

Laboratorio 05

Realizado por:

Pablo Daniel Barillas Moreno, Carné No. 22193

Competencias para desarrollar

Distribuir la carga de trabajo entre hilos utilizando programación en C y OpenMP.

Instrucciones

Esta actividad se realizará individualmente. Al finalizar los períodos de laboratorio o clase, deberá entregar este archivo en formato PDF y los archivos .c en la actividad correspondiente en Canvas.

1. (18 pts.) Explica con tus propias palabras los siguientes términos:

- a) **Private:** En OpenMP, la cláusula private se utiliza para declarar que cada hilo en una región paralela debe tener su propia copia de una variable. Esto significa que la variable se inicializa como indefinida al inicio de la región paralela y cualquier modificación que haga un hilo no afecta a los demás hilos ni a la variable original fuera de la región paralela. Es útil cuando se necesita que cada hilo trabaje de forma independiente con su propia versión de una variable. (TylerMSFT, 16 de junio del 2023).
- b) **Shared:** La cláusula shared en OpenMP indica que una variable es compartida entre todos los hilos de una región paralela. Esto significa que todos los hilos pueden leer y escribir en la misma variable, lo cual puede llevar a condiciones de carrera si no se maneja correctamente. Para evitar problemas, a menudo se combina con otras construcciones de sincronización (TylerMSFT, 16 de junio del 2023).
- c) **Firstprivate:** La cláusula firstprivate es similar a private, pero además de crear una copia privada de una variable para cada hilo, inicializa cada copia con el valor de la variable original en el momento de la entrada a la región paralela. Esto es útil cuando se necesita que todos los hilos comiencen con el mismo valor inicial, pero luego trabajen independientemente (TylerMSFT, 16 de junio del 2023).
- d) **Barrier:** Una barrier en OpenMP es un punto de sincronización en el que todos los hilos de un equipo deben esperar hasta que todos hayan llegado a ese punto antes de continuar. Esto asegura que ciertas partes del código no se ejecuten hasta que todos los hilos hayan completado las secciones anteriores. Es útil para coordinar el trabajo entre hilos (TylerMSFT, 18 de junio del 2023).

- e) **Critical:** La directiva critical en OpenMP se utiliza para proteger una sección crítica de código, garantizando que solo un hilo pueda ejecutarla a la vez. Esto es necesario cuando se accede a recursos compartidos, como variables globales, para evitar condiciones de carrera. Las secciones críticas pueden tener nombres para diferenciarlas y así permitir que diferentes secciones críticas se ejecuten en paralelo si no interfieren entre sí (TylerMSFT, 18 de junio del 2023).

- f) **Atomic:** La directiva atomic en OpenMP asegura que una operación en una variable compartida se realice de manera atómica, es decir, sin interrupción por otros hilos. Es más eficiente que critical cuando se necesita proteger una sola instrucción de memoria, como un incremento o una suma, ya que evita el overhead de bloquear una región de código completo (TylerMSFT, 18 de junio del 2023).

2. (12 pts.) Escribe un programa en C que calcule la suma de los primeros N números naturales utilizando un ciclo **for paralelo**. Utiliza la cláusula **reduction con +** para acumular la suma en una variable compartida.
- Define N como una constante grande, por ejemplo, $N = 1000000$.
 - Usa `omp_get_wtime()` para medir los tiempos de ejecución.

```
C laboratorio05_Parte2.c > main()
1 // -----
2 // Nombre: Pablo Daniel Barillas Moreno
3 // Universidad: Universidad del Valle de Guatemala
4 // Curso: Programación de microprocesadores
5 // Programa: calculo de la suma de los primeros N números naturales utilizando un ciclo for paralelo - Parte 2 - Laboratorio 05
6 // Versión: 1.0
7 // Fecha: 21/08/2024
8 // Descripción: Este programa en C calcula la suma de los primeros N números naturales utilizando OpenMP para paralelizar
9 // el bucle for. La cláusula 'reduction(+:sum)' se emplea para acumular la suma de forma segura entre múltiples
10 // hilos. El tiempo de ejecución se mide usando 'omp_get_wtime()' y se imprime junto con el resultado de la suma.
11 // -----
12
13 #include <stdio.h> // Incluir la biblioteca estándar de entrada y salida para utilizar printf
14 #include <omp.h> // Incluir la biblioteca de OpenMP para utilizar funciones de paralelización
15
16 #define N 1000000 // Definir N como una constante grande con un valor de 1,000,000
17
18 int main() {
19     int sum = 0; // Declarar e inicializar la variable sum, que almacenará la suma de los primeros N números naturales
20
21     // Medir el tiempo de inicio usando omp_get_wtime(), que devuelve el tiempo actual en segundos
22     double start_time = omp_get_wtime();
23
24     // Paralelizar el ciclo for utilizando la cláusula reduction para acumular la suma
25     // La cláusula 'reduction(+:sum)' asegura que cada hilo acumule su parte de la suma de forma segura
26     #pragma omp parallel for reduction(+:sum)
27     for (int i = 1; i <= N; i++) {
28         sum += i; // En cada iteración, se suma el valor de i a la variable sum
29     }
30
31     // Medir el tiempo de finalización usando omp_get_wtime() para calcular el tiempo total de ejecución
32     double end_time = omp_get_wtime();
33
34     // Imprimir el resultado
35     printf("\n-----Suma de los primeros números naturales-----\n");
36     // Imprimir el resultado de la suma de los primeros N números naturales
37     printf("\nLa suma de los primeros %d números naturales es: %d\n", N, sum);
38     // Imprimir el tiempo total de ejecución del ciclo for paralelo
39     printf("\nTiempo de ejecución: %f segundos\n\n", end_time - start_time);
40
41     return 0; // Finalizar el programa con un valor de retorno 0, indicando que todo salió correctamente
42 }
```

```
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> gcc -fopenmp -o laboratorio05_Parte2 laboratorio05_Parte2.c
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> ./laboratorio05_Parte2
```

```
-----Suma de los primeros n||meros naturales-----
La suma de los primeros 1000000 n||meros naturales es: 1784293664
Tiempo de ejecuci||n: 0.005000 segundos
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> |
```

3. (15 pts.) Escribe un programa en C que ejecute tres funciones diferentes en paralelo usando la **directiva #pragma omp sections**. Cada sección debe ejecutar una función distinta, por ejemplo, una que calcule el factorial de un número, otra que genere la serie de Fibonacci, y otra que encuentre el máximo en un arreglo, operaciones matemáticas no simples. Asegúrate de que cada función sea independiente y no tenga dependencias con las otras.

```
Laboratorio-05_Daniel-Barillas > C laboratorio05_Parte3.c > main()
1 // -----
2 // Nombre: Pablo Daniel Barillas Moreno
3 // Universidad: Universidad del Valle de Guatemala
4 // Curso: Programación de microprocesadores
5 // Programa: Operaciones Matemáticas Paralelas - Parte 3 - Laboratorio 05
6 // Versión: 1.0
7 // Fecha: 21/08/2024
8 // Descripción: Este programa en C utiliza OpenMP para ejecutar tres funciones matemáticas diferentes
9 //              en paralelo. Las funciones incluyen: el cálculo del factorial de un número, la
10 //              generación de la serie de Fibonacci, y la búsqueda del valor máximo en un arreglo.
11 //              Cada función se ejecuta en su propia sección paralela, demostrando el uso eficiente de
12 //              la directiva #pragma omp sections para dividir el trabajo entre múltiples hilos.
13 // -----
14
15 #include <stdio.h> // Incluir la biblioteca estándar de entrada y salida para usar printf
16 #include <omp.h> // Incluir la biblioteca OpenMP para la paralelización
17
18 // -----
19 // Función para calcular el factorial de un número
20 // -----
21 unsigned long long factorial(int n) {
22     unsigned long long result = 1; // Inicializar la variable result en 1 para el cálculo del factorial
23     for (int i = 2; i <= n; i++) { // Bucle desde 2 hasta n para multiplicar todos los enteros positivos
24         result *= i; // Multiplicar result por el valor de i en cada iteración para calcular el factorial
25     }
26     return result; // Retornar el resultado del factorial
27 }
28
29 // -----
30 // Función para generar la serie de Fibonacci hasta n
31 // -----
32 void fibonacci(int n) {
33     unsigned long long a = 0, b = 1, temp; // Inicializar las variables para la serie de Fibonacci
34     printf("\nSerie de Fibonacci: "); // Imprimir el encabezado para la serie de Fibonacci
35     for (int i = 1; i <= n; i++) { // Bucle desde 1 hasta n para generar la serie
36         printf("%llu ", a); // Imprimir el número actual en la serie de Fibonacci
37         temp = a + b; // Calcular el siguiente número en la serie
38         a = b; // Actualizar a al valor de b
39         b = temp; // Actualizar b al nuevo valor calculado
40     }
41     printf("\n"); // Imprimir un salto de línea después de la serie
42 }
43
```

```

44 // -----
45 // Función para encontrar el máximo en un arreglo
46 // -----
47 int find_max(int arr[], int size) {
48     int max = arr[0]; // Inicializar max con el primer elemento del arreglo
49     for (int i = 1; i < size; i++) { // Bucle desde el segundo elemento hasta el final del arreglo
50         if (arr[i] > max) { // Comparar el elemento actual con el valor máximo encontrado hasta ahora
51             max = arr[i]; // Si el elemento actual es mayor, actualizar max
52         }
53     }
54     return max; // Retornar el valor máximo encontrado en el arreglo
55 }
56
57 int main() {
58     int n = 10; // Definir el valor de n, que se utilizará en las funciones de factorial y Fibonacci
59     int arr[] = {3, 5, 7, 2, 8, -1, 4, 10, 12}; // Definir el arreglo de enteros para la función de máximo
60     int size = sizeof(arr) / sizeof(arr[0]); // Calcular el tamaño del arreglo
61
62     // -----
63     // Ejecutar las funciones en paralelo usando OpenMP sections
64     // -----
65     #pragma omp parallel sections // Iniciar una región paralela dividiendo el trabajo en secciones
66     {
67         #pragma omp section // Iniciar la primera sección paralela
68         {
69             unsigned long long fact = factorial(n); // Llamar a la función factorial y almacenar el resultado
70             printf("\n-----\n");
71             printf("Factorial de %d es: %llu\n", n, fact); // Imprimir el resultado del factorial
72             printf("-----\n");
73         }
74
75         #pragma omp section // Iniciar la segunda sección paralela
76         {
77             printf("\n-----\n");
78             fibonacci(n); // Llamar a la función fibonacci para generar e imprimir la serie
79             printf("-----\n");
80         }
81
82         #pragma omp section // Iniciar la tercera sección paralela
83         {
84             int max = find_max(arr, size); // Llamar a la función find_max y almacenar el valor máximo
85             printf("\n-----\n");
86             printf("El máximo valor en el arreglo es: %d\n", max); // Imprimir el valor máximo encontrado
87             printf("-----\n");
88         }
89     }
90
91     return 0; // Finalizar el programa con un valor de retorno 0, indicando que todo salió correctamente
92 }

```

```

PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> cd "c:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores\Laboratorio-05_Daniel-Barillas" ; if ($?) { gcc laboratorio05_Parte3.c -o laboratorio05_Parte3 } ; if ($?) { .\laboratorio05_Parte3 }

```

Factorial de 10 es: 3628800

Serie de Fibonacci: 0 1 1 2 3 5 8 13 21 34

El máximo valor en el arreglo es: 12

```

PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores\Laboratorio-05_Daniel-Barillas>

```

4. **(15 pts.)** Escribe un programa en C que tenga un ciclo for donde se modifiquen dos variables de manera paralela usando `#pragma omp parallel for`.
 - a. Usa la cláusula `shared` para gestionar el acceso a la variable1 dentro del ciclo.
 - b. Usa la cláusula `private` para gestionar el acceso a la variable2 dentro del ciclo.
 - c. Prueba con ambas cláusulas y explica las diferencias observadas en los resultados.

5. **(30 pts.)** Analiza el código en el programa `Ejercicio_5A.c`, que contiene un programa secuencial. Indica cuántas veces aparece un valor `key` en el vector `a`. Escribe una versión paralela en OpenMP utilizando una descomposición de tareas **recursiva**, en la cual se generen tantas tareas como hilos.

6. **REFLEXIÓN DE LABORATORIO:** se habilitará en una actividad independiente.

Referencias:

1. TylerMSFT. (16 de junio del 2023). *Cláusulas de OpenMP*. Microsoft.com.
<https://learn.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-clauses?view=msvc-170#private-openmp>
2. TylerMSFT. (18 de junio del 2023). *Directivas de OpenMP*. Microsoft.com.
<https://learn.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170#barrier>