	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No. de Práctica(s):	12
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	23 de enero de 2021
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 12: ALGORITMOS PARALELOS

Objetivo: El estudiante conocerá y aprenderá a utilizar algunos 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.

Actividad 1

Se implementó el programa solicitado en la práctica, el cual consiste en obtener el valor máximo en un arreglo de elementos, de forma secuencial y paralela. En el segundo caso, se utilizó el constructo **omp parallel for** para dividir las iteraciones entre los ciclos, y además, se utilizó el constructo **critical** para evitar posibles condiciones de carrera. A continuación, se muestran ambas versiones:

```
void actividad1()
{
    int *a = malloc( _Size: sizeof(int) * N);
    llenaArreglo(a);
    printf( _Format: "Obteniendo maximo...\n");
    printf( _Format: "Version serial:\n");
    printf( _Format: "%d\n", maximo_serial(a, N));
    printf( _Format: "Version paralela:\n");
    printf( _Format: "%d\n", maximo_paralelo(a, N));
}
```

Función principal

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

Función que obtiene el valor máximo de forma serial

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

Función que obtiene el valor máximo de forma serial

```
Actividad 1
1       7       4       0       9       4       8       8       2       4
Obteniendo maximo...
Version serial:
9
Version paralela:
9
```

Salida del programa

Actividad 2

Se implementó el programa solicitado en la práctica, el cual consiste en obtener el producto punto entre dos vectores, de forma secuencial y paralela. En el segundo caso, se utilizó el constructo **omp parallel for** para dividir las iteraciones entre los ciclos, y además, se utilizó un arreglo que mantuviera los resultados temporales par cada hilo. Finalmente, éstos se sumaron para obtener el resultado. La segunda versión del programa se realizó utilizando el constructo **reduction**, el cual realiza de forma automática las tareas mencionadas.

```c
void actividad2()
{
    int *a = malloc( _Size: sizeof(int) * N);
    llenaArreglo(a);
    int *b = malloc( _Size: sizeof(int) * N);
    llenaArreglo(b);

    printf( _Format: "Obteniendo producto punto...\n");
    printf( _Format: "Version serial:\n");
    printf( _Format: "%d\n", producto_punto_serial(a, b, N));
    printf( _Format: "Version paralela 1:\n");
    printf( _Format: "%d\n", producto_punto_paralelo1(a, b, N));
    printf( _Format: "Version paralela 2:\n");
    printf( _Format: "%d\n", producto_punto_paralelo2(a, b, N));
}
```

Función principal

```c
int producto_punto_serial(int *a, int *b, int n)
{
    int result = 0;
    for (int i = 0; i < n; i++)
    {
        result += a[i] * b[i];
    }
    return result;
}
```

Función que obtiene el producto punto entre dos vectores de forma serial

```c
int producto_punto_paralelo1(int *a, int *b, int n)
{
    int result = 0, results[NUM_THREADS];
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        results[tid] = 0;
        #pragma omp for
        for (int i = 0; i < n; i++)
        {
            results[tid] += a[i] * b[i];
        }
    }
    for (int i = 0; i < NUM_THREADS; i++)
    {
        result += results[i];
    }
    return result;
}
```

Función que obtiene el producto punto entre dos vectores de forma paralela

```c
int producto_punto_paralelo2(int *a, int *b, int n)
{
    int result = 0;
    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < n; i++)
    {
        result += a[i] * b[i];
    }
    return result;
}
```

Función que obtiene el producto punto entre dos vectores de forma paralela, utilizando el constructo reduction

```
Actividad 2
5       5       1       7       1       1       5       2       7       6
1       4       2       3       2       2       1       6       8       5
Obteniendo producto punto...
Version serial:
155
Version paralela 1:
155
Version paralela 2:
155
```

Salida del programa

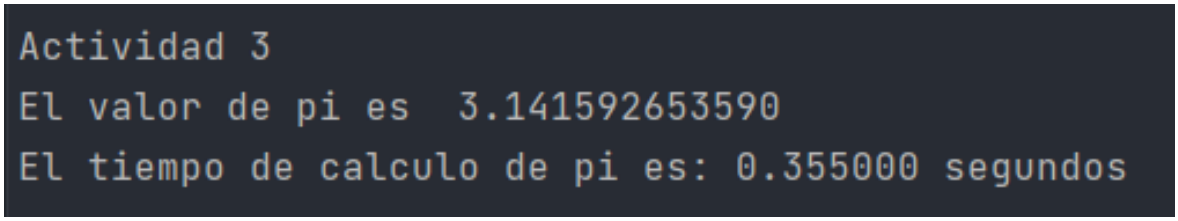
Actividad 3

Se implementó el programa solicitado en la práctica, el cual consiste en paralelizar el código mostrado para obtener una aproximación del número **pi**. Para lograrlo, se añadió el constructo **omp parallel for reduction** en el ciclo mostrado, "reduciendo" el resultado de éste utilizando la operación suma y colocándolo en la variable **sum**.

```
void actividad3()
{
    long long num_steps = 100000000;
    double step = 1.0 / num_steps, empezar = omp_get_wtime(), terminar, pi, sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < num_steps; i++)
    {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1 + x*x);
    }
    pi = sum * step;
    terminar = omp_get_wtime();

    printf( _Format: "El valor de pi es %15.12f\n", pi);
    printf( _Format: "El tiempo de calculo de pi es: %lf segundos\n", terminar - empezar);
}
```

Salida del programa



Salida del programa

Actividad 5

Se implementó el programa mostrado en la guía. A continuación, se muestran las respuestas a las preguntas planteadas:

¿Qué sucede si se quita la barrera? Debido a que el constructo **master** indica un bloque de código que se ejecutará por el hilo maestro, pero sin esperar a que éste finalice su ejecución, el constructo **barrier** asegura que, antes de ejecutar el siguiente bloque de código, todos los hilos finalicen su ejecución hasta ese punto, incluyendo al maestro. Por lo tanto, si se quita la barrera, el siguiente bloque se ejecutará sin importar que el hilo maestro no haya terminado su ejecución. Lo anterior resulta en resultados inesperados, ya que el hilo maestro imprime los elementos del arreglo con elementos inconsistentes, es decir, algunos ya se han actualizado a un nuevo valor, mientras que otros no.

Si en lugar de utilizar el constructor master se utilizara single, ¿qué otros cambios se tienen que hacer en el código? El constructor single permite que el bloque afectado únicamente se ejecute por un solo hilo, y provoca que los demás hilos esperen a que éste termine su ejecución. Por lo tanto, podemos reemplazar el constructo **master** por **single**, y eliminar el innecesario constructo **barrier**.

A continuación, se muestra el programa implementado:

```
void actividad5()
{
    int a[5];
    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < 5; i++)
        {
            a[i] = i * i;
        }
        #pragma omp single
        for (int i = 0; i < 5; ++i)
        {
            printf( _Format: "a[%d] = %d\n", i, a[i]);
        }

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

Función actividad5

```
Actividad 5
a[0] = 0
a[1] = 1
a[2] = 4
a[3] = 9
a[4] = 16
```

Salida del programa

CONCLUSIONES

El paralelismo nos permite distribuir tareas entre distintas unidades de ejecución, para lograr aprovechar los recursos presentes en computadoras con arquitectura de memoria compartida. En la práctica, se implementaron algunos algoritmos paralelos, partiendo de sus versiones secuenciales. Debido a la naturaleza de las computadoras donde éstos se ejecutaron, los tiempos de ejecución mejoraron considerablemente, ya que se lograron aprovechar las distintas unidades de ejecución.

Además, se analizaron algunos constructos de la librería **OpenMP** durante la implementación de los ejercicios, los cuales facilitaron sin duda alguna su realización.