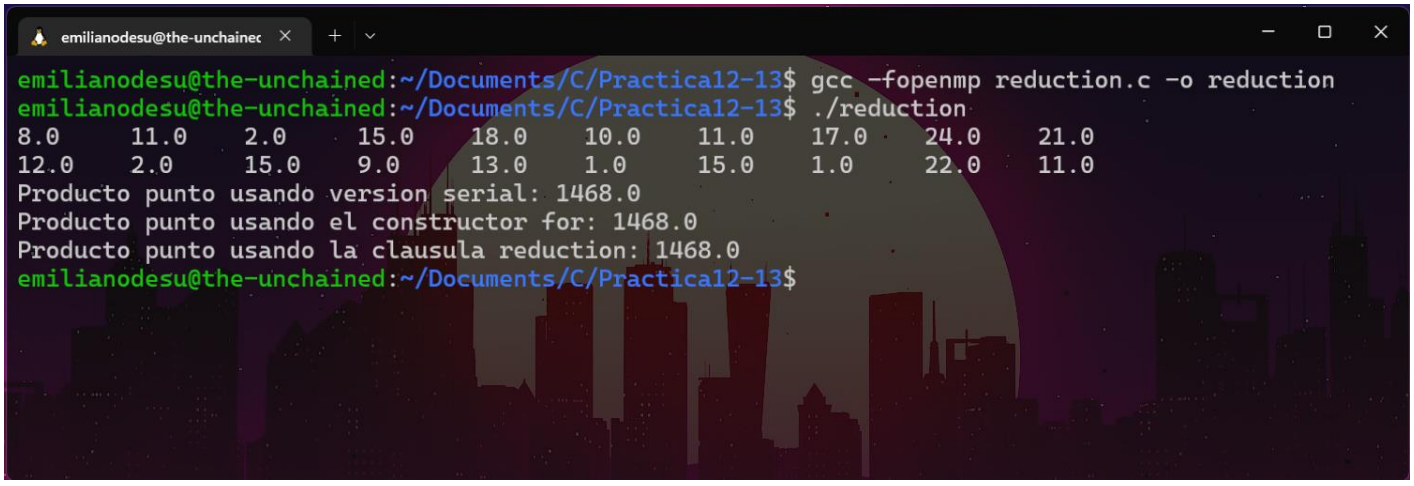
Para implementar una solución paralela se puede hacer que *n* hilos cooperen en la solución. Cada hilo puede trabajar con diferentes elementos de los vectores *A* y *B* y cada uno obtener resultados parciales. Además, se utilizará un arreglo *resp[]* del tamaño del número de hilos que se quiere tener en la región paralela y en cada elemento guardar las soluciones parciales de cada hilo. Al final, solo un hilo suma esos resultados parciales para obtener el resultado final. La solución queda:

```
double prodpuntoFor(double *a, double *b, int n)
{
    int n_hilos = 8, i, tid, nth;
    double res = 0, resp[n_hilos];
#pragma omp nthreads(n_hilos) parallel private(tid)
    {
        tid = omp_get_thread_num();
        resp[tid] = 0;
#pragma omp for
        for (i = 0; i < n; i++)
        {
            resp[tid] += a[i] * b[i];
        }
        if (tid == 0)
        {
            nth = omp_get_num_threads();
            for (i = 0; i < nth; i++)
                res += resp[i];
        }
    }
    return res;
}
```

Ahora, en lugar de utilizar un arreglo para almacenar resultados parciales se puede utilizar la cláusula *reduction*. Esta clausula toma el valor de una variable aportada por cada hilo y aplica la operación indicada sobre esos datos para obtener un resultado. Así, en este ejemplo, cada hilo aporta su cálculo parcial *res* y después se sumarán todos los *res* y el resultado quedará sobre la misma variable. Usando la cláusula reduction se tiene:

```c
double prodpuntoReduction(double *a, double *b, int n)
{
    double res = 0;
    int i;
#pragma omp parallel for reduction(+: res)
    for (i = 0; i < n; i++)
    {
        res += a[i] * b[i];
    }
    return res;
}
```

Ejecución del ejemplo de uso de *reduction*



Para este ejemplo se crearon dos arreglos de 10 elementos con números aleatorios (casteados a *double*). Se invocó a cada una de las tres funciones para obtener el producto punto de los vectores.

Ejemplo *barrier* y *master*

El constructor *barrier* coloca una barrera explícita para que cada hilo espere hasta que todos lleguen a la barrera. El constructor *master* permite definir un bloque básico de código dentro de una región paralela; estas actividades son hechas por el hilo maestro y no se tiene una barrera implícita, es decir, los hilos restantes no esperan a que el hilo maestro termine la actividad.

En el siguiente código se genera una región paralela, y dentro de esta, con el constructor *for* cada hilo asigna valores a diferentes elementos del arreglo *a*. Después, con el constructor *master* se indica que solo el hilo maestro imprima el contenido del arreglo. Como el constructor *master* no tiene barrera implícita, se usa el constructor *barrier* para lograr que todos los hilos esperen a que se imprima el arreglo antes de modificarlo. Se tiene el siguiente código:

```
#include <stdio.h>

int main()
{
    int a[5], i;
#pragma omp parallel
    {
#pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

#pragma omp master
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);

#pragma omp barrier

#pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

Ejecución del ejemplo de uso de *barrier* y *master*

Usando *barrier* la ejecución es más segura porque las operaciones que se realizan después de la impresión del arreglo (a cargo del hilo maestro) no inician hasta haber terminado el hilo maestro.

Actividad 1

La implementación de los programas se realizó como parte de la demostración de los ejemplos.

Actividad 4

Para el siguiente programa obtener dos versiones paralelas, una utilizando el constructor *section* y otra con el constructor *for*.

```c
#include <stdio.h>
#include <omp.h>
#define N 100000

int main(int argc, char *argv[])
{
    double empezar, terminar;
    int i, j;
    float a[N], b[N], c[N], d[N], e[N], f[N];

    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    empezar = omp_get_wtime();

    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i];

    for (j = 0; j < N; j++)
        d[j] = e[j] + f[j];

    terminar = omp_get_wtime();
    printf("TIEMPO = %lf\n", terminar - empezar);
}
```

Modificación con el constructor *sections*

```
#include <stdio.h>
#include <omp.h>
#define N 100000

int main(int argc, char *argv[])
{
    double empezar, terminar;
    int i, j;
    float a[N], b[N], c[N], d[N], e[N], f[N];

    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    empezar = omp_get_wtime();

#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            for (i = 0; i < N; i++)
                c[i] = a[i] + b[i];

#pragma omp section
            for (j = 0; j < N; j++)
                d[j] = e[j] + f[j];
        }
    }
    terminar = omp_get_wtime();
    printf("TIEMPO = %lf\n", terminar - empezar);
}
```

Se nota que hay dos ciclos *for* que pueden ser ejecutados cada uno en diferentes hilos usando *sections*. Entonces, cada sección contiene un ciclo *for*

Modificación con el constructor *for*

```
#include <stdio.h>
#include <omp.h>
#define N 100000

int main(int argc, char *argv[])
{
    double empezar, terminar;
    int i, j;
    float a[N], b[N], c[N], d[N], e[N], f[N];

    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;

    empezar = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp for
        for (i = 0; i < N; i++)
            c[i] = a[i] + b[i];

#pragma omp for
        for (j = 0; j < N; j++)
            d[j] = e[j] + f[j];
    }
    terminar = omp_get_wtime();
    printf("TIEMPO = %lf\n", terminar - empezar);
}
```

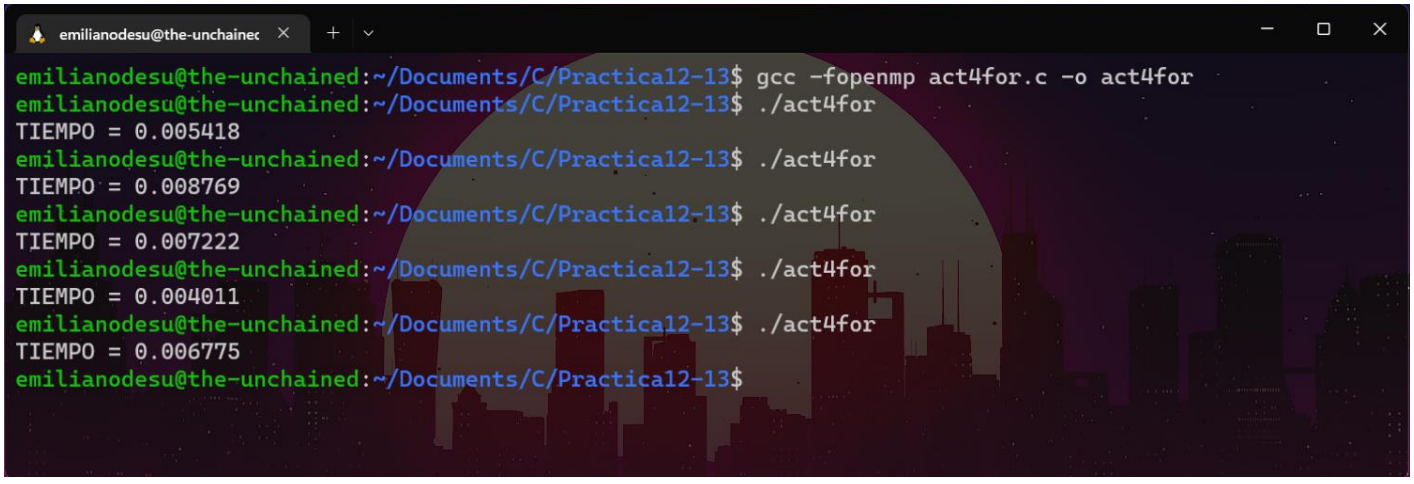
En este caso, cada *for* se ejecuta con su respectivo constructor. Esto implica que se dividen las iteraciones entre el total de hilos que ejecutan el programa y no uno por sección como en el caso anterior. Intuitivamente podemos pensar que esta modificación será más rápida.

Ejecución del programa paralelo versión *sections*

Se ejecutó un total de 5 veces para obtener un promedio de tiempo

$$Promedio_{sections} = 0.0080294[s]$$

Ejecución del programa paralelo versión *for*



Se ejecutó un total de 5 veces para obtener un promedio de tiempo

$$Promedio_{for} = 0.006439[s]$$

¿Cuál de las dos versiones tarda más tiempo?

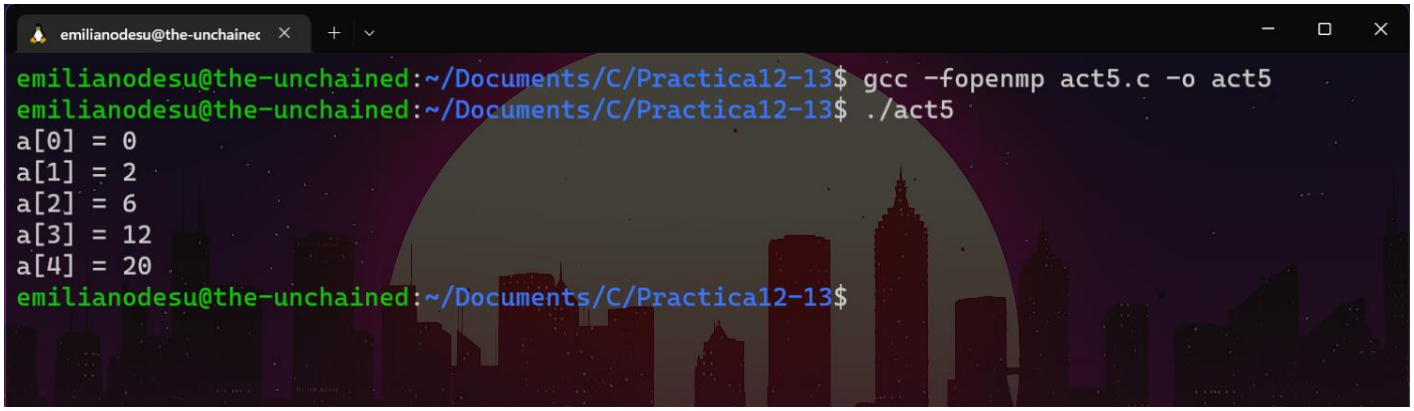
Como se mencionó antes, era de esperarse que la versión con el constructor *for* se ejecute en menos tiempo porque las iteraciones de los ciclos se reparten entre el número total de hilos y en la versión con *sections* un hilo ejecuta por completo el ciclo que le corresponde.

Actividad 5

Probar el siguiente ejemplo:

```c
#include <stdio.h>

int main()
{
    int a[5], i;
#pragma omp parallel
    {
#pragma omp for
        for (i = 0; i < 5; i++)
            a[i] = i * i;

#pragma omp master
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);

#pragma omp barrier

#pragma omp for
        for (i = 0; i < 5; i++)
            a[i] += i;
    }
}
```

¿Qué sucede si se quita la barrera?



El constructor *master* no tiene una barrera implícita, es decir, los demás hilos no esperan a que terminen las instrucciones del bloque *master*. Como consecuencia, el bloque a continuación con el constructor *for* empieza a modificar los valores del arreglo mientras este está siendo imprimido por el hilo principal, causando una salida no deseada.

Si en lugar de utilizar el constructor *master* se utilizara *single,* ¿qué otros cambios se tienen que hacer en el código?

Cambiando el *master* por *single* deja de ser necesario colocar un *barrier* dado que el constructor *single* tiene una barrera implícita. El resto del código permanece igual.

```
#pragma omp single
        for (i = 0; i < 5; i++)
            printf("a[%d] = %d\n", i, a[i]);
            // Hilos esperan
```

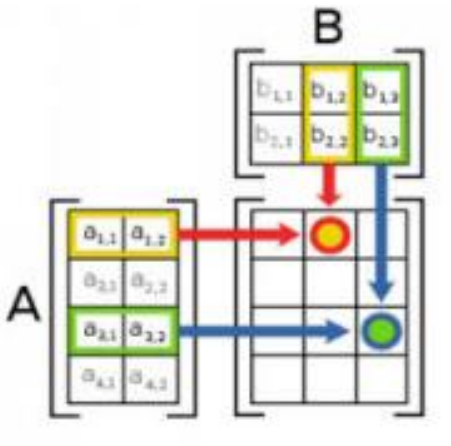
Practica 13

Ejemplo 1. Multiplicación de matrices

Recordando la fórmula para obtener un elemento de *C*, donde *C* es el producto de dos matrices *A* y *B* de orden *nxm* y *mxr*, es

$$C_{i\,j} = \sum_{k=0}^{n-1} a_{ik} b_{k\,j}$$

para $0 \leq i,\ j < n$



Para dividir este problema se realiza un paralelismo de datos, donde cada hilo trabaja con datos distintos, pero con el mismo algoritmo.

El planteamiento de solución paralela es considerar que, en un ambiente con varios hilos, cada uno calcula *i* renglones de *C*, requiriendo *i* renglones de *A* y toda *B*. El algoritmo secuencial queda de la siguiente manera:

```
for (i = 0; i < NRA; i++)
    {
        for (j = 0; j < NCB; j++)
            for (k = 0; k < NCA; k++)
                c[i][j] += a[i][k] * b[k][j];
    }
```

Con vista en lo anterior, las iteraciones del primer ciclo *for* (iésimo renglón de *A*) se pueden dividir de manera que cada hilo tome *n* diferentes renglones de la matriz *A*. Cada hilo usará distintos valores del índice *i* para la lectura de *A* y para el cálculo de *C*.

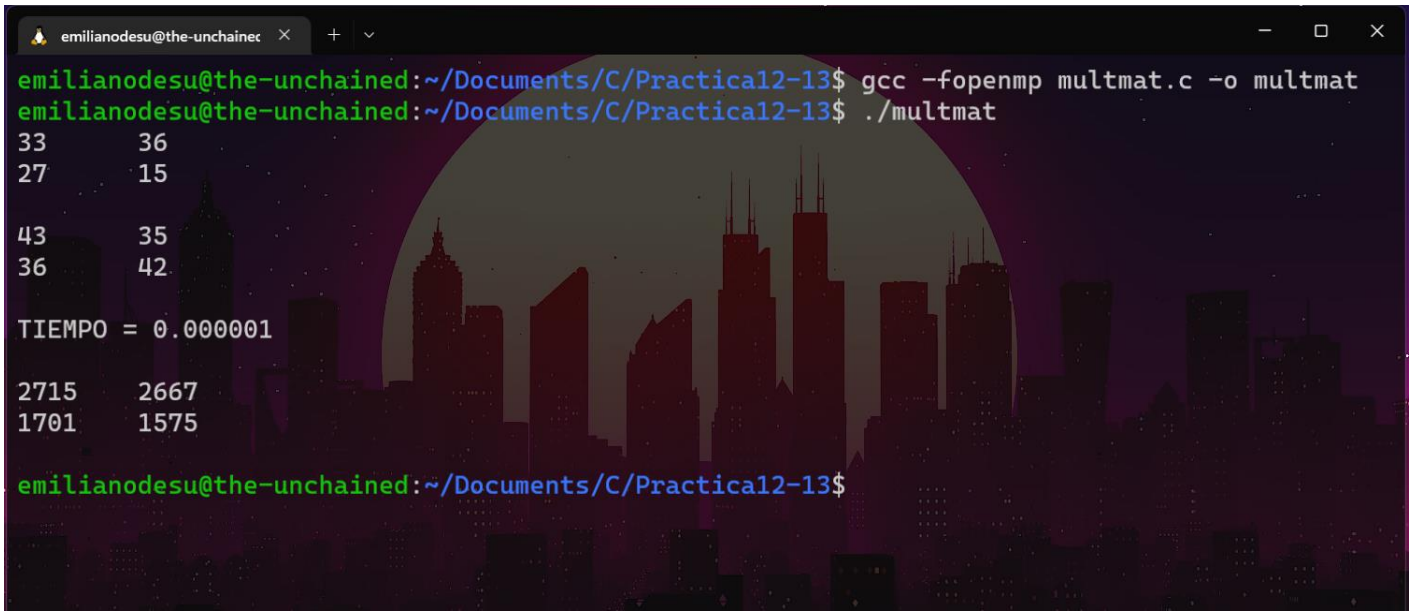
También es necesario considerar qué variables deben ser privadas. En este caso, *A*, *B*, y *C* deben ser compartidas ya que en ningún momento los hilos tratan de escribir un elemento compartido, pero *j*, y *k* deben ser privadas porque cada hilo las modifica al realizar los ciclos internos. El segmento paralelizado queda:

```
#pragma omp for private(j, k)
            for (i = 0; i < NRA; i++)
            {
                for (j = 0; j < NCB; j++)
                    for (k = 0; k < NCA; k++)
                        c[i][j] += a[i][k] * b[k][j];
            }
```

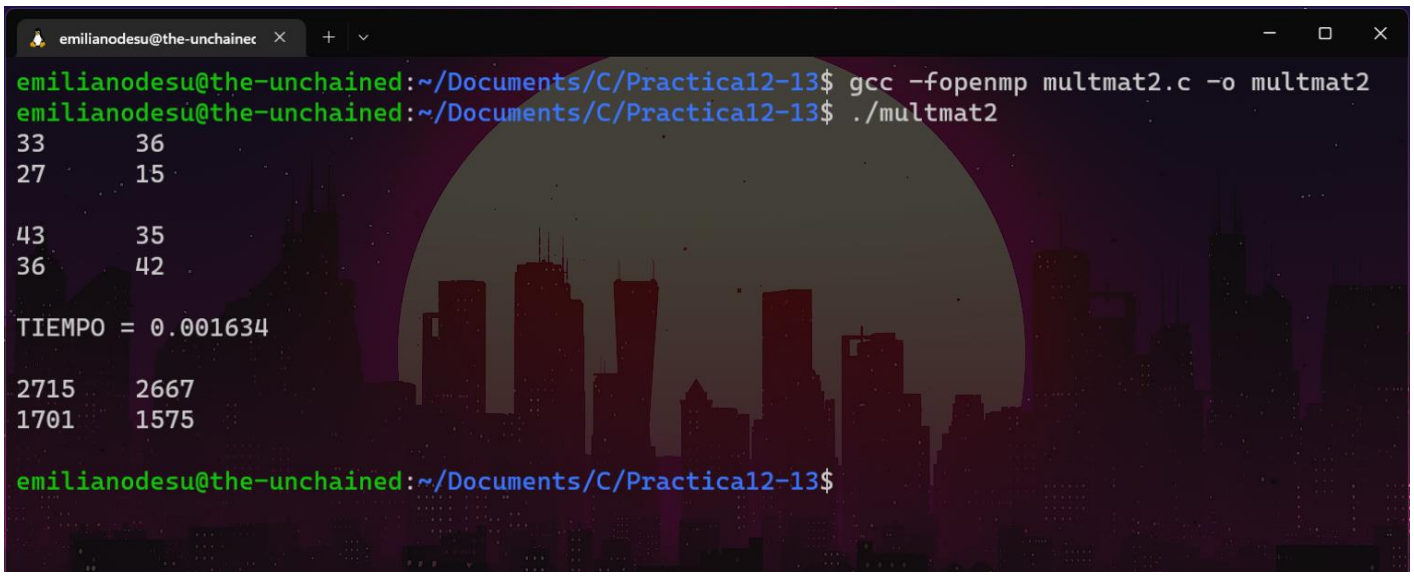
Actividad 1

Completar la versión serie y paralela del ejemplo1 explicado en la guía y medir el tiempo de ejecución de ambas versiones utilizando matrices de orden 500x500.

Comprobando el funcionamiento de la versión serie con una matriz de 2x2



Comprobando el funcionamiento de la versión paralela con una matriz 2x2

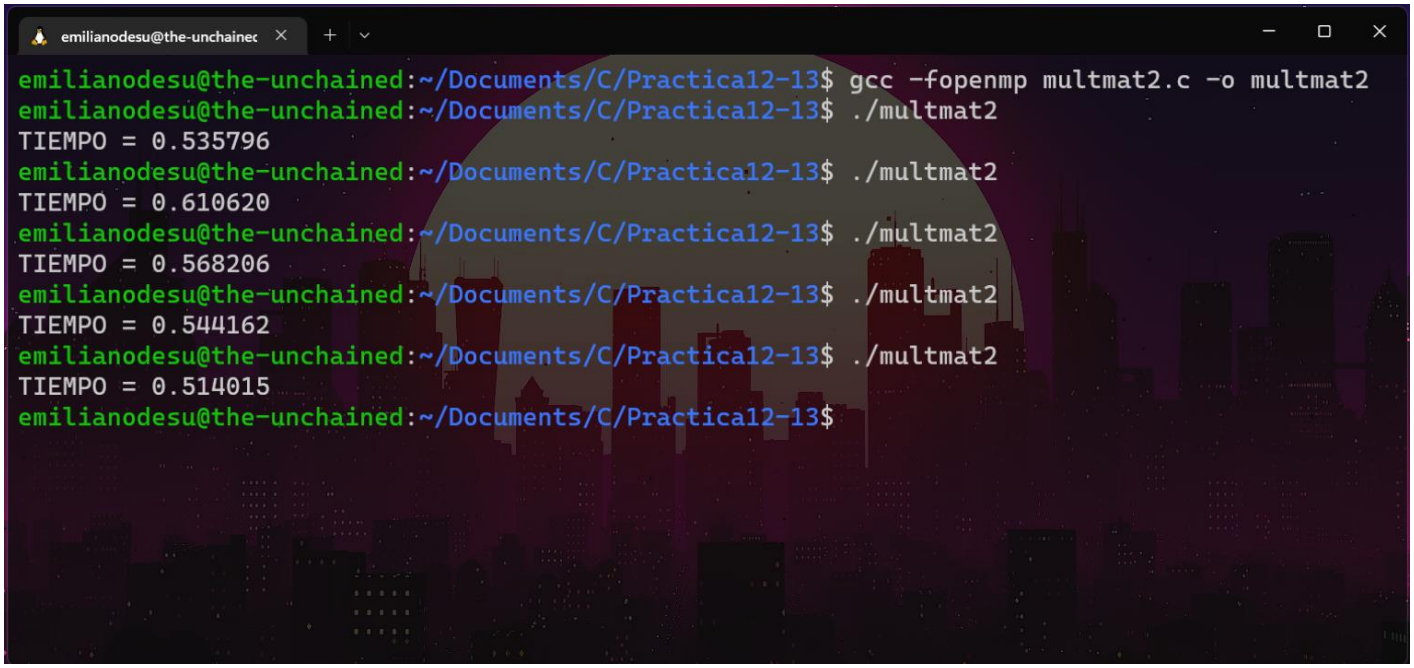


Los resultados de las multiplicaciones son correctos, así que para las matrices de 500x500 se omitirá la impresión de las matrices para mayor eficiencia.

Tiempos de la versión en serie

$$Promedio_{serie} = 1.6396188[s]$$

Tiempos de la versión paralela



$$Promedio_{paralela} = 0.5545598[s]$$

Podemos observar que en el ejemplo con matrices de 2x2 el tiempo con la versión en serie fue más rápido que la versión en paralelo, pero con matrices de 500x500 el tiempo de la versión paralela mejora bastante y es cuando se ve la diferencia entre una manera y la otra.

Parte 2. Programación paralela y concurrente en Java

2.1 Semáforos en Java

a) Revisa la API de Java y realiza una breve descripción de los métodos más importantes de esta clase (*Semaphore*) (al menos 5).

1. ***acquire()*:** Adquiere un permiso de este semáforo si hay al menos alguno disponible. A su vez, se reduce el numero de permisos disponibles en uno. Si el hilo actual es interrumpido entonces se arroja la excepción *InterruptedException*.
2. ***drainPermits()*:** Adquiere y regresa el numero actual de permisos disponibles.
3. ***reducePermits(int reduction)*:** Reduce el número disponible de permisos por la reducción especificada.
4. ***release()*:** Libera un permiso, incrementando el número de permisos disponibles del semáforo en uno.
5. ***getQueuedThreads()*:** Regresa una colección que contiene los hilos que están en espera de adquirir un permiso. La colección es una estimación dado que los hilos pueden cambiar en cualquier momento y no se listan en ningún orden en particular.

b) Revisa las clases del proyecto Semáforos y realiza un análisis indicando los siguientes aspectos:

- **Explica qué hacen los programas**
- **Explica como se aplica el concepto de semáforo**
- **Indica en que aplicaciones se podrían utilizar**

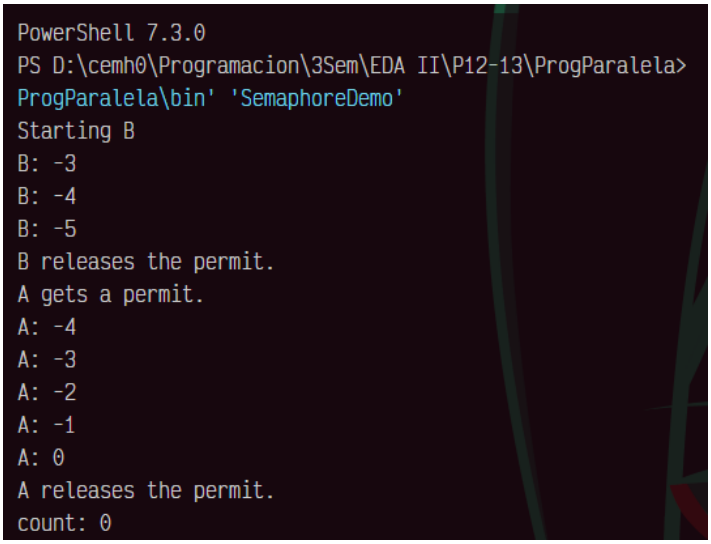
SemaphoreDemo

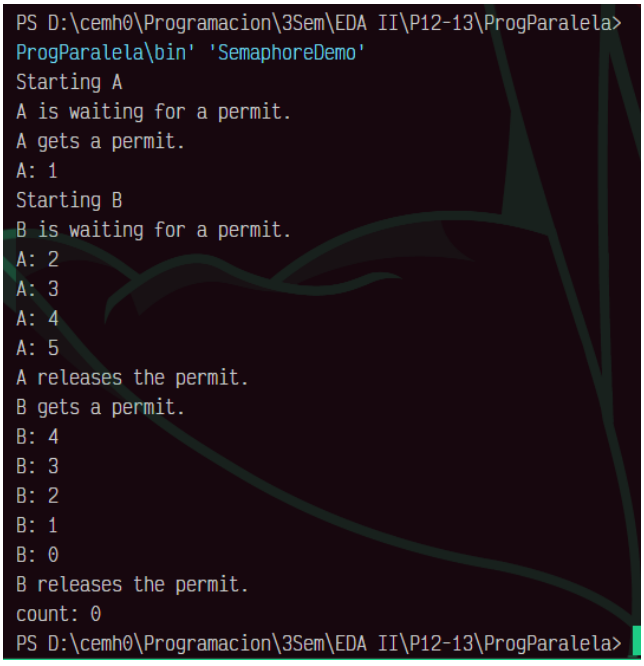
En este ejemplo se tiene una clase llamada *Shared* que contiene un atributo estático, *count*. Este atributo será de uso compartido para dos instancias de la clase *MyThread*, que implementa hilos. En términos generales, en el programa se usa un semáforo para controlar el acceso a la variable compartida *count*.

En la clase *MyThread* se incluye un semáforo como atributo, además del nombre del hilo. En el método *run()* de la clase, *Shared.count* es incrementada 5 veces por el hilo “A” y decrementada 5 veces por el hilo “B”. Para evitar que ambos hilos accedan a la variable *count*, los hilos solicitan acceso al recurso usando el método *acquire()* y, el semáforo debe otorgar un permiso a la vez a los hilos. Después de obtener el acceso y haber realizado sus acciones, el hilo que obtuvo el permiso lo libera para que el otro hilo pueda proceder.

Es importante notar que al principio de la ejecución existe una condición de carrera entre los hilos A y B, pero al haber ganado el permiso, el hilo que se ejecuta realiza todas sus instrucciones y luego el otro. Por ello, pueden salir dos salidas diferentes del programa dependiendo de que hilo haya obtenido el permiso primero, pero el resultado final siempre será 0. Es decir, ambos hilos ejecutan todas sus instrucciones en su respectivo “turno”.

Salida del programa





Este mecanismo de sincronización se aplica para permitir el acceso a diferentes partes de programas que involucren secciones críticas, es decir, donde se manipulen variables o recursos que deben ser accedidos con especial precaución. De esta forma solo un proceso puede acceder al recurso a la vez.

SemaphoreTest

En este ejemplo, está la clase *SemaphoreTest*, que contiene un semáforo estático de 4 permisos y una clase interna: *PruebaHilos*. En la clase *PruebaHilos* se implementa un hilo, cuyo método *run()* hace lo siguiente:

1. Enseña la cantidad de permisos disponibles al iniciar la ejecución.
2. El hilo solicita un permiso.
3. Con el permiso dado, se ejecutan 5 instrucciones dentro de un ciclo *for*.
4. Finalmente, libera el permiso y vuelve a imprimir el número de permisos disponibles en el momento.

Lo interesante sucede en el método *main*, donde se crean 6 hilos “A-F” y se ejecutan. Cada ejecución del programa puede dar una salida diferente por lo que analizamos un ejemplo de ejecución:

```
Total available Semaphore permits : 4
A : acquiring lock...
A : available Semaphore permits now: 4
B : acquiring lock...
D : acquiring lock...
A : got the permit!
```

El hilo A gana el permiso.

```
C : acquiring lock...
C : available Semaphore permits now: 3
```

En este punto, A tiene otorgado un permiso. Quedan 3 disponibles.

```
F : acquiring lock...
A : is performing operation 1, available Semaphore permits : 3
```

A ejecuta la primera iteración del ciclo *for*. Aún no se ha otorgado un permiso a otro hilo.

```
D : available Semaphore permits now: 3
```

```
E : acquiring lock...
B : available Semaphore permits now: 3
E : available Semaphore permits now: 1
D : got the permit!
D : is performing operation 1, available Semaphore permits : 0
F : available Semaphore permits now: 2
C : got the permit!
B : got the permit!
```

En este punto, los hilos D, C y B obtienen los permisos restantes. No olvidar que A tiene el primer permiso. A partir de este momento, A, B, C y D ejecutan sus instrucciones mientras E y F están a la espera.

```
B : is performing operation 1, available Semaphore permits : 0
C : is performing operation 1, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
A : is performing operation 2, available Semaphore permits : 0
D : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
A : is performing operation 3, available Semaphore permits : 0
D : is performing operation 3, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
A : is performing operation 4, available Semaphore permits : 0
C : is performing operation 4, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
D : is performing operation 4, available Semaphore permits : 0
```

```
C : is performing operation 5, available Semaphore permits : 0
A : is performing operation 5, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
D : is performing operation 5, available Semaphore permits : 0
C : releasing lock...
C : available Semaphore permits now: 1
```

C termina de ejecutar sus instrucciones y libera el permiso que tenía.

```
E : got the permit!
```

E obtiene el permiso que ha dejado libre C.

```
E : is performing operation 1, available Semaphore permits : 0
```

E empieza a ejecutar sus instrucciones.

```
A : releasing lock...
```

A termina de ejecutar sus instrucciones y libera un permiso.

```
B : releasing lock...
```

B también termina de ejecutar sus instrucciones.

```
A : available Semaphore permits now: 0
F : got the permit!
```

En este punto las impresiones parecen no tener sentido puesto que según se muestra, A y B liberan sus permisos antes del mensaje que indica que hay 0 hilos disponibles y por lo tanto no tendría lógica que F obtuviera un permiso. Es importante recordar que las operaciones se realizan simultáneamente y puede que una se haya iniciado primero, pero otra termine antes. Podemos interpretar esta parte como que F obtuvo el permiso de A tan rápido que se refleja en el mensaje impreso inmediatamente antes.

```
B : available Semaphore permits now: 1
```

Ahora sí, se refleja el permiso disponible que libera B.

```
F : is performing operation 1, available Semaphore permits : 1
D : releasing lock...
```

F empieza a ejecutar sus instrucciones mientras el hilo D termina las suyas.

```
D : available Semaphore permits now: 2
```

En este punto, los hilos que aun no han terminado de ejecutar sus instrucciones son E y F.

```
E : is performing operation 2, available Semaphore permits : 2
F : is performing operation 2, available Semaphore permits : 2
F : is performing operation 3, available Semaphore permits : 2
E : is performing operation 3, available Semaphore permits : 2
F : is performing operation 4, available Semaphore permits : 2
E : is performing operation 4, available Semaphore permits : 2
F : is performing operation 5, available Semaphore permits : 2
E : is performing operation 5, available Semaphore permits : 2
F : releasing lock...
F : available Semaphore permits now: 3
E : releasing lock...
E : available Semaphore permits now: 4
```

Ambos hilos terminan sus instrucciones y vuelven a estar disponibles 4 permisos.

El programa utiliza un semáforo para limitar la cantidad de hilos que trabajan simultáneamente. Esto se puede aplicar cuando se requiera que más de un proceso utilice el recurso simultáneamente pero no más de los especificados.

Los semáforos, como se mencionó en el problema del productor-consumidor, pueden ser utilizados para manejar situaciones que sigan este patrón. Otra aplicación de los semáforos es coordinar el acceso a un recurso compartido en un sistema distribuido, donde múltiples procesos pueden necesitar acceder al mismo tiempo a un recurso compartido, como puede ser un archivo o una base de datos. En general, los semáforos tienen aplicación en una variedad de situaciones que requieran de control de acceso a recursos compartidos en ambientes con múltiples hilos.

2.2 Sincronización en Java

Compila y ejecuta los ejemplos de Sincronización y realiza un análisis de lo que sucede en cada ejemplo y comenta la importancia de la sincronización de objetos.

Ejemplo 0

Este ejemplo implementa el uso de un método sincronizado. Para ponerlo a prueba, existe una clase, *Table*, cuyo método *printTable(n)* imprime los primeros 5 múltiplos de *n*. Además, se tienen las clases *MyThread1* y *MyThread2*, que en su método *run()* ejecutan el método *printTable()* con diferentes valores de *n*.

Ahora vemos que en el método *main* se crean instancias de *MyThread1* y *MyThread2,* pero su atributo *obj* de la clase *Table* es compartido, es decir, comparten el mismo objeto en sus atributos.

Al ejecutar el programa sin la palabra reservada *synchronized* se obtiene:

```
5
100
200
10
300
15
400
20
500
25
```

Esto sucede porque ambos hilos están usando al objeto *obj* al mismo tiempo para imprimir los números.

Al ejecutar el programa con la palabra reservada *synchronized*:

```
100
200
300
400
500
5
10
15
20
25
```

En este caso se puede notar que las iteraciones del método *printTable()* se ejecutan por completo y es hasta después de haber terminado cuando se permite empezar a la siguiente invocación al método.

Ejemplo 1

En este ejemplo se tienen bloques de código sincronizados. Para los bloques sincronizados se debe especificar el objeto que se sincroniza.

Vemos que el programa se compone de dos hilos: A, el hilo principal, y B. El hilo B suma los primeros 99 números enteros en un ciclo *for* dentro de un bloque sincronizado; luego, notifica (despierta al hilo en espera) una vez que ha terminado.

Por otra parte, en el método *main* del hilo principal, A, se inicia la ejecución de un hilo B. Después de iniciarse el hilo B, se incluye un bloque de código sincronizado con el hilo B, para que el hilo principal, A, se ponga en espera hasta que B termine de ejecutarse, cuando este termina, se despierta de nuevo al hilo A y continua su ejecución.

Al ejecutar el programa se obtiene:

```
Waiting for b to complete...
Total is: 4950
```

Ejemplo 2

En este ejemplo se tienen dos clases además de la principal. La clase *Sender* tiene el método *send(msg)* que “envía” un mensaje (en realidad pone en espera al hilo) e imprime el mensaje enviado.

La clase *ThreadedSend* implementa un hilo que utiliza un bloque sincronizado con el objeto *Sender* para enviar un mensaje. Esto implica que con el mismo *Sender* solo se puede enviar un mensaje a la vez.

En la clase principal, *Syncro*, se crean dos hilos de tipo *ThreadedSend* y se ejecutan. Como se mencionó anteriormente, solo un mensaje puede ser enviado a la vez usando este objeto *Sender*. Por ello, en la ejecución del programa, el segundo hilo se espera a que el primero termine de mandar el mensaje y así poder enviar el suyo.

Al ejecutar el programa se obtiene:

```
Sending Mensaje 1

Mensaje 1 Sent
Sending Mensaje 2

Mensaje 2 Sent
```

Podemos intuir que los bloques sincronizados pueden ser de mucha utilidad cuando se necesita un control mayor en instrucciones con granularidad fina, es decir, cuando queremos tener un control más específico de los métodos sincronizados. También puede ser útil si solo se quiere sincronizar una parte de un método o para usar un objeto diferente para la sincronización.

Es importante notar que solo un permiso es asociado a un objeto, a pesar de que la clase pueda tener varios métodos sincronizados. Esto significa que, si múltiples hilos están intentando ejecutar diferentes métodos sincronizados del mismo objeto, solo un hilo será capaz de hacerlo a la vez.

Tanto los métodos como los bloques sincronizados son herramientas muy útiles para controlar el acceso a recursos compartidos y prevenir condiciones de carrera y otros problemas relacionados con el uso de hilos.

Considero que la sincronización es importante para la programación paralela porque es un mecanismo que permite que múltiples hilos puedan acceder a recursos compartidos, de forma segura y sin interferencia. La sincronización ayuda a prevenir problemas como condiciones de carrera y bloqueos asegurando que solo un hilo pueda manipular el recurso compartido al mismo tiempo. Entonces, la sincronización es clave para la programación paralela, asegurando el correcto funcionamiento de un programa y previniendo errores.

CONCLUSIONES

En esta práctica se utilizaron directivas de *OpenMP* que permiten lo siguiente:
Critical: Demarca una región crítica en la región paralela, implementa exclusión mutua para usar el recurso compartido.
Reduction: Toma el valor de una variable aportada por cada hilo y aplica la operación indicada sobre esos datos para obtener un resultado.
Sections: Permite asignar código independiente a hilos diferentes que trabajen de forma paralela.
Barrier: Coloca una barrera explícita para que cada hilo espere hasta que todos lleguen a la barrera.
Single: Permite definir un bloque de código dentro de una región paralela, que debe ser ejecutado por un único hilo.
Master: Similar a *single* pero las actividades son hechas por el hilo maestro y no se tiene una barrera implícita.

Por otra parte, se revisaron algunos conceptos en Java como métodos y bloques sincronizados. Un método sincronizado es un método usado para controlar el acceso a un recurso compartido en un ambiente multihilo. Cuando un hilo quiere ejecutar un método sincronizado, primero debe obtener un permiso del objeto asociado al método; además, solo un hilo puede ejecutar un método sincronizado al mismo tiempo, lo que significa que otros hilos que quieran ejecutar el mismo método estarán detenidos hasta que el primer hilo termine la ejecución del método y libere el permiso. En contraparte, un bloque sincronizado es usado con el mismo objetivo que un método sincronizado pero su diferencia es que un bloque puede estar asociado a un objeto en particular y puede ser implementado para una porción de código.

Otro método de sincronización que se revisó fue el de semáforos. En general, los semáforos son una herramienta útil para controlar el acceso a recursos compartidos. Pueden permitir que múltiples hilos accedan al recurso compartido, pero limitan el número de hilos que puede acceder al recurso al mismo tiempo.

Dicho todo lo anterior, considero que se ha cumplido con el objetivo planteado para el desarrollo de esta práctica dado que se conocieron y utilizaron algunas directivas de OpemMP para algoritmos paralelos; y, además, se profundizó en el uso de programación concurrente en Java.