Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3086 Programación de microprocesadores

Ciclo 2 de 2,024
Temario A

# 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.

a) Define N como una constante grande, por ejemplo, N = 1000000.

b) Usa omp_get_wtime() para medir los tiempos de ejecución.

laboratorio05_Parte2.c > main()

```c
// ------------------------------------------------------------------------------------------------
// Nombre: Pablo Daniel Barillas Moreno
// Universidad: Universidad del Valle de Guatemala
// Curso: Programación de microprocesadores
// Programa: calculo de la suma de los primeros N números naturales utilizando un ciclo for paralelo - Parte 2 - Laboratorio 05
// Versión: 1.0
// Fecha: 21/08/2024
// Descripción: Este programa en C calcula la suma de los primeros N números naturales utilizando OpenMP para paralelizar
//              el bucle for. La cláusula `reduction(+:sum)` se emplea para acumular la suma de forma segura entre múltiples
//              hilos. El tiempo de ejecución se mide usando `omp_get_wtime()` y se imprime junto con el resultado de la suma.
// ------------------------------------------------------------------------------------------------

#include <stdio.h>  // Incluir la biblioteca estándar de entrada y salida para utilizar printf
#include <omp.h>    // Incluir la biblioteca de OpenMP para utilizar funciones de paralelización

#define N 1000000   // Definir N como una constante grande con un valor de 1,000,000

int main() {
    int sum = 0;    // Declarar e inicializar la variable sum, que almacenará la suma de los primeros N números naturales

    // Medir el tiempo de inicio usando omp_get_wtime(), que devuelve el tiempo actual en segundos
    double start_time = omp_get_wtime();

    // Paralelizar el ciclo for utilizando la cláusula reduction para acumular la suma
    // La cláusula 'reduction(+:sum)' asegura que cada hilo acumule su parte de la suma de forma segura
    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= N; i++) {
        sum += i;    // En cada iteración, se suma el valor de i a la variable sum
    }

    // Medir el tiempo de finalización usando omp_get_wtime() para calcular el tiempo total de ejecución
    double end_time = omp_get_wtime();

    // Imprimir el resultado
    printf("\n-----Suma de los primeros números naturales-----\n");
    // Imprimir el resultado de la suma de los primeros N números naturales
    printf("\nLa suma de los primeros %d números naturales es: %d\n", N, sum);
    // Imprimir el tiempo total de ejecución del ciclo for paralelo
    printf("\nTiempo de ejecución: %f segundos\n\n", end_time - start_time);

    return 0; // Finalizar el programa con un valor de retorno 0, indicando que todo salió correctamente
}
```

```
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()

```c
// ------------------------------------------------------------------------------------------
// Nombre: Pablo Daniel Barillas Moreno
// Universidad: Universidad del Valle de Guatemala
// Curso: Programación de microprocesadores
// Programa: Operaciones Matemáticas Paralelas - Parte 3 - Laboratorio 05
// Versión: 1.0
// Fecha: 21/08/2024
// Descripción: Este programa en C utiliza OpenMP para ejecutar tres funciones matemáticas diferentes
//              en paralelo. Las funciones incluyen: el cálculo del factorial de un número, la
//              generación de la serie de Fibonacci, y la búsqueda del valor máximo en un arreglo.
//              Cada función se ejecuta en su propia sección paralela, demostrando el uso eficiente de
//              la directiva #pragma omp sections para dividir el trabajo entre múltiples hilos.
// ------------------------------------------------------------------------------------------

#include <stdio.h>      // Incluir la biblioteca estándar de entrada y salida para usar printf
#include <omp.h>        // Incluir la biblioteca OpenMP para la paralelización

// ------------------------------------------------------------------------------------------
// Función para calcular el factorial de un número
// ------------------------------------------------------------------------------------------
unsigned long long factorial(int n) {
    unsigned long long result = 1;  // Inicializar la variable result en 1 para el cálculo del factorial
    for (int i = 2; i <= n; i++) {  // Bucle desde 2 hasta n para multiplicar todos los enteros positivos
        result *= i;  // Multiplicar result por el valor de i en cada iteración para calcular el factorial
    }
    return result;  // Retornar el resultado del factorial
}

// ------------------------------------------------------------------------------------------
// Función para generar la serie de Fibonacci hasta n
// ------------------------------------------------------------------------------------------
void fibonacci(int n) {
    unsigned long long a = 0, b = 1, temp;  // Inicializar las variables para la serie de Fibonacci
    printf("\nSerie de Fibonacci: ");  // Imprimir el encabezado para la serie de Fibonacci
    for (int i = 1; i <= n; i++) {  // Bucle desde 1 hasta n para generar la serie
        printf("%llu ", a);  // Imprimir el número actual en la serie de Fibonacci
        temp = a + b;  // Calcular el siguiente número en la serie
        a = b;  // Actualizar a al valor de b
        b = temp;  // Actualizar b al nuevo valor calculado
    }
    printf("\n");  // Imprimir un salto de línea después de la serie
}

```

```c
// ------------------------------------------------------------------------------------------------
// Función para encontrar el máximo en un arreglo
// ------------------------------------------------------------------------------------------------
int find_max(int arr[], int size) {
    int max = arr[0];  // Inicializar max con el primer elemento del arreglo
    for (int i = 1; i < size; i++) {  // Bucle desde el segundo elemento hasta el final del arreglo
        if (arr[i] > max) {  // Comparar el elemento actual con el valor máximo encontrado hasta ahora
            max = arr[i];  // Si el elemento actual es mayor, actualizar max
        }
    }
    return max;  // Retornar el valor máximo encontrado en el arreglo
}

int main() {
    int n = 10;  // Definir el valor de n, que se utilizará en las funciones de factorial y Fibonacci
    int arr[] = {3, 5, 7, 2, 8, -1, 4, 10, 12};  // Definir el arreglo de enteros para la función de máximo
    int size = sizeof(arr) / sizeof(arr[0]);  // Calcular el tamaño del arreglo

    // ------------------------------------------------------------------------------------------------
    // Ejecutar las funciones en paralelo usando OpenMP sections
    // ------------------------------------------------------------------------------------------------
    #pragma omp parallel sections  // Iniciar una región paralela dividiendo el trabajo en secciones
    {
        #pragma omp section  // Iniciar la primera sección paralela
        {
            unsigned long long fact = factorial(n);  // Llamar a la función factorial y almacenar el resultado
            printf("\n-------------------------------------------\n");
            printf("Factorial de %d es: %llu\n", n, fact);  // Imprimir el resultado del factorial
            printf("-------------------------------------------\n");
        }

        #pragma omp section  // Iniciar la segunda sección paralela
        {
            printf("\n-------------------------------------------\n");
            fibonacci(n);  // Llamar a la función fibonacci para generar e imprimir la serie
            printf("-------------------------------------------\n");
        }

        #pragma omp section  // Iniciar la tercera sección paralela
        {
            int max = find_max(arr, size);  // Llamar a la función find_max y almacenar el valor máximo
            printf("\n-------------------------------------------\n");
            printf("El máximo valor en el arreglo es: %d\n", max);  // Imprimir el valor máximo encontrado
            printf("-------------------------------------------\n");
        }
    }

    return 0;  // Finalizar el programa con un valor de retorno 0, indicando que todo salió correctamente
}
```

```
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.

Laboratorio-05_Daniel-Barillas > C laboratorio05_Parte4.c > ...

```c
// ------------------------------------------------------------------------------------------
// Nombre: Pablo Daniel Barillas Moreno
// Universidad: Universidad del Valle de Guatemala
// Curso: Programación de microprocesadores
// Programa: Manipulación de Variables en Paralelo con OpenMP - Laboratorio 05
// Versión: 1.0
// Fecha: 21/08/2024
// Descripción: Este programa en C utiliza OpenMP para modificar y manejar dos variables dentro
//              de un ciclo `for` paralelo. Se demuestra el uso de las cláusulas `shared` y `private`
//              para gestionar el acceso a las variables dentro del ciclo.
//              - `variable1` es compartida entre todos los hilos y su valor final refleja las
//                modificaciones realizadas por todos los hilos.
//              - `variable2` es privada para cada hilo, lo que significa que cada hilo tiene
//                su propia copia independiente, y su valor no es consistente fuera del ciclo paralelo.
//              Este programa ilustra cómo manejar correctamente variables compartidas y privadas
//              en un entorno de paralelización para evitar condiciones de carrera y asegurar
//              la integridad de los datos.
// ------------------------------------------------------------------------------------------
#include <stdio.h>     // Biblioteca estándar para entrada/salida
#include <omp.h>       // Biblioteca para paralelización con OpenMP

int main() {
    // Declaración e inicialización de variables
    int variable1 = 0;  // variable1 es compartida entre todos los hilos
    int variable2 = 0;  // variable2 será privada para cada hilo

    // Paralelizar el ciclo for usando OpenMP
    // 'shared(variable1)' indica que variable1 es compartida entre los hilos
    // 'private(variable2)' indica que cada hilo tendrá su propia copia de variable2
    #pragma omp parallel for shared(variable1) private(variable2)
    for (int i = 0; i < 10; i++) {  // Bucle iterando 10 veces
        variable1 += i;  // variable1 es modificada por todos los hilos (compartida)
        variable2 += i;  // variable2 es modificada independientemente por cada hilo (privada)

        // Imprimir el número de hilo, la iteración y los valores actuales de las variables
        printf("\nThread %d | Iteration %d | variable1 (shared): %d | variable2 (private): %d\n",
               omp_get_thread_num(), i, variable1, variable2);
    }

    // Imprimir el valor final de variable1 después de que todos los hilos hayan completado el bucle
    printf("\n-----------------------------------------------------------\n");
    printf("Valor final de variable1 (compartida): %d\n", variable1);
    printf("-----------------------------------------------------------\n");

```

```
45      // Nota: variable2 no se imprime aquí porque es privada para cada hilo
46      // y su valor no es consistente fuera del ciclo paralelo
47
48      return 0;  // Finalizar el programa y retornar 0, indicando ejecución exitosa
49  }
```

```
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores\Laboratorio-05_Daniel-Barillas> gcc -fopenmp -o laboratorio05_Parte4  laboratorio05_Parte4.c
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores\Laboratorio-05_Daniel-Barillas> ./laboratorio05_Parte4

Thread 1 | Iteration 2 | variable1 (shared): 2 | variable2 (private): 1873653818

Thread 1 | Iteration 3 | variable1 (shared): 44 | variable2 (private): 1873653821

Thread 2 | Iteration 4 | variable1 (shared): 12 | variable2 (private): 1873654108

Thread 5 | Iteration 7 | variable1 (shared): 19 | variable2 (private): 1873655263

Thread 3 | Iteration 5 | variable1 (shared): 24 | variable2 (private): 1873654829

Thread 0 | Iteration 0 | variable1 (shared): 24 | variable2 (private): 0

Thread 0 | Iteration 1 | variable1 (shared): 45 | variable2 (private): 1

Thread 7 | Iteration 9 | variable1 (shared): 41 | variable2 (private): 1873655697

Thread 4 | Iteration 6 | variable1 (shared): 8 | variable2 (private): 1873654542

Thread 6 | Iteration 8 | variable1 (shared): 32 | variable2 (private): 1873655552

------------------------------------------------------------
Valor final de variable1 (compartida): 45
------------------------------------------------------------
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores\Laboratorio-05_Daniel-Barillas>
```

**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.

## ***Análisis del código en el programa Ejercicio_5A.c***

Este código es un programa secuencial que cuenta cuántas veces aparece un valor específico (key) en un arreglo de números aleatorios. Aquí te explico cómo funciona el programa y cuál es el resultado esperado:

**Descripción del Programa:**

1. **Generación del Arreglo:**
    - Se define un arreglo a de tamaño N (131072).
    - El arreglo se llena con números aleatorios generados por la función rand() que son menores a N.
2. **Inserción Manual del Valor key:**
    - Se inserta manualmente el valor key (que es 42) en tres posiciones específicas del arreglo:
        - a[N % 43] = key;
        - a[N % 73] = key;
        - a[N % 3] = key;
    - Estas operaciones garantizan que key esté presente al menos en tres posiciones del arreglo.
3. **Cuenta de key:**
    - La función count_key() recorre el arreglo desde la primera posición hasta la última, contando cuántas veces aparece key.
    - El valor nkey guarda el número total de veces que key aparece en el arreglo.
4. **Impresión del Resultado:**
    - Finalmente, el programa imprime cuántas veces se encontró key en el arreglo.

**Resultados Esperados:**

- **Número de veces que aparece key:**
    - El valor key aparece al menos 3 veces en el arreglo (debido a las inserciones manuales).
    - Dependiendo de los valores generados aleatoriamente, key podría aparecer más veces en el arreglo.
    - El valor exacto impreso por nkey será la suma de las inserciones manuales y cualquier otra aparición de key generada aleatoriamente.

**Explicación Detallada:**

- **Inserciones Manuales de key:**
    - a[N % 43] = key;: Inserta key en la posición que resulta de N % 43.
    - a[N % 73] = key;: Inserta key en la posición que resulta de N % 73.
    - a[N % 3] = key;: Inserta key en la posición que resulta de N % 3.

Dado que N es 131072, las posiciones específicas son:

- N % 43 = 4, N % 73 = 32, y N % 3 = 2.

- **Función count_key():**
    - Esta función recorre el arreglo y, por cada coincidencia con key, incrementa el contador count.

## ***Código sin reescribir***

Ejercicio_5A.c > count_key(long, long *, long)

```c
/*
 * Archivo: Ejercicio_5A.c
 * Descripción: Este programa cuenta cuántas veces aparece un valor específico
 * ('key') en un arreglo de números aleatorios.
 *
 * Funcionalidad:
 * - Genera un arreglo de tamaño N con valores aleatorios.
 * - Inserta manualmente el valor 'key' en tres posiciones específicas del arreglo.
 * - Cuenta cuántas veces aparece 'key' en el arreglo usando una función llamada count_key().
 * - Imprime el número de apariciones de 'key' en el arreglo.
 *
 * Referencia:
 * Chandra, . R. et al. Parallel Programming in OpenMP
 *
 * Fecha modificación: 08-16-2024
 */

#include <stdio.h>
#include <stdlib.h>

#define N 131072

long count_key(long Nlen, long *a, long key) {
    long count = 0;
    for (int i = 0; i < Nlen; i++)
        if (a[i] == key) count++;
    return count;
}

int main() {
    long a[N], key = 42, nkey = 0;
    for (long i = 0; i < N; i++) a[i] = rand() % N;
    a[N % 43] = key;
    a[N % 73] = key;
    a[N % 3] = key;

    nkey = count_key(N, a, key);  // cuenta key secuencialmente
    printf("Numero de veces que 'key' aparece secuencialmente: %ld\n", nkey);

    return 0;
}

```

Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3086 Programación de microprocesadores

Ciclo 2 de 2,024
Temario A

***Código reescrito en paralela en OpenMP utilizando una descomposición de tareas recursiva, en la cual se generen tantas tareas como hilos***

Ejercicio_5A_Corregido.c > main()

```c
// ------------------------------------------------------------------------------------------
// Nombre: Pablo Daniel Barillas Moreno
// Universidad: Universidad del Valle de Guatemala
// Curso: Programación de microprocesadores
// Programa: Conteo Paralelo de Apariciones de un Valor en un Arreglo - Laboratorio 05
// Versión: 1.0
// Fecha: 21/08/2024
// Descripción: Este programa en C cuenta cuántas veces aparece un valor específico ('key') en un
//              arreglo de números aleatorios utilizando OpenMP para paralelizar la operación.
//              Se emplea una descomposición de tareas recursiva que genera tantas tareas como
//              hilos disponibles, dividiendo el arreglo en subarreglos más pequeños y combinando
//              los resultados. Esto demuestra un uso eficiente de OpenMP para tareas de conteo
//              paralelizadas, mejorando el rendimiento en sistemas con múltiples núcleos.
// ------------------------------------------------------------------------------------------

#include <stdio.h>      // Incluir la biblioteca estándar de entrada/salida para usar printf
#include <stdlib.h>     // Incluir la biblioteca estándar para usar funciones como rand()
#include <omp.h>        // Incluir la biblioteca OpenMP para paralelización

#define N 131072        // Definir el tamaño del arreglo como una constante (131072)


// Función recursiva para contar las apariciones de 'key' en un subarreglo
long parallel_count_key(long *a, long key, long start, long end) {
    long count = 0;  // Inicializar contador para las apariciones de 'key'

    // Si el subarreglo es pequeño, realizar la cuenta secuencialmente
    if (end - start < N / omp_get_max_threads()) {
        // Bucle para recorrer el subarreglo y contar las apariciones de 'key'
        for (long i = start; i < end; i++) {
            if (a[i] == key) count++;  // Incrementar el contador si se encuentra 'key'
        }
        return count;  // Retornar el número de apariciones encontradas en este subarreglo
    } else {
        // Si el subarreglo es grande, dividirlo en dos mitades
        long mid = start + (end - start) / 2;
        long left_count = 0, right_count = 0;  // Inicializar contadores para ambas mitades

        // Generar tarea recursiva para contar en la primera mitad del subarreglo
        #pragma omp task shared(left_count)
        left_count = parallel_count_key(a, key, start, mid);

        // Generar tarea recursiva para contar en la segunda mitad del subarreglo
        #pragma omp task shared(right_count)
        right_count = parallel_count_key(a, key, mid, end);

```

```c
        // Esperar a que ambas tareas terminen
        #pragma omp taskwait

        // Combinar los resultados de las tareas y retornar el total
        return left_count + right_count;
    }
}

int main() {
    long a[N], key = 42, nkey = 0;  // Declarar el arreglo, la clave a buscar, y el contador de apariciones

    // Llenar el arreglo con valores aleatorios
    for (long i = 0; i < N; i++) a[i] = rand() % N;

    // Insertar manualmente 'key' en tres posiciones específicas del arreglo
    a[N % 43] = key;
    a[N % 73] = key;
    a[N % 3] = key;

    // Contar las apariciones de 'key' en paralelo usando OpenMP y la función recursiva
    #pragma omp parallel  // Iniciar una región paralela
    {
        #pragma omp single  // Asegurar que solo un hilo inicie la tarea recursiva inicial
        nkey = parallel_count_key(a, key, 0, N);
    }

    // Imprimir el número de veces que 'key' aparece en el arreglo
    printf("\nNumero de veces que 'key' aparece en paralelo: %ld\n\n", nkey);

    return 0;  // Finalizar el programa y devolver 0, indicando que todo salió correctamente
}
```

```
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> gcc -fopenmp -o Ejercicio_5A_Corregido  Ejercicio_5A_Corregido.c
PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores> ./Ejercicio_5A_Corregido

Numero de veces que 'key' aparece en paralelo: 8

PS C:\Users\Daniel Barillas\Desktop\Lab 05_Microprocesadores>
```

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

***Se encontrará la reflexión en su respectivo espacio.***

# 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