CC3069 Computación Paralela y Distribuida
Sección 11
Erick Stiv Junior Guerra Muñoz - 21781

Hoja de Trabajo 1 - Computación Paralela y Distribuida

Repositorio de GitHub: [stiv2202/HDT1_Paralela]

Ejercicio 1

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.

Imagen 1.1: Programa creado

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>

bool isNumber(const char *str) {
    while (*str) {
        if (!isdigit(*str)) {
            return false;
        }
        str++;
    }
    return true;
}

int main(int argc, char *argv[]) {
    int nthreads = 10;

    if (argc > 1) {
        if (isNumber(argv[1])) {
            nthreads = atoi(argv[1]);
        } else {
            printf("Error: El parámetro ingresado no es un número válido. Utilizando valor por defecto.\n");
            nthreads = 10;
        }
    }

    omp_set_num_threads(nthreads);

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

    return 0;
}
```

Imagen 1.2: Salida del programa indicando número de threads

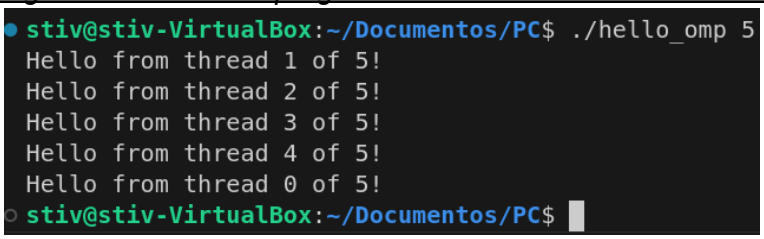
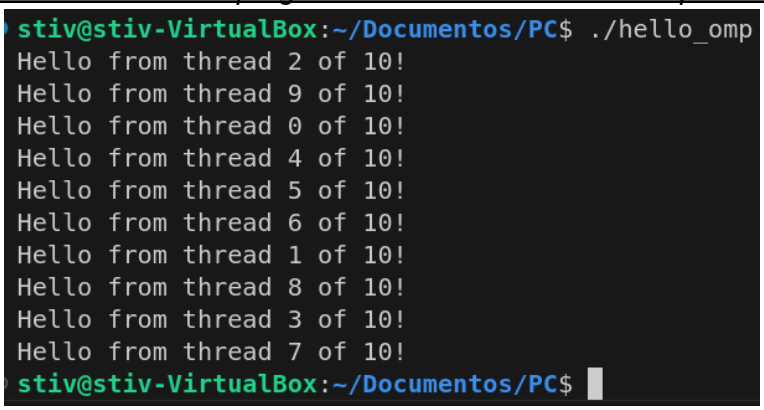


Imagen 1.3: Salida del programa con número de threads por defecto



- ¿Por qué al ejecutar su código los mensajes no están desplegados en orden?
  R: Puesto que cada uno de los mensajes se está ejecutando en un hilo distinto por OpenMP, cada hilo se ejecutará de forma concurrente, por lo que terminarán la ejecución del código en tiempos ligeramente diferentes en función de la arquitectura del procesador que se está utilizando.

Ejercicio 2

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>
```

Imagen 2.1: Programa creado

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>

bool isNumber(const char *str) {
    while (*str) {
        if (!isdigit(*str)) {
            return false;
        }
        str++;
    }
    return true;
}

int main(int argc, char *argv[]) {
    int age = 10;

    if (argc > 1) {
        if (isNumber(argv[1])) {
            age = atoi(argv[1]);
        } else {
            printf("Error: El parámetro ingresado no es un número válido. Utilizando valor por defecto.\n");
            age = 10;
        }
    }

    omp_set_num_threads(age);

    #pragma omp parallel
    {
        int thread_num = omp_get_thread_num();
        int total_threads = omp_get_num_threads();

        if (thread_num % 2 == 0)
            printf("Saludos del hilo %d.\n", thread_num);
        else
            printf("Feliz cumpleaños número %d!\n", total_threads);
    }

    return 0;
}
```

Imagen 2.2: Salida del programa indicando número de threads

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./hbd_omp 5
Feliz cumpleaños número 5!
Feliz cumpleaños número 5!
Saludos del hilo 4.
Saludos del hilo 2.
Saludos del hilo 0.
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Imagen 1.2: Salida del programa con número de threads por defecto

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./hbd_omp
Feliz cumpleaños número 10!
Saludos del hilo 2.
Feliz cumpleaños número 10!
Saludos del hilo 0.
Saludos del hilo 6.
Feliz cumpleaños número 10!
Feliz cumpleaños número 10!
Saludos del hilo 8.
Feliz cumpleaños número 10!
Saludos del hilo 4.
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Ejercicio 3

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 $f(x)$ 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 = 1159.9999999998
```

Imagen 3.1: Programa creado

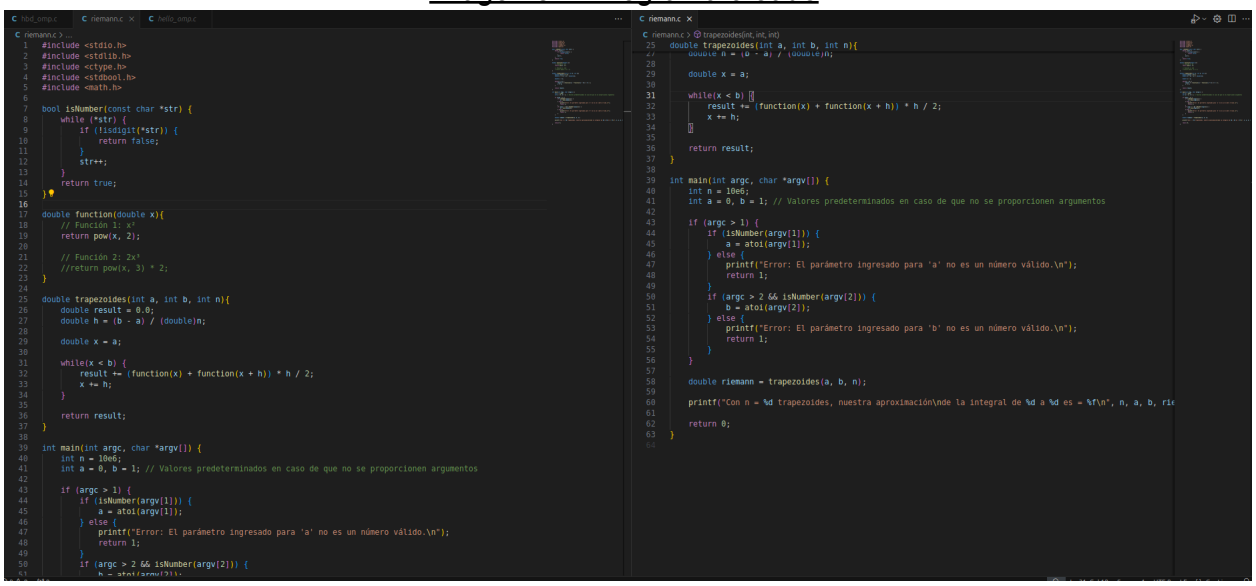


Imagen 3.2: Salida del programa (función 1)

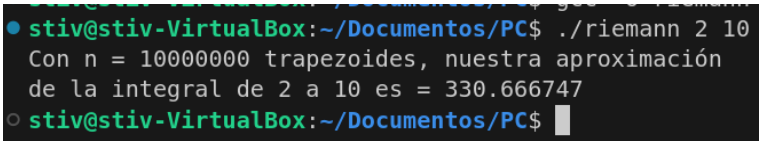


Imagen 3.3: Salida del programa (función 2)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann 3 7
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 3 a 7 es = 1160.000000
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Imagen 3.4: Salida del programa (función 3)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann 0 1
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 0 a 1 es = 0.459698
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Ejercicio 4

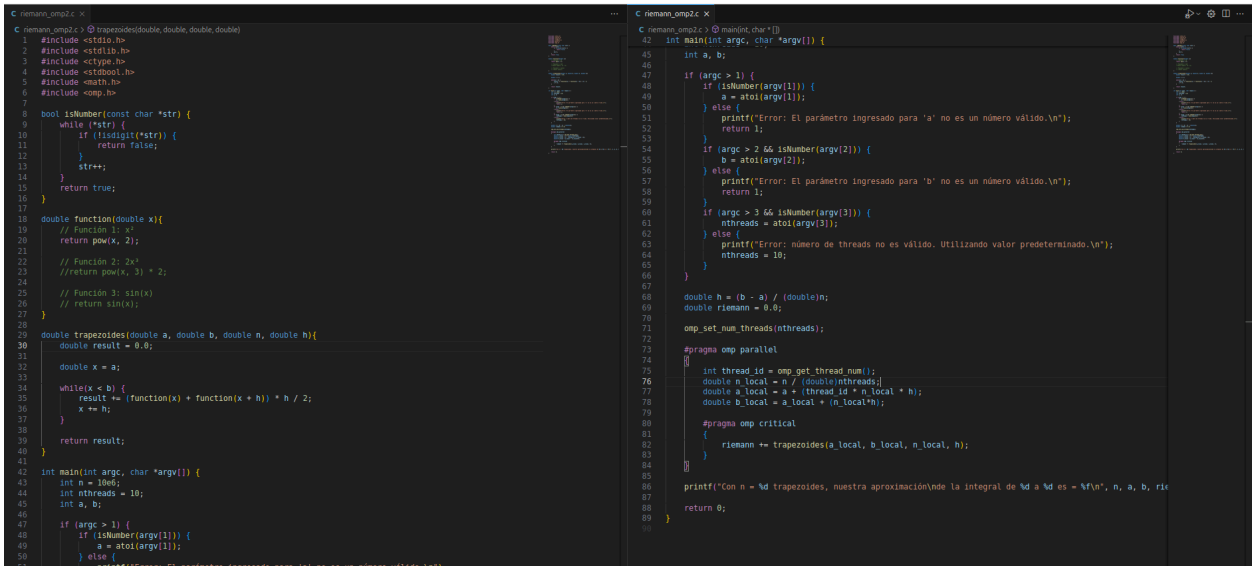
Ahora que su programa “riemann.c” calcula serialmente una aproximación de la integral de una función $f(x)$ necesitamos paralelizar este proceso. Para ello cree una copia de su programa y renómbrelo como *“riemann_omp2.c”*.

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.

Imagen 4.1: Programa creado



```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <math.h>
#include <omp.h>

bool isNumber(const char *str) {
    while (*str) {
        if (!isdigit(*str)) {
            return false;
        }
        str++;
    }
    return true;
}

double function(double x){
    // Función 1: x²
    return pow(x, 2);

    // Función 2: 2x³
    //return pow(x, 3) * 2;

    // Función 3: sin(x)
    // return sin(x);
}

double trapezoides(double a, double b, double n, double h){
    double result = 0.0;

    double x = a;

    while(x < b) {
        result += (function(x) + function(x + h)) * h / 2;
        x += h;
    }

    return result;
}

int main(int argc, char *argv[]) {
    int n = 10e6;
    int nthreads = 10;
    int a, b;

    if (argc > 1) {
        if (isNumber(argv[1])) {
            a = atoi(argv[1]);
        } else {
```

```
int main(int argc, char *argv[]) {
    int a, b;

    if (argc > 1) {
        if (isNumber(argv[1])) {
            a = atoi(argv[1]);
        } else {
            printf("Error: El parámetro ingresado para 'a' no es un número válido.\n");
            return 1;
        }
        if (argc > 2 && isNumber(argv[2])) {
            b = atoi(argv[2]);
        } else {
            printf("Error: El parámetro ingresado para 'b' no es un número válido.\n");
            return 1;
        }
        if (argc > 3 && isNumber(argv[3])) {
            nthreads = atoi(argv[3]);
        } else {
            printf("Error: número de threads no es válido. Utilizando valor predeterminado.\n");
            nthreads = 10;
        }
    }

    double h = (b - a) / (double)n;
    double riemann = 0.0;

    omp_set_num_threads(nthreads);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        double n_local = n / (double)nthreads;
        double a_local = a + (thread_id * n_local * h);
        double b_local = a_local + (n_local*h);

        #pragma omp critical
        {
            riemann += trapezoides(a_local, b_local, n_local, h);
        }
    }

    printf("Con n = %d trapezoides, nuestra aproximación\nde la integral de %d a %d es = %f\n", n, a, b, ri

    return 0;
}
```

Imagen 4.2: Salida del programa (función 1)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann_omp2 2 10 100
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 2 a 10 es = 330.668179
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Imagen 4.3: Salida del programa (función 2)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann_omp2 3 7 100
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 3 a 7 es = 1160.000890
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Imagen 4.4: Salida del programa (función 3)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann_omp2 0 1 100
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 0 a 1 es = 0.459701
stiv@stiv-VirtualBox:~/Documentos/PC$
```

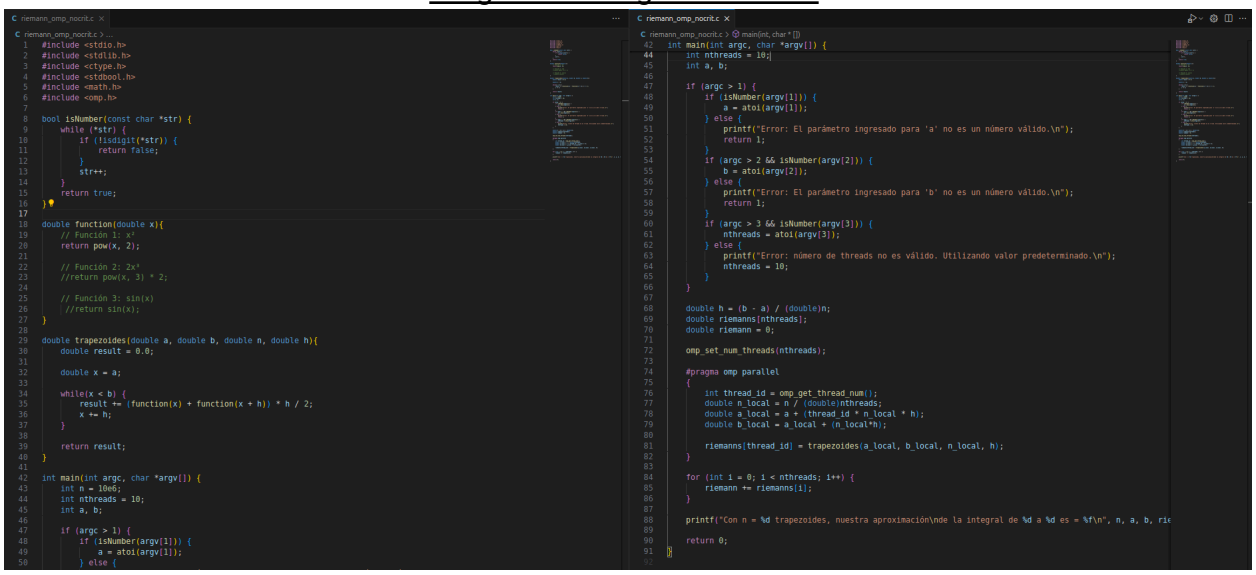
- ¿Por qué es necesario el uso de la directiva `#pragma omp critical`?
  R: La función de esta directiva es proteger secciones del código para que no sean accedidas simultáneamente por más de un thread. En este caso, es necesaria puesto que de esta forma se asegura que se modificará la variable global "riemann" únicamente un thread a la vez, evitando condiciones de carrera.

Ejercicio 5

Ahora que ya tiene un programa que paraleliza la resolución de calcular una integral a través de la *"Regla Trapezoidal"* / *"Sumas de Riemann"* lo que haremos es evitar el uso de la directiva `#pragma omp critical`. Para ello cree una nueva versión del código que programó en el ejercicio 4 y renómbrelo a *"riemann_omp_nocrit.c"*.

En esta nueva iteración del programa, deberá agregar un arreglo global en donde almacene el cálculo que realizó localmente cada uno de los threads. Finalmente, sume los resultados del arreglo para obtener el resultado final y despliéguelo en pantalla.

Imagen 5.1: Programa creado



```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <math.h>
#include <omp.h>

bool isNumber(const char *str) {
    while (*str) {
        if (!isdigit(*str)) {
            return false;
        }
        str++;
    }
    return true;
}

double function(double x){
    // Función 1: x²
    return pow(x, 2);

    // Función 2: 2x³
    //return pow(x, 3) * 2;

    // Función 3: sin(x)
    //return sin(x);
}

double trapezoides(double a, double b, double n, double h){
    double result = 0.0;

    double x = a;

    while(x < b) {
        result += (function(x) + function(x + h)) * h / 2;
        x += h;
    }

    return result;
}

int main(int argc, char *argv[]) {
    int n = 10e6;
    int nthreads = 10;
    int a, b;

    if (argc > 1) {
        if (isNumber(argv[1])) {
            a = atoi(argv[1]);
        } else {
            printf("Error: El parámetro ingresado para 'a' no es un número válido.\n");
            return 1;
        }
        if (argc > 2 && isNumber(argv[2])) {
            b = atoi(argv[2]);
        } else {
            printf("Error: El parámetro ingresado para 'b' no es un número válido.\n");
            return 1;
        }
        if (argc > 3 && isNumber(argv[3])) {
            nthreads = atoi(argv[3]);
        } else {
            printf("Error: número de threads no es válido. Utilizando valor predeterminado.\n");
            nthreads = 10;
        }
    }

    double h = (b - a) / (double)n;
    double riemanns[nthreads];
    double riemann = 0;

    omp_set_num_threads(nthreads);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        double n_local = n / (double)nthreads;
        double a_local = a + (thread_id * n_local * h);
        double b_local = a_local + (n_local*h);

        riemanns[thread_id] = trapezoides(a_local, b_local, n_local, h);
    }

    for (int i = 0; i < nthreads; i++) {
        riemann += riemanns[i];
    }

    printf("Con n = %d trapezoides, nuestra aproximación\nde la integral de %d a %d es = %f\n", n, a, b, ri

    return 0;
}
```

Imagen 5.2: Salida del programa (función 1)

```
stiv@stiv-VirtualBox:~/Documentos/PC$ ./riemann_omp_nocrit 2 10 100
Con n = 10000000 trapezoides, nuestra aproximación
de la integral de 2 a 10 es = 330.668179
stiv@stiv-VirtualBox:~/Documentos/PC$
```

Imagen 5.3: Salida del programa (función 2)

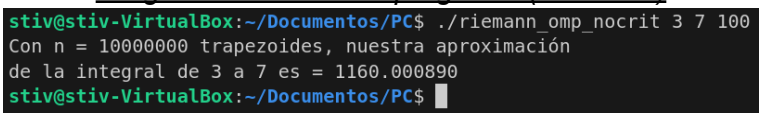
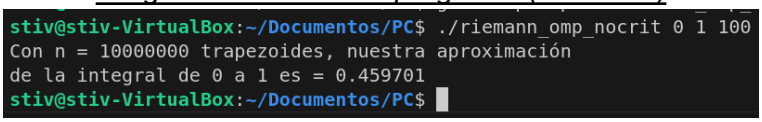


Imagen 5.4: Salida del programa (función 3)



- ¿Qué diferencia hay entre usar una variable global para añadir los resultados a un arreglo?
  R: Entre las ventajas de utilizar un arreglo para almacenar los resultados individuales de cada thread, en primer lugar, está que eliminamos el riesgo de que exista alguna condición de carrera, puesto que ahora cada thread tendrá acceso libre a su posición asignada del arreglo, en lugar de verificar que solo uno pueda modificar la variable global a la vez. Por otro lado, esto también apoya la escalabilidad del programa, puesto que ahora se elimina la parte secuencial relacionada a la modificación de la variable global, y en cambio se permite modificar al arreglo de manera paralela, para finalmente solo realizar la suma de sus componentes.