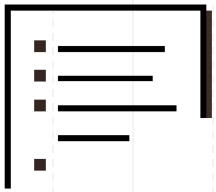
UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

Programación con Memoria Compartida: OpenMP

CC3069 – Computación Paralela y Distribuida
Ciclo 1 - 2023 – Semana 4 (ver 1.0)

Agenda

- Generales de Memoria Compartida y OpenMP
- Uso de directivas de compilador para crear concurrencia
- #pragmas (directivas) y cláusulas generales
- Paralelismo de hilos



Motivación

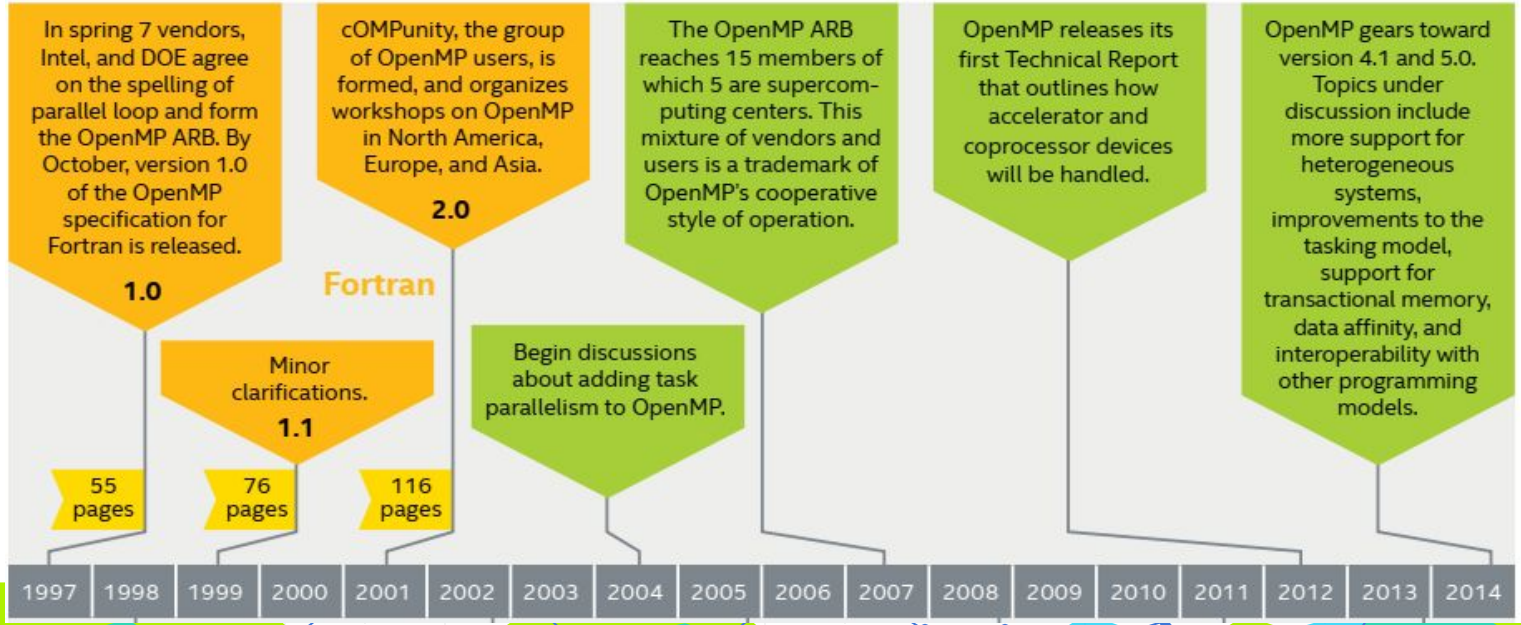
- El proceso de partición es complicado y tedioso
- OpenMP diseñado para reducir el esfuerzo
  - Conversión incremental secuencial –> paralelo
  - Aprovecha paralelismo implícito
- Tan efectivo como sea la implementación del compilador

Open specificifications for Multi Processing - OpenMP

- Modelo estándar de programación paralela (memoria compartida) – API
- Portátil y multi-plataforma (arquitecturas)
- Permite incremento del paralelismo
- Basado en extensiones del compilador y funciones de bibliotecas
- Fortran; C/C++

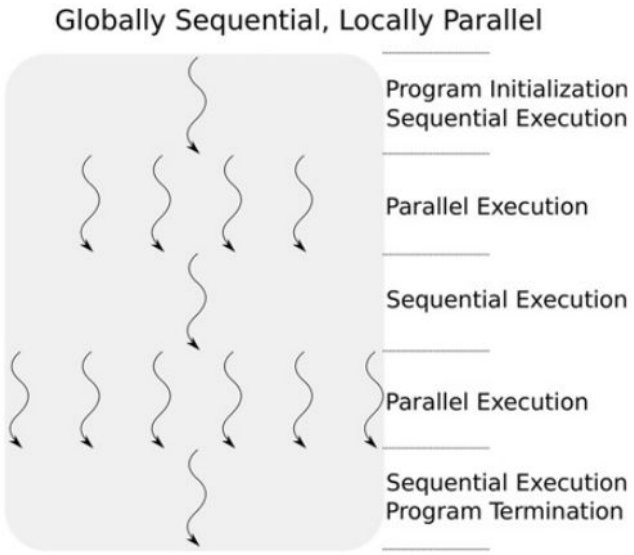
Timeline de OpenMP

En 1996 habían muchas soluciones para paralelismo de loops, pero había problemas de portabilidad y mantenimiento.



Paradigma de OpenMP

- Programas Globalmente Secuenciales, Localmente Paralelos
- Fork-join



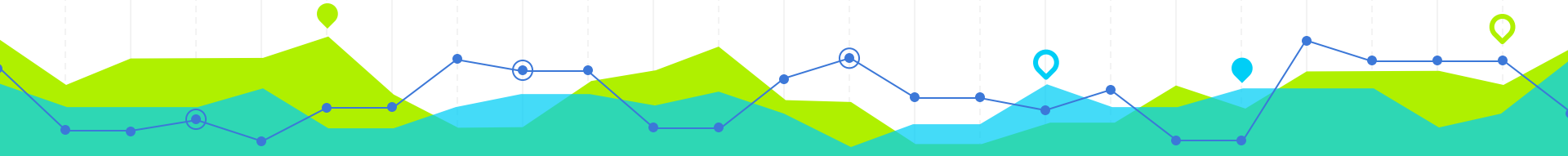
Fuente: G. Barlas, 2015

Conceptos principales de OpenMP

- **Bloque estructurado** – enunciado o conjunto de instrucciones con un punto de entrada y uno de salida.
- **Constructo** – directiva de OpenMP y sus enunciados asociados, para controlar un bloque estructurado o for-loop.
- **Región** – el código dentro de una construcción, incluyendo las llamadas a otras funciones o subrutinas.
- **Región paralela** – una región ejecutada concurrentemente por varios hilos. Una región es dinámica, una construcción, estática.
- **Hilo maestro** – el hilo que ejecuta la parte secuencial y que genera a los hilos hijos.
- **Equipo** – conjunto de hilos ejecutando una región paralela.

“

Hello World!



“Hello World” con OpenMP

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

void openmp_hello(void);

int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);

#pragma omp parallel num_threads(thread_count)
   openmp_hello();

   return 0;
}

void openmp_hello(void) {

   printf("Hola mundo\n");

}
```

Trate de identificar las definiciones anteriores en el código

Compilación de programas

- Incluir el encabezado <omp.h>
- Con g++ / gcc compile de la siguiente forma:

```
$gcc hello.cpp -fopenmp -o hello -lstdc++
```

```
$g++ hello.cpp -fopenmp -o hello
```

- La ejecución se hará llamando al ejecutable según su plataforma. Pase como argumento el número de hilos:

```
$./hello 4
```

Ejercicio 1

- Verifique tener GCC en su computadora (gcc en WLS, Linux – gcc con homebrew en MacOS)
- Escriba y compile el programa mostrado anteriormente en un archivo **openmp_hello.c**

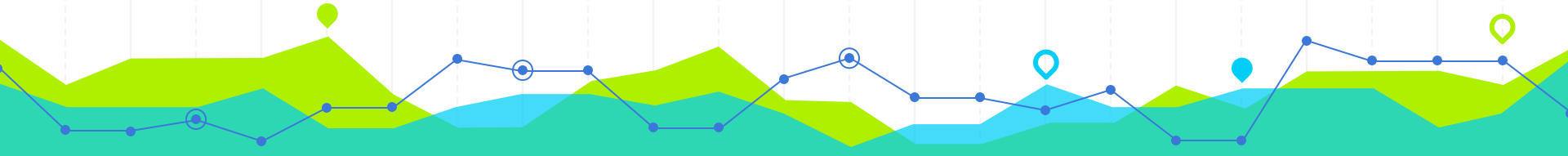
  gcc -o hello hello_omp.c -fopenmp
- Ejecute el programa con 4 hilos:

  ./hello 4

Directivas #pragma

- `#pragma` son directivas o instrucciones para acceder al preprocesador del compilador
- OpenMP usa sus propias directivas.
  - Cada directiva tiene opciones adicionales indicadas mediante ***cláusulas***.
- Si el compilador no soporta la API, estas instrucciones son ignoradas (*)

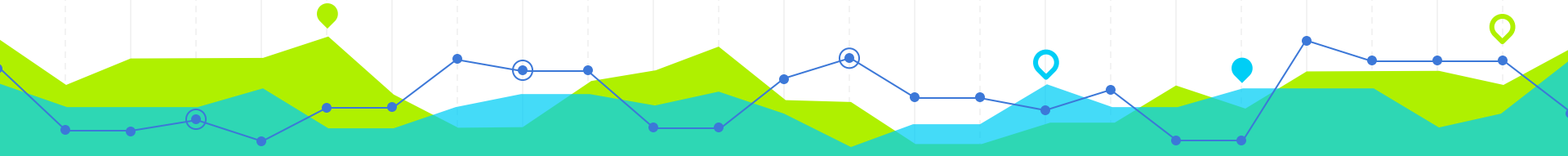
```
#pragma omp parallel num_threads()
```



parallel

- Directiva básica.
- Número de threads que ejecutan la sección crítica determinado por el sistema durante run-time.
- Usa modelo fork-join para crear hilos derivados

```
#pragma omp parallel
```



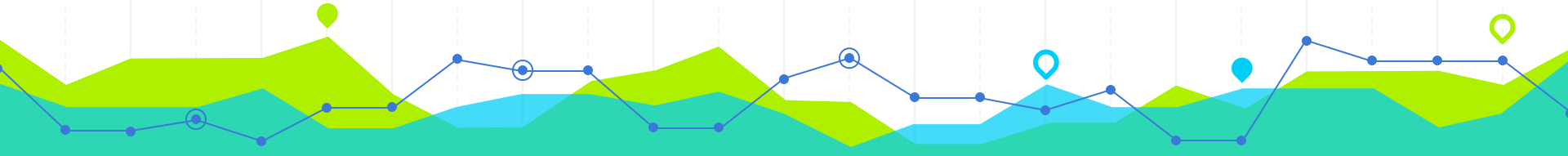
barrera implícita

- Todo bloque paralelo tiene una barrera definida por el sistema
- Se asegura que todos los hilos regresen de la llamada a la rutina dentro del bloque paralelo

```
#pragma omp parallel num_threads(thread_count)
    openmp_hello();
    return 0;
}
```

restricciones

- Número límite de hilos por programa definido por OS.
- El estándar de OpenMP no garantiza que se inicien tantos hilos como **`thread_count`**. Pero ...
  - Sistemas modernos − cientos / miles de hilos por programa.
  - Casi siempre vamos a obtener el número deseado de hilos (a menos que sean miles)



funciones importantes

- **omp_get_num_threads()** – retorna ***int*** con el número de hilos en ejecución al momento de la llamada.
- **omp_get_thread_num()** – retorna ***int*** con el identificador del hilo, un número entre 0 y n-1.
- **omp_get_max_threads()** – retorna ***int*** con el número máximo de hilos que el programa usará.
- **omp_get_num_procs()** – retorna ***int*** con el número de núcleos o procesadores del computador.

sincronía

- **#pragma omp barrier** – crea una barrera explícita para los hilos iniciados en el bloque paralelo anterior.
- **#pragma omp critical** – crea una región crítica donde se necesita que los hilos eviten condición de competencia con acceso de uno en uno.
- **#pragma omp atomic** – crea una región crítica para que un hilo actualice una referencia de memoria, por ejemplo incremento / reducción.

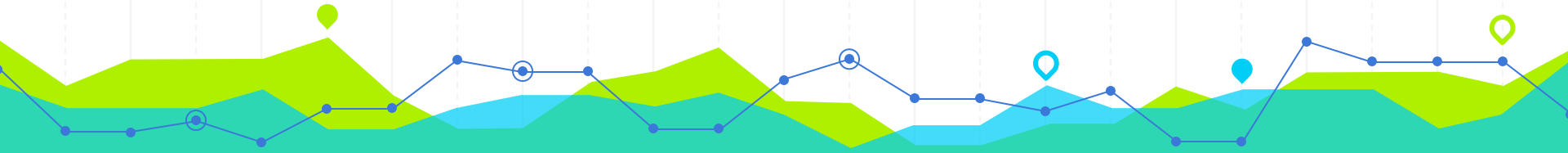
Ejercicio 2

- Modifique el programa openmp_hello.c para que imprima el ID de cada hilo y el total de hilos <N>. Guárdelo con otro nombre
  - Use la función correcta y asígnela a variables tipo ***int***.
- Imprima los siguientes mensajes dependiendo del ID:
  - ID impar – “Feliz cumpleaños número <N> !”
  - ID par – “Saludos del hilo <ID>”

```
./ejectuable <SU EDAD>
```

Control del número de hilos en el equipo

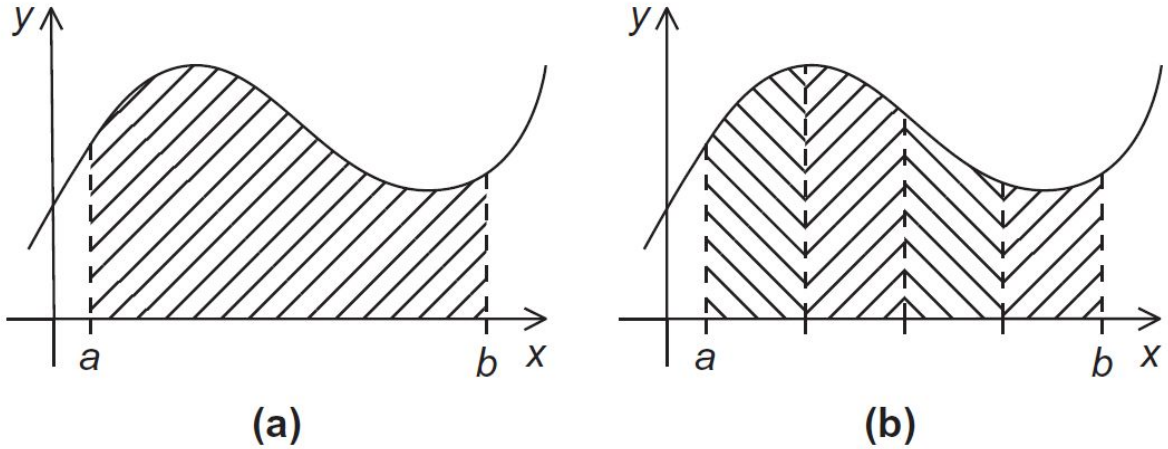
- Via la variable de ambiente OMP_NUM_THREADS
  - echo $OMP_NUM_THREADS (si existe, devuelve resultados)
  - export OMP_NUM_THREADS=”x” (shells POSIX sh, dash, bsh…)
  - setenv OMP_NUM_THREADS=”x” (csh)
- **A nivel de programa**: mediante la función `omp_set_number_threads` fuera del constructor de OpenMP
- **A nivel de pragmas**: mediante la claúsula `num_threads`.
- La función `omp_get_num_threads` devuelve el número de hilos activos en una región paralela



Regla Trapezoidal

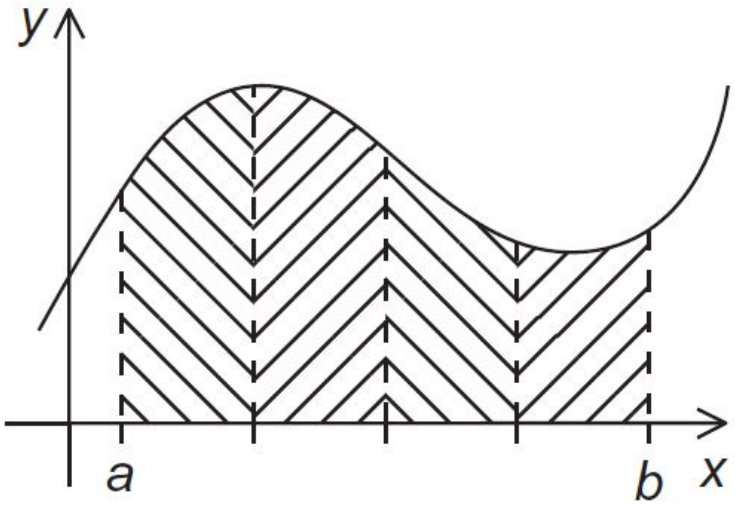
Área bajo la curva

Podemos estimar el área bajo una curva dividiéndola en trapezoides de tamaño finito.



Trapezoides

- Intervalo definido a-b
- N subintervalos iguales
- Trapezoides:
  - base h = (b − a)/ N
  - Lados: $f(x_i)$ y $f(x_{i+1})$.
  - Lado 4, secante entre $f(x_i)$ y $f(x_{i+1})$.

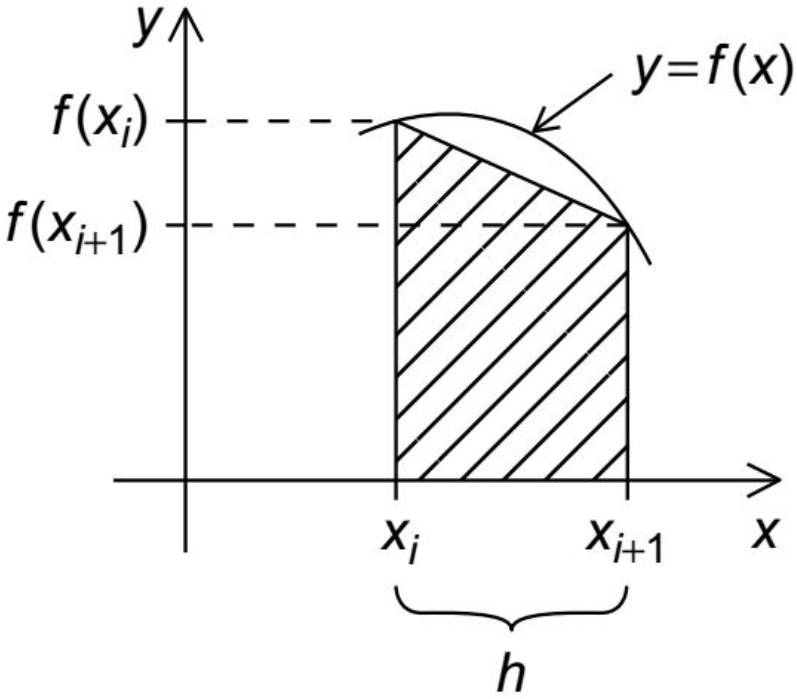


Área del trapezoide

$$A = h(a_2 + a_1)/2$$

- $h = x_{i+1} - x_i$.
- $a_1 = f(x_i)$; $a_2 = f(x_{i+1})$.

$$A = \frac{h}{2}\,[f(x_{i+1}) + f(x_i)]$$

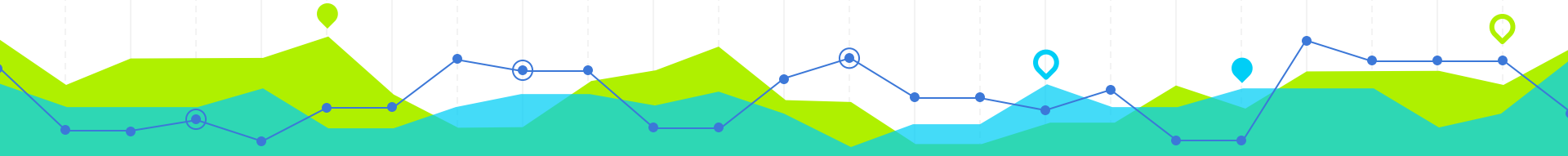


Enfoque computacional (1)

Input

$$
\begin{cases}
x_0 = a \\
x_1 = a + h \\
x_2 = a + h + h = a + 2h \\
x_{n-1} = a + (n-1)h \\
x_n = b
\end{cases}
$$

Secuenciamos cada término como una progresión del inicio del intervalo y el ancho de cada trapezoide



Enfoque computacional (2)

Output:

$$Sum = T_0 + T_1 + T_2 + T_3 + \ldots + T_n$$

$$= h/2\,[f(x_0) + f(x_1)] + h/2\,[f(x_1) + f(x_2)] + h/2\,[f(x_2) + f(x_3)] + \ldots$$

$$+ h/2\,[f(x_{n-1}) + f(x_n)]$$

$$= \mathbf{h/2}\, f(x_0) + \mathbf{h} f(x_1) + \mathbf{h} f(x_2) + \mathbf{h} f(x_3) + \ldots + \mathbf{h/2}\, f(x_n)$$

Solamente $f(x_0)$ y $f(x_n)$ son divididos por 2

Serialización

$$Sum = h\,[f(x_0)/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + f(x_n)/2]$$

- $Subsum\ 1 = [f(x_0) + f(x_n)]/2$
- $Subsum\ 2 = f(x_1) + f(x_2) + \ldots + f(x_{n-1})$
- $Sum = h(Subsum\ 1 + Subsum\ 2)$

Código secuencial

$$Sum = h\,[f(x_0)/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}) + f(x_n)/2]$$

```
integral = (f(a) + f(b)) / 2.0;

for(k = 1; k <= n-1; k++) {
    integral += f(a + k*h);
}
integral = integral * h;
```

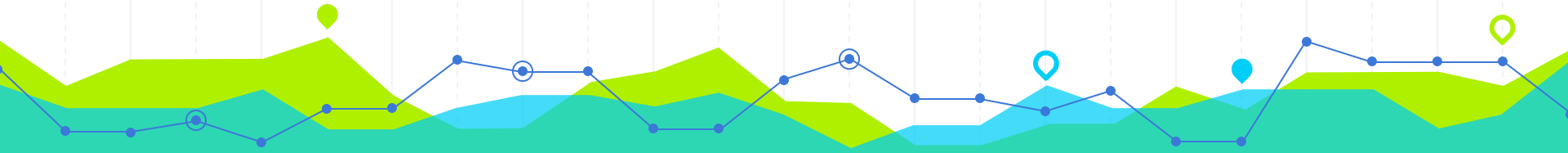
Ejercicio 3

- Descargue y compile el programa `riemann.c`

```
gcc -o riemann riemann.c
```

- En parejas, discuta cómo funciona el programa, el cálculo de las sumas y el paso de parámetros
- Modifique el programa para incluir un bloque paralelo en el segmento de código correcto.
- Modifique el programa para pasar el número de hilos como argumento.

```
gcc -o riemann2 riemann2.c
./riemann2 a b threads
```

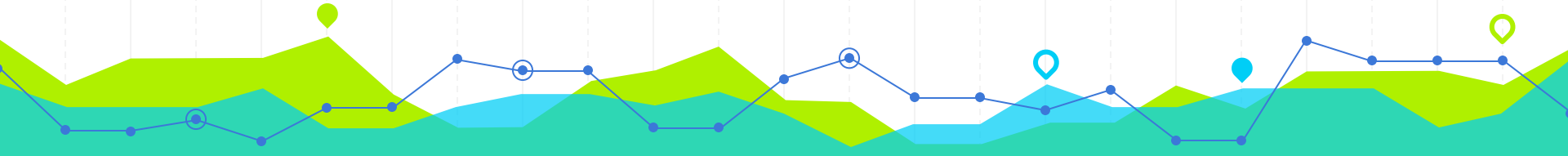


Paralelizando el programa secuencial

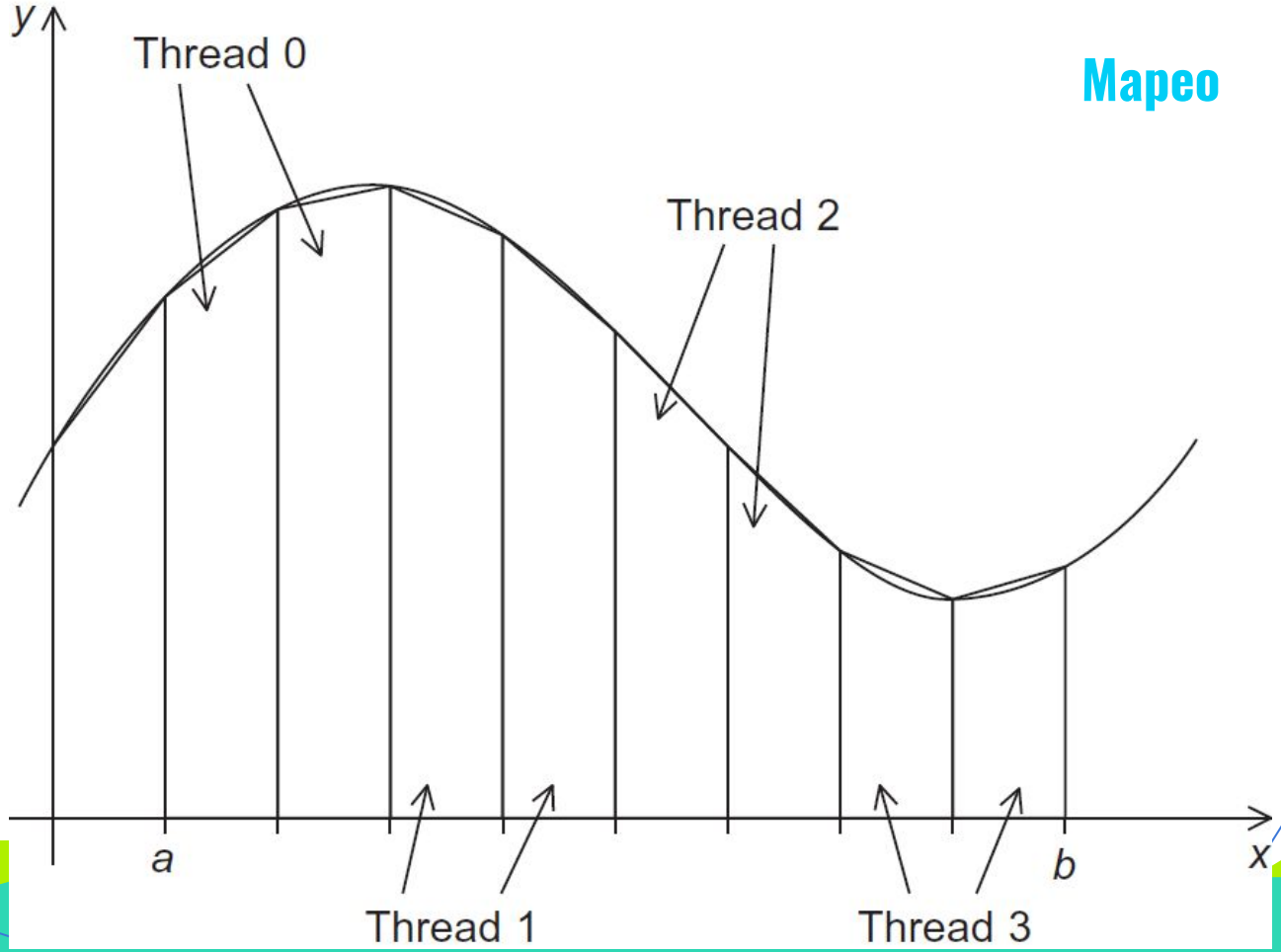
División de Dominio

Asumimos que habrán muchos más trapezoides que núcleos.

- Vamos a dividir el total de trapezoides (data) dentro del número de hilos – División de Dominio
- Asegurarse que el número trapezoides sea múltiplo del número de hilos.



Mapeo



Parámetros locales

Para asegurarnos que cada hilo realice el trabajo correcto, debemos crear explícitamente los parámetros de cada uno.

- Número local de trapezoides
- Valor inicial local del rango – a
- Valor final local del rango – b

Parámetros locales

- Número local de trapezoides
  - trapezoides / hilos
- Valor inicial local del rango – a_local
  - a + (ID * n_local * ancho_h)
- Valor final local del rango – b_local
  - a_local + (n_local*anch_h)

Parámetros locales

```
double integral, h, x, local_sum;
double local_a, local_b;
int i, local_n;
int thread_ID = omp_get_thread_num();
int thread_count = omp_get_num_threads();

//---- Ancho de cada trapezoide
h = (b-a)/n;
//---- Valores locales del hilo
local_n = n/thread_count;
local_a = a + thread_ID*local_n*h;
local_b = local_a + local_n*h;
```

Parámetros locales

- a = 0, b= 100, n=500, hilos = 4
- h = (100-0)/500 = 0.2, n_local = 500/4=125

ID	span = n_local*h offset = ID*span	a_local = a + offset	b_local = a_local + 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

Ejercicio 4

- Modifique el programa riemann2 para incluir los valores locales
  - n_local
  - a_local
  - b_local
- Recuerde incluir las funciones OpenMP para obtener:
  - Número de hilos
  - ID del hilo

Race condition

Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Varios hilos accediendo a la sección crítica (condición de competencia):

suma_global += suma_local;

Exclusión

La directiva CRITICAL controla el acceso de un hilo a una sección crítica (lo serializa):

```
# pragma omp critical
      global_result += my_result ;
//sección crítica
```

OpenMP ver.1
Llamada a rutina Trap

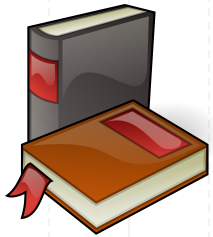
```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
} /* Trap */
```

OpenMP
Rutina Trap con directiva *CRITICAL*

Ejercicio 5

- Modifique el programa riemann2 para agregar
  - suma_local
  - suma global como puntero
  - acumulación de suma_local a suma_global
- Recuerde:
  - Pasar la suma_global como referencia (&) a la rutina de hilos
  - Proteger la sección crítica donde acumulamos la suma_local a la suma_global



Referencias

- **Barlas, G**. “4 − Shared-memory programming: OpenMP” Multicore and GPU Programming An Integrated Approach. Morgan Kauffmann (2015).
- **Pacheco,** . “5 − Shared-memory programming with OpenMP” Multicore and GPU Programming An Integrated Approach. Morgan Kauffmann (2015)
- **Trobec, R**. **et al** “3 − Programming Multi-core and Shared Memory Multiprocessors Using OpenMP” Introduction to Parallel Computing. Springer (2018)