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

Javier Alexander Linares Chang – 231135

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

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

   a) **private:** Esta cláusula hace que cada hilo tenga su propia copia de una variable. Para que asi cada hilo trabaje de forma independiente con su copia, sin interferir con las copias de otros hilos.

   b) **shared:** Con esta cláusula todos los hilos comparten la misma variable. Esto puede ser útil en ciertos casos, pero también arriesgado porque varios hilos pueden intentar modificar la variable al mismo tiempo.

   c) **firstprivate:** Es muy similar a private, pero con la diferencia de que cada hilo comienza con una copia de la variable inicializada al valor que tenía antes de entrar en la región paralela.

   d) **barrier:** Una barrera es un punto donde todos los hilos se detienen hasta que todos hayan llegado. Esto asegura que ningún hilo avance hasta que todos los hilos hayan terminado las tareas previas.

   e) **critical:** Uso esta directiva para proteger secciones de código que no deben ser ejecutadas por más de un hilo al mismo tiempo. Así evito problemas de concurrencia.

   f) **atomic:** Similar a critical, pero más eficiente para operaciones simples como sumar o restar. Asegura que solo un hilo realiza la operación en un momento dado, evitando conflictos.

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.

   Resultado:

```
PS C:\Users\linar\OneDrive\Escritorio> gcc -fopenmp -o Lab5_2 Lab5_2.c
PS C:\Users\linar\OneDrive\Escritorio> ./Lab5_2
La suma de los primeros 1000000 numeros naturales es: 500000500000
Tiempo de ejecucion: 0.002000 segundos
```

Código realizado:

```c
#include <stdio.h>
#include <omp.h>  // Incluimos la librería de OpenMP para programación paralela

#define N 1000000  // Definimos N como una constante grande, el número de primeros números naturales que vamos a sumar

int main() {
    long long sum = 0;  // Variable para almacenar la suma de los números naturales
    double start_time, end_time;  // Variables para medir el tiempo de ejecución

    // Medimos el tiempo de inicio de la operación
    start_time = omp_get_wtime();

    // Paralelizamos el bucle con reducción para sumar los números
    // La cláusula reduction(+:sum) asegura que la variable 'sum' se acumule de manera segura entre todos los hilos
    #pragma omp parallel for reduction(+:sum)
    for (int i = 1; i <= N; i++) {
        sum += i;  // Acumulamos el valor de 'i' en 'sum'
    }

    // Medimos el tiempo de finalización de la operación
    end_time = omp_get_wtime();

    // Mostramos el resultado de la suma y el tiempo de ejecución
    printf("La suma de los primeros %d numeros naturales es: %lld\n", N, sum);
    printf("Tiempo de ejecucion: %f segundos\n", end_time - start_time);

    return 0;  // Finalizamos el programa
}
```

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.

```c
#include <stdio.h>
#include <omp.h>

// Función para calcular el factorial de un número
void calcularFactorial(int num) {
    long long factorial = 1;
    for (int i = 1; i <= num; i++) {
        factorial *= i;
    }
    printf("El factorial de %d es: %lld\n", num, factorial);
}

// Función para generar la serie de Fibonacci hasta un número dado
void generarFibonacci(int num) {
    int fib[num];  // Array para almacenar los números de Fibonacci
    fib[0] = 0;
    fib[1] = 1;
    printf("Serie de Fibonacci hasta %d: %d %d", num, fib[0], fib[1]);

    for (int i = 2; i < num; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
        printf(" %d", fib[i]);
    }
    printf("\n");
}
```

```c
// Función para encontrar el máximo en un arreglo
void encontrarMaximo(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    printf("El maximo en el arreglo es: %d\n", max);
}

int main() {
    // Parámetros de las funciones
    int numFactorial = 10;  // Número para calcular el factorial
    int numFibonacci = 10;  // Número de términos para la serie de Fibonacci
    int arr[] = {1, 5, 3, 9, 2, 8, 4};  // Arreglo para encontrar el máximo
    int size = sizeof(arr) / sizeof(arr[0]);  // Tamaño del arreglo

    // Ejecutamos las tres funciones en paralelo usando secciones
    #pragma omp parallel sections
    {
        // Sección 1: Calcular el factorial
        #pragma omp section
        {
            calcularFactorial(numFactorial);
        }

        // Sección 2: Generar la serie de Fibonacci
        #pragma omp section
        {
            generarFibonacci(numFibonacci);
        }

        // Sección 3: Encontrar el máximo en un arreglo
        #pragma omp section
        {
            encontrarMaximo(arr, size);
        }
    }

    return 0;
}
```

```
El factorial de 10 es: 3628800
Serie de Fibonacci hasta 10: 0 1 1 2 3 5 8 13 21 34El maximo en el arreglo es: 9
```

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.

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int N = 10;  // Definimos el tamaño del ciclo
    int variable1 = 0;  // Variable compartida entre los hilos (shared)
    int variable2 = 0;  // Variable privada para cada hilo (private)

    // Ciclo for paralelo con cláusulas shared y private
    #pragma omp parallel for shared(variable1) private(variable2)
    for (int i = 0; i < N; i++) {
        // Incremento de variable1 (compartida)
        variable1 += i;

        // Inicializamos variable2 en cada hilo de forma independiente
        variable2 = i * 2;

        // Imprimimos dentro del ciclo para ver los resultados
        printf("Numero de Hilo %d - Iteracion %d: variable1 = %d, variable2 = %d\n", omp_get_thread_num(), i, variable1, variable2);
    }

    // Resultados finales
    printf("\nResultado final:\n");
    printf("variable1 (shared) = %d\n", variable1);

    return 0;
}
```

```
PS C:\Users\linar\OneDrive\Escritorio> gcc -fopenmp -o Lab5_4 Lab5_4.c
PS C:\Users\linar\OneDrive\Escritorio> ./Lab5_4
Numero de Hilo 0 - Iteracion 0: variable1 = 6, variable2 = 0
Numero de Hilo 0 - Iteracion 1: variable1 = 18, variable2 = 2
Numero de Hilo 0 - Iteracion 2: variable1 = 20, variable2 = 4
Numero de Hilo 3 - Iteracion 8: variable1 = 17, variable2 = 16
Numero de Hilo 3 - Iteracion 9: variable1 = 29, variable2 = 18
Numero de Hilo 1 - Iteracion 3: variable1 = 9, variable2 = 6
Numero de Hilo 1 - Iteracion 4: variable1 = 33, variable2 = 8
Numero de Hilo 1 - Iteracion 5: variable1 = 38, variable2 = 10
Numero de Hilo 2 - Iteracion 6: variable1 = 6, variable2 = 12
Numero de Hilo 2 - Iteracion 7: variable1 = 45, variable2 = 14

Resultado final:
variable1 (shared) = 45
PS C:\Users\linar\OneDrive\Escritorio>
```

Análisis del código y resultados:

Cláusula shared: variable1 es compartida entre todos los hilos debido a la cláusula shared. Esto significa que todos los hilos acceden y modifican la misma instancia de variable1.

Dado que múltiples hilos están intentando modificar variable1 al mismo tiempo, puede ocurrir una condición de carrera (race condition). Esto sucede porque las operaciones de incremento variable1 += i; no son atómicas, y el resultado depende del orden y el momento en que cada hilo accede y modifica la variable.

Cláusula private: variable2 es privada para cada hilo debido a la cláusula private.

Cada hilo tiene su propia instancia de variable2, que se inicializa y utiliza independientemente de los demás hilos, Por lo tanto, no hay interferencia entre hilos cuando modifican variable2.

Diferencias observadas en los resultados:
Los resultados impresos muestran que variable1 (compartida) tiene valores inconsistentes y parecen estar modificados de manera no ordenada. Esto se debe a que cada hilo accede a variable1 de forma concurrente y no se asegura la exclusividad durante la operación de incremento.

Por otra parte, los valores de variable2 (privada) son coherentes con las iteraciones del ciclo y el número del hilo. Cada hilo inicializa y utiliza su propia copia de variable2, por lo que no hay interferencia o condición de carrera.

5. **(30 pts.)** Analizar 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

Definición de Constantes y Declaraciones:

- El programa define una constante N para establecer el tamaño del arreglo.
- Declara un arreglo a de tamaño N para almacenar números enteros generados aleatoriamente.
- Declara una variable key que contiene el valor a buscar en el arreglo y nkey para almacenar el número de ocurrencias de key.

Generación de Datos:

- Llena el arreglo a con números aleatorios generados por la función rand().

Inserción del Valor key:

- Inserta el valor key en tres posiciones específicas del arreglo para asegurar que key aparece al menos tres veces.

Conteo de Ocurrencias:

- Utiliza una función count_key para contar cuántas veces aparece el valor key en el arreglo.
- La función count_key recorre el arreglo y compara cada elemento con key, incrementando un contador cada vez que encuentra una coincidencia.

Impresión del Resultado:

- El programa imprime el número de veces que key aparece en el arreglo.

El resultado muestra el número de veces que el valor key aparece en el arreglo, considerando tanto las inserciones manuales como cualquier otra ocurrencia que pueda resultar de la generación aleatoria de datos.

```
PS C:\Users\linar\OneDrive\Escritorio> gcc -fopenmp -o Ejercicio_5A Ejercicio_5A.c
PS C:\Users\linar\OneDrive\Escritorio> ./Ejercicio_5A
Numero de veces que 'key' aparece secuencialmente: 8
```