

Hoja de Trabajo #1

Fecha de Entrega: 11 de Agosto, 2023.

Descripción: En esta hoja de trabajo empezará a familiarizarse con las directivas, cláusulas y funciones de OpenMP haciendo uso de ellas en pequeños ejercicios que demostrarán su funcionamiento. Explorará el uso de OpenMP con 2 pequeños ejercicios introductorios para familiarizarse con el API y luego creará un programa que resuelve y ejecuta un método numérico y lo transformará de secuencial a paralelo.

Entregables: Deberá entregar un documento con las respuestas a las preguntas planteadas en cada ejercicio (incluyendo diagramas o screenshots si es necesario), junto con todos los archivos de código que programe debidamente comentados e identificados.

Materiales: necesitará una máquina virtual con Linux.

Contenido:

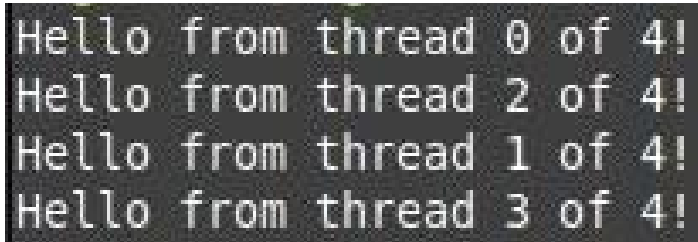
Ejercicio 1 (10 puntos)

Verifique y asegúrese que el compilador GCC está disponible en su sistema. Luego, escriba y compile un programa en C llamado "hello_omp.c" que, haciendo uso del api de OpenMP, imprima N veces el mensaje "Hello from thread <número de thread> of <cantidad de threads> !".

El mensaje debe incluir el *número de thread* del thread realizando el *printf* y la *cantidad de threads* que su programa ejecutará. La cantidad de threads debe ser ingresada desde línea de comando al ejecutar el programa. También, valide que si el número de threads no fue ingresado desde línea de comando, su programa automáticamente utilizará 10 threads como valor *default*.

NO utilice ciclos *for* en su implementación.

Ejemplo del resultado esperado:



```
Hello from thread 0 of 4!  
Hello from thread 2 of 4!  
Hello from thread 1 of 4!  
Hello from thread 3 of 4!
```

Recuerde agregar screenshots de la ejecución de sus programas.

PREGUNTA: ¿Por qué al ejecutar su código los mensajes no están desplegados en orden?

Codigo

```
// Ejercicio#1- Paralela
// Esteban Aldana Guerra - 20591

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

int main(int argc, char *argv[])
{
    int N = 10; // valor por defecto

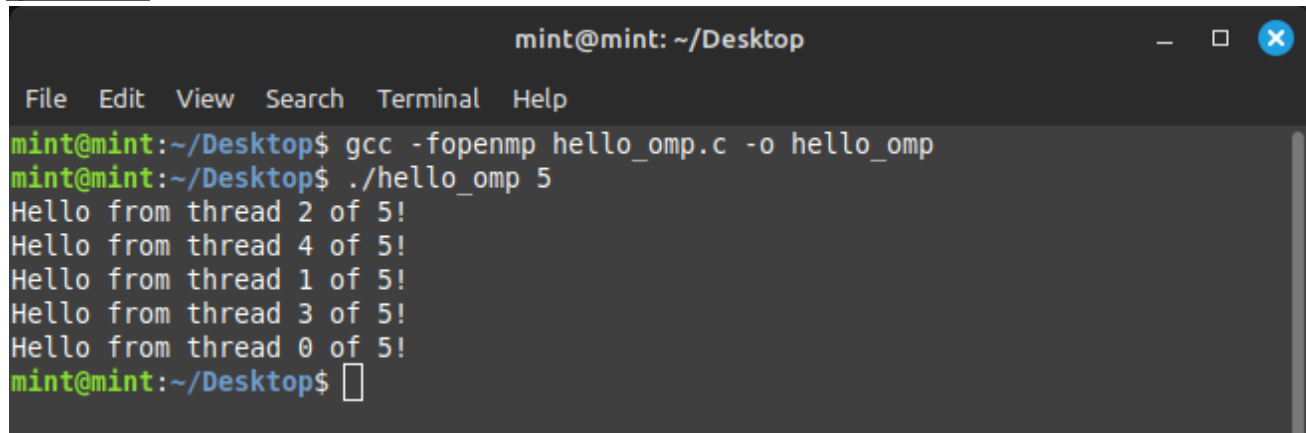
    if (argc > 1)
    {
        N = atoi(argv[1]); // convierte el argumento ingresado en un entero
    }

    omp_set_num_threads(N); // establece el número de threads a usar

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        printf("Hello from thread %d of %d!\n", thread_id, total_threads);
    }

    return 0;
}
```

Ejerciciosón



```
mint@mint: ~/Desktop
File Edit View Search Terminal Help
mint@mint:~/Desktop$ gcc -fopenmp hello_omp.c -o hello_omp
mint@mint:~/Desktop$ ./hello_omp 5
Hello from thread 2 of 5!
Hello from thread 4 of 5!
Hello from thread 1 of 5!
Hello from thread 3 of 5!
Hello from thread 0 of 5!
mint@mint:~/Desktop$
```

PREGUNTA: ¿Por qué al ejecutar su código los mensajes no están desplegados en orden?

Los mensajes no están desplegados en orden porque OpenMP crea múltiples threads que se ejecutan concurrentemente. Cada thread se ejecuta en su propio ritmo, y no hay orden en el que los threads accederán a la función printf, por lo que terminan no estando en orden.

Ejercicio 2 (10 puntos)

Copie el código del ejercicio # 1 y luego modifíquelo para que haga lo siguiente. Cree un programa en C llamado "hbd_omp.c" que imprima el ID de cada thread y la cantidad de threads. Imprima los siguientes mensajes dependiendo de si el ID del thread es par o impar:

- ID Impar: "Feliz cumpleaños número <cantidad de threads>!".
- ID Par: "Saludos del hilo <id del thread>"

Al momento de ejecutar su programa envíe como parámetro de cantidad de threads su edad. Compílelo y ejecútelo con el comando:

```
./<nombre_ejecutable> <SU_EDAD>
```

Ejemplo del resultado esperado:

```
Feliz cumpleaños número 10!  
Saludos del hilo 0  
Feliz cumpleaños número 10!  
Saludos del hilo 2  
Saludos del hilo 4  
Feliz cumpleaños número 10!  
Saludos del hilo 6  
Feliz cumpleaños número 10!  
Saludos del hilo 8  
Feliz cumpleaños número 10!
```

Código implementado:

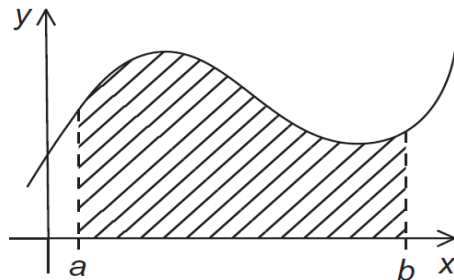
```
C hbd_omp.c > ...  
1 // Ejercicio#2- Paralela  
2 // Esteban Aldana Guerra - 20591  
3 #include <stdio.h>  
4 #include <stdlib.h>  
5 #include <omp.h>  
6  
7 int main(int argc, char *argv[])  
8 {  
9     if (argc < 2)  
10    {  
11        printf("Por favor, especifique la cantidad de threads (su edad) como argumento.\n");  
12        return 1;  
13    }  
14  
15    int threads = atoi(argv[1]);  
16    omp_set_num_threads(threads); // Seteamos la cantidad de threads.  
17  
18    #pragma omp parallel  
19    {  
20        int id = omp_get_thread_num(); // Obtenemos el ID del thread.  
21        int total_threads = omp_get_num_threads(); // Obtenemos el total de threads.  
22  
23        if (id % 2 == 0)  
24        {  
25            printf("Saludos del hilo %d\n", id);  
26        }  
27        else  
28        {  
29            printf("Feliz cumpleaños número %d!\n", total_threads);  
30        }  
31    }  
32  
33    return 0;  
34 }  
35
```

Respuesta al correrlo

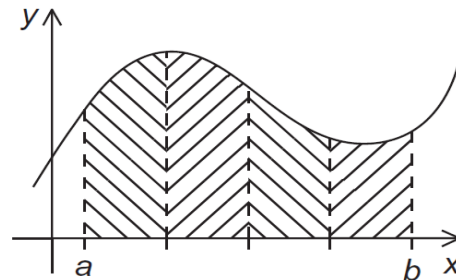
```
mint@mint: ~/Desktop
File Edit View Search Terminal Help
mint@mint:~/Desktop$ gcc -fopenmp hbd_omp.c -o hbd_omp
mint@mint:~/Desktop$ ./hbd_omp 21
Feliz cumpleaños número 21!
Saludos del hilo 0
Saludos del hilo 20
Feliz cumpleaños número 21!
Feliz cumpleaños número 21!
Feliz cumpleaños número 21!
Feliz cumpleaños número 21!
Saludos del hilo 6
Saludos del hilo 10
Saludos del hilo 8
Feliz cumpleaños número 21!
Feliz cumpleaños número 21!
Saludos del hilo 2
Feliz cumpleaños número 21!
Saludos del hilo 12
Saludos del hilo 4
Saludos del hilo 16
Feliz cumpleaños número 21!
Saludos del hilo 18
Saludos del hilo 14
Feliz cumpleaños número 21!
mint@mint:~/Desktop$
```

Ejercicio 3 (40 puntos)

La “Regla Trapezoidal” o “Sumas de Riemann” es un método numérico que nos permite realizar aproximaciones sobre integrales que no tienen una solución directa con los métodos tradicionales de integrales (sustitución, por partes, etc...). Lo que el método nos dice es que podemos estimar el área bajo una curva dividiéndola en trapezoides de tamaño finito.



(a)



(b)

Para resolver este problema se define un intervalo (a, b) de los puntos sobre los cuales queremos encontrar la estimación del área bajo la curva. Luego lo dividiremos en n subintervalos iguales que conformaran nuestros trapezoides. Mientras mayor sea n mejor será la estimación. Entonces, para estimar el área bajo la curva de una función $ff(x)$ debemos sumar el área del trapezoide T_0 hasta T_n .

Dicha suma se puede expresar con la siguiente fórmula

$$\int_a^b ff(x) \approx \frac{h}{2} [ff(XX_0) + ff(XX_1) + ff(XX_2) + ff(XX_3) + \dots + ff(XX_{n-1}) + ff(XX_n)]$$

Donde h representa la base de cada trapezoide y se ve de la siguiente manera:

$$h = \frac{b - a}{n}$$

Y donde los inputs de nuestra función $ff(x)$ se ven de la siguiente manera:

$$\begin{aligned} XX_0 &= a \\ XX_1 &= a + h \\ XX_2 &= a + h + h = a + 2h \\ XX_{n-1} &= a + (n - 1)h \\ XX_n &= b \end{aligned}$$

Cree un archivo llamado "riemann.c" en donde codificará su programa. Investigue acerca de las "Sumas de Riemann" y sobre la "Regla trapezoidal" de ser necesario para mejorar su entendimiento sobre el programa que codificará. Su programa deberá contener la función "trapezoides" para calcular de forma secuencial una suma de riemann para una función $ff(x)$ previamente definida. Los intervalos (a, b) para el cálculo deben ser ingresados desde línea de comando.

```
./<nombre_ejecutable> a b
```

Ejecute su programa, utilizando $n = 10e6$ para las siguientes funciones y con los siguientes intervalos:

- x^2 (2, 10)
- $2x^3$ (3, 7)
- $\sin(x)$ (0, 1)

Imprima el resultado de su programa de la siguiente manera:

```
Con n = 10000000 trapezoides, nuestra aproximacion
de la integral de 3.000000 a 7.000000 es = 1159.999999998
```

Recuerde agregar screenshots de la ejecución de sus programas.

Codigo

```
C riemann.c -
6 #include <math.h>
7
8 // Función para integrar x^2
9 double f1(double x)
10 {
11     return x * x;
12 }
13
14 // Función para integrar 2x^3
15 double f2(double x)
16 {
17     return 2 * x * x * x;
18 }
19
20 // Función para integrar sin(x)
21 double f3(double x)
22 {
23     return sin(x);
24 }
25
26 // Función que implementa la regla del trapecioide
27 double trapezoides(double (*func)(double), double a, double b, int n)
28 {
29     double h = (b - a) / n;
30     double result = 0.5 * (func(a) + func(b));
31     for (int i = 1; i < n; i++)
32     {
33         result += func(a + i * h);
34     }
35     result *= h;
36     return result;
37 }
38
39 int main(int argc, char **argv)
40 {
41     if (argc != 3)
42     {
43         printf("Usage: %s <a> <b>\n", argv[0]);
44         return 1;
45     }
46
47     double a = atof(argv[1]);
48     double b = atof(argv[2]);
49     int n = 10000000; // 10e6
50
51     if (a == 2.0 && b == 10.0)
52     {
53         printf("Con n = %d trapezoides, nuestra aproximacion de la integral de %lf a %lf de x^2 es %lf\n",
54             n, a, b, trapezoides(f1, a, b, n));
55     }
56     else if (a == 3.0 && b == 7.0)
57     {
58         printf("Con n = %d trapezoides, nuestra aproximacion de la integral de %lf a %lf de 2x^3 es %lf\n",
59             n, a, b, trapezoides(f2, a, b, n));
60     }
61     else if (a == 0.0 && b == 1.0)
62     {
63         printf("Con n = %d trapezoides, nuestra aproximacion de la integral de %lf a %lf de sin(x) es %lf\n",
64             n, a, b, trapezoides(f3, a, b, n));
65     }
66     else
67     {
68         printf("Intervalo no reconocido.\n");
69     }
70
71     return 0;
72 }
73
```

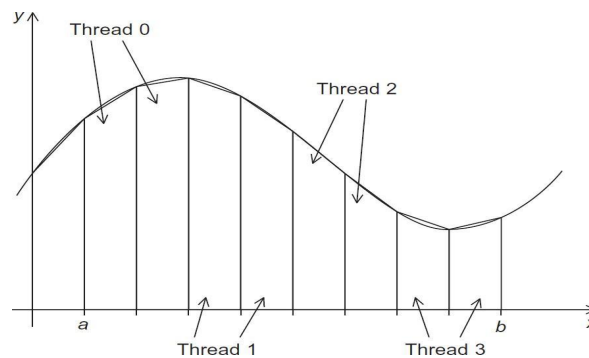
Correr el codigo

```
mint@mint: ~/Desktop
File Edit View Search Terminal Help
mint@mint:~/Desktop$ gcc riemann.c -o riemann -lm
mint@mint:~/Desktop$ ./riemann 2 10
Con n = 10000000 trapezoides, nuestra aproximacion de la integral de 2.000000 a 10.000000 de x^2 es 330.666667
mint@mint:~/Desktop$ ./riemann 3 7
Con n = 10000000 trapezoides, nuestra aproximacion de la integral de 3.000000 a 7.000000 de 2x^3 es 1160.000000
mint@mint:~/Desktop$ ./riemann 0 1
Con n = 10000000 trapezoides, nuestra aproximacion de la integral de 0.000000 a 1.000000 de sin(x) es 0.459698
mint@mint:~/Desktop$
```

Ejercicio 4 (20 puntos)

Ahora que su programa "riemann.c" calcula serialmente una aproximación de la integral de una función $f(x)$ necesitamos paralelizar este proceso. Para ello cree una copia de su programa y renómbrelo como "remann_omp2.c".

Para hacer esta conversión asumimos que manejamos muchos más trapezoides que threads. Vamos a dividir el total de trapezoides (data) dentro del número de threads, es decir, **división del dominio**. Asegúrese que el número de trapezoides sea múltiplo del número de threads.



Necesitará hacer uso de lo que llamaremos "parámetros locales" y para asegurarnos de que cada thread realice el trabajo correcto, debemos crear explícitamente los parámetros locales de cada uno:

- número local de trapezoides = $n_{\text{local}} = \text{trapezoides} / \text{threads}$
- valor inicial local del rango = $a_{\text{local}} = a + (\text{ID_thread} * n_{\text{local}} * \text{ancho_h})$
- valor final local del rango = $b_{\text{local}} = a_{\text{local}} + (n_{\text{local}} * \text{ancho_h})$

Ejemplo:

- $a = 0, b = 100, n = 500, \text{threads} = 4$
- $h = (100 - 0) / 500 = 0.2$
- $n_{\text{local}} = 500 / 4 = 125$

ID	$\text{span} = n_{\text{local}} * h$ $\text{offset} = \text{ID} * \text{span}$	$a_{\text{local}} = a + \text{offset}$	$b_{\text{local}} = a_{\text{local}} + \text{span}$
0	$0 * (125 * 0.2) = 0$	$0 + 0 = 0$	$0 + (125 * 0.2) = 25$
1	$1 * (125 * 0.2) = 25$	$0 + 25 = 25$	$25 + (125 * 0.2) = 50$
2	$2 * (125 * 0.2) = 50$	$0 + 50 = 50$	$50 + (125 * 0.2) = 75$
3	$3 * (125 * 0.2) = 75$	$0 + 75 = 75$	$75 + (125 * 0.2) = 100$

Realice los ajustes a su código para recibir como parámetro adicional la cantidad de threads a utilizar.

```
./<nombre_ejecutable> a b <cantidad de threads>
```

Además, cuando un thread ejecute la función “trapezoides” este debe calcular una parte de la integral e irlo añadiendo a una variable global (utilice la directiva **#pragma omp critical** en este punto). Despliegue el resultado en pantalla.

NO paralelice los ciclos *for* que pudiese tener adentro de su función.

PREGUNTA: ¿Por qué es necesario el uso de la directiva **#pragma omp critical?**

La directiva **#pragma omp critical** se utiliza para asegurar que una sección específica del código solo sea ejecutada por un thread a la vez. Es esencial cuando varios threads acceden o modifican recursos compartidos, como variables globales. Si no esto puede llevar a ocasionar race conditions.

Codigo

```
// Función para integrar 2x^3
double f2(double x)
{
    return 2 * x * x * x;
}

// Función para integrar sin(x)
double f3(double x)
{
    return sin(x);
}

// Función que implementa la regla del trapecioide
double trapezoides(double (*func)(double), double a, double b, int n, int threads)
{
    double h = (b - a) / n;
    double result = 0.5 * (func(a) + func(b));

    int n_local = n / threads;

#pragma omp parallel num_threads(threads)
    {
        int id = omp_get_thread_num();
        double a_local = a + id * n_local * h;
        double b_local = a_local + n_local * h;

        double local_result = 0.0;
        for (int i = 1; i <= n_local; i++)
        {
            local_result += func(a_local + i * h);
        }

#pragma omp critical
        {
            result += local_result;
        }
    }

    result *= h;
    return result;
}

int main(int argc, char *argv[])
{
    if (argc != 4)
    {
        printf("Usage: ./executable a b num_threads\n");
        return 1;
    }

    double a = atof(argv[1]);
    double b = atof(argv[2]);
    int threads = atoi(argv[3]);

    if (a == 2 && b == 10)
    {
        printf("con n = 10000000 trapezoides, usando %d threads, nuestra aproximacion ", threads);
        printf("de la integral de %lf a %lf de x^2 es %lf\n", a, b, trapezoides(f2, a, b, 10000000, threads));
    }
    else if (a == 3 && b == 7)
    {
        printf("con n = 10000000 trapezoides, usando %d threads, nuestra aproximacion ", threads);
        printf("de la integral de %lf a %lf de 2x^3 es %lf\n", a, b, trapezoides(f2, a, b, 10000000, threads));
    }
    else if (a == 0 && b == 1)
    {
        printf("con n = 10000000 trapezoides, usando %d threads, nuestra aproximacion ", threads);
        printf("de la integral de %lf a %lf de sin(x) es %lf\n", a, b, trapezoides(f3, a, b, 10000000, threads));
    }
    else
    {
        printf("Función no especificada para el intervalo dado.\n");
    }

    return 0;
}
```

Resultado

```
mint@mint: ~/Desktop
File Edit View Search Terminal Help
mint@mint:~/Desktop$ gcc riemann_omp2.c -o riemann_omp2 -lm -fopenmp
mint@mint:~/Desktop$ ./riemann_omp2 2 10 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 2.000000 a 10.000000 de x^2 es 330
.666747
mint@mint:~/Desktop$ ./riemann_omp2 3 7 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 3.000000 a 7.000000 de 2x^3 es 116
0.000274
mint@mint:~/Desktop$ ./riemann_omp2 0 1 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 0.000000 a 1.000000 de sin(x) es 0
.459698
mint@mint:~/Desktop$
```

```

mint@mint: ~/Desktop
File Edit View Search Terminal Help
mint@mint:~/Desktop$ gcc riemann_omp_nocrit.c -o riemann_omp_nocrit -lm -fopenmp
mint@mint:~/Desktop$ ./riemann_omp_nocrit 2 10 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 2.000000 a 10.000000 de x^2 es 330
.666747
mint@mint:~/Desktop$ ./riemann_omp_nocrit 3 7 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 3.000000 a 7.000000 de 2x^3 es 116
0.000274
mint@mint:~/Desktop$ ./riemann_omp_nocrit 0 1 4
con n = 10000000 trapezoides, usando 4 threads, nuestra aproximacion de la integral de 0.000000 a 1.000000 de sin(x) es 0
.459698
mint@mint:~/Desktop$

```

Recuerde agregar screenshots de la ejecución de sus programas.