

Introducción

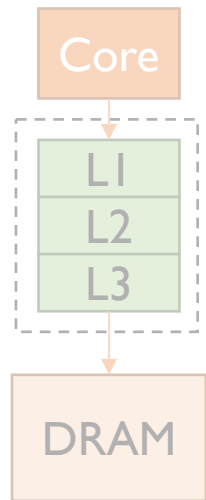
Memoria Compartida

Programación en Paralelo

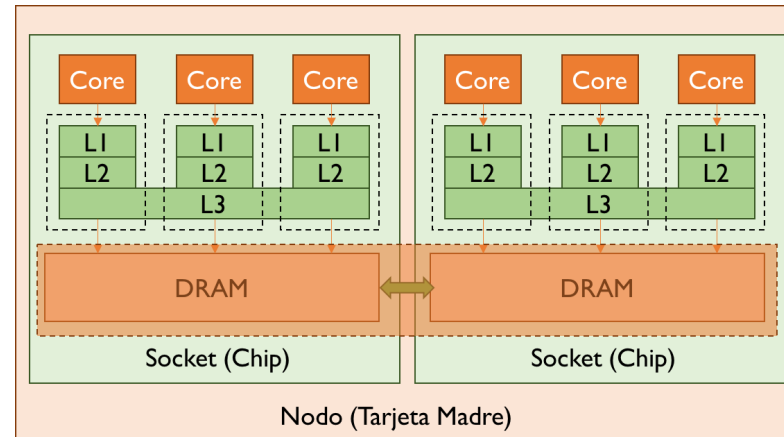
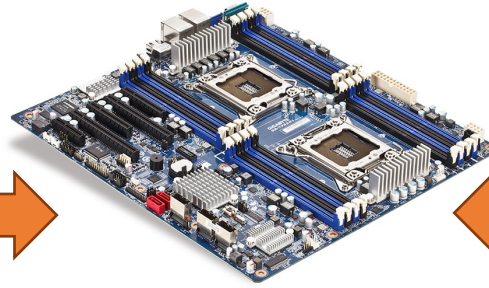
ICME Summer Workshop @ Santiago: Fundamentals of Data Science

Cindy Orozco

Hardware



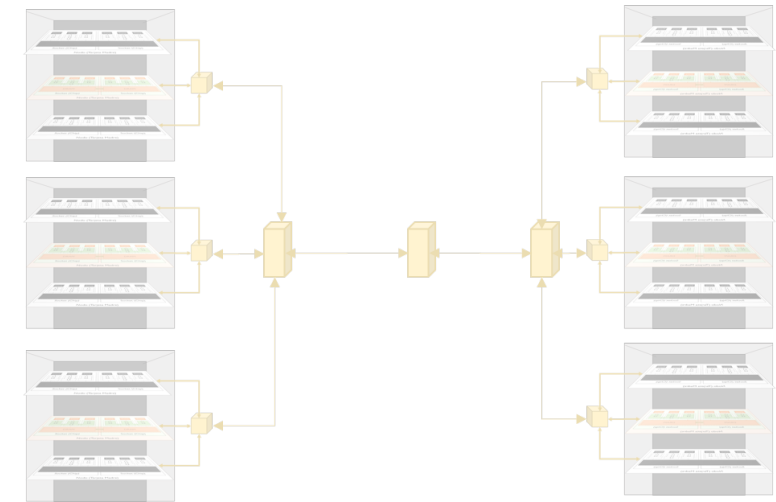
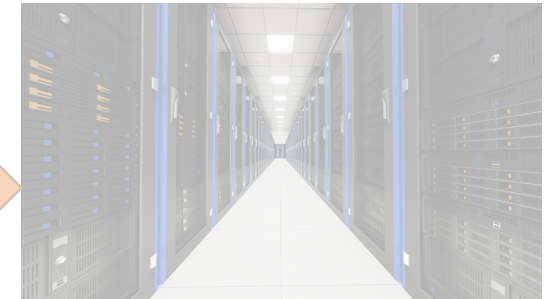
0) Procesamiento Secuencial



1) Memoria Compartida

- ✓ Evitar conflictos memoria
- ✓ Distribuir Tareas

OpenMP



2) Memoria Distribuida

- ✓ Comunicación entre nodos

MPI

- ✓ Asignar Tareas

Planificador de Tareas

OpenMP

Es una API (Interfaz de programación de aplicaciones) para escribir programas con multithreads.

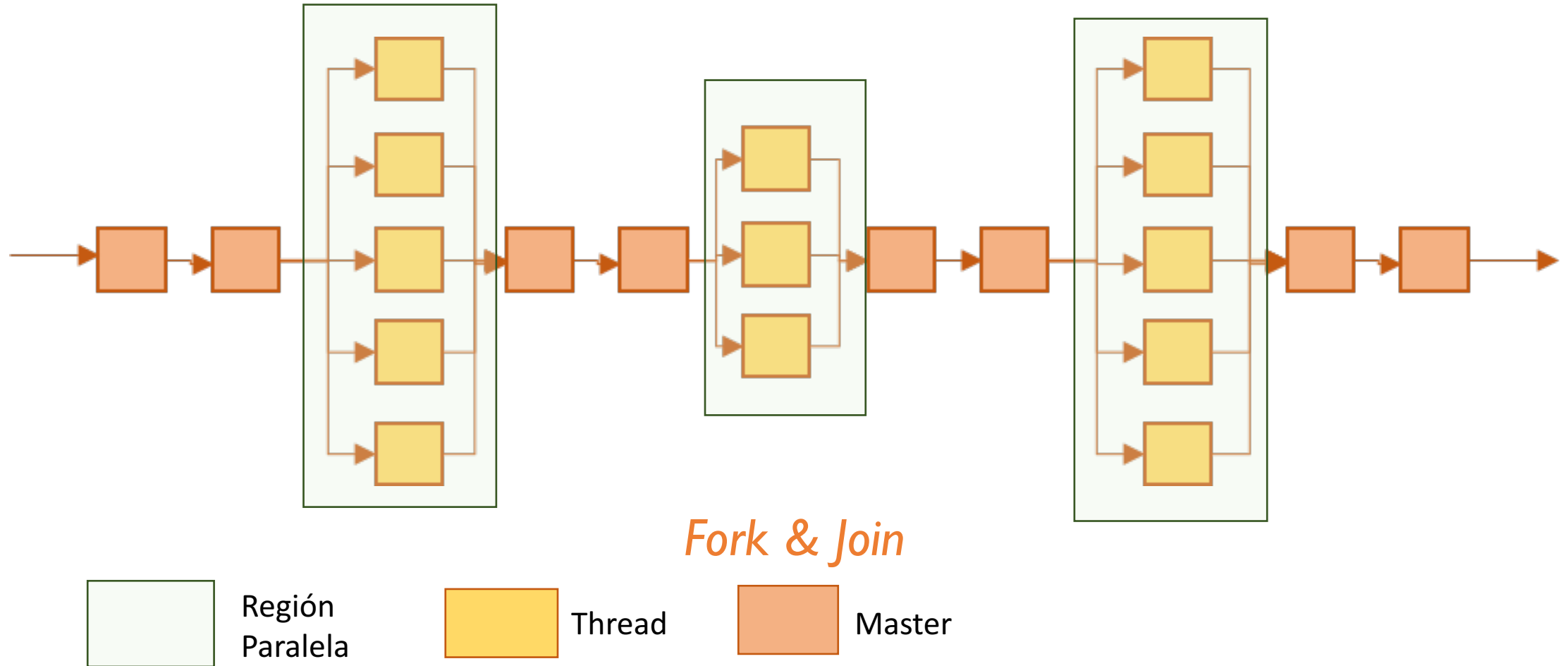
- ✓ Contiene directrices de compilación + librería de rutinas
- ✓ Facilita la implementación en Fortran, C y C++
- ✓ Tiene soporte de reconocidos fabricantes de software y hardware (Intel, IBM, Cray).
Está incluido en los compiladores más usados:
 - ✓ GNU: gcc
 - ✓ LLVM: clang

Este ejercicio y muchos más recursos en:

<http://www.openmp.org>

Granularidad Fina

- Pequeñas subrutinas del algoritmo pueden dividirse en múltiples procesadores

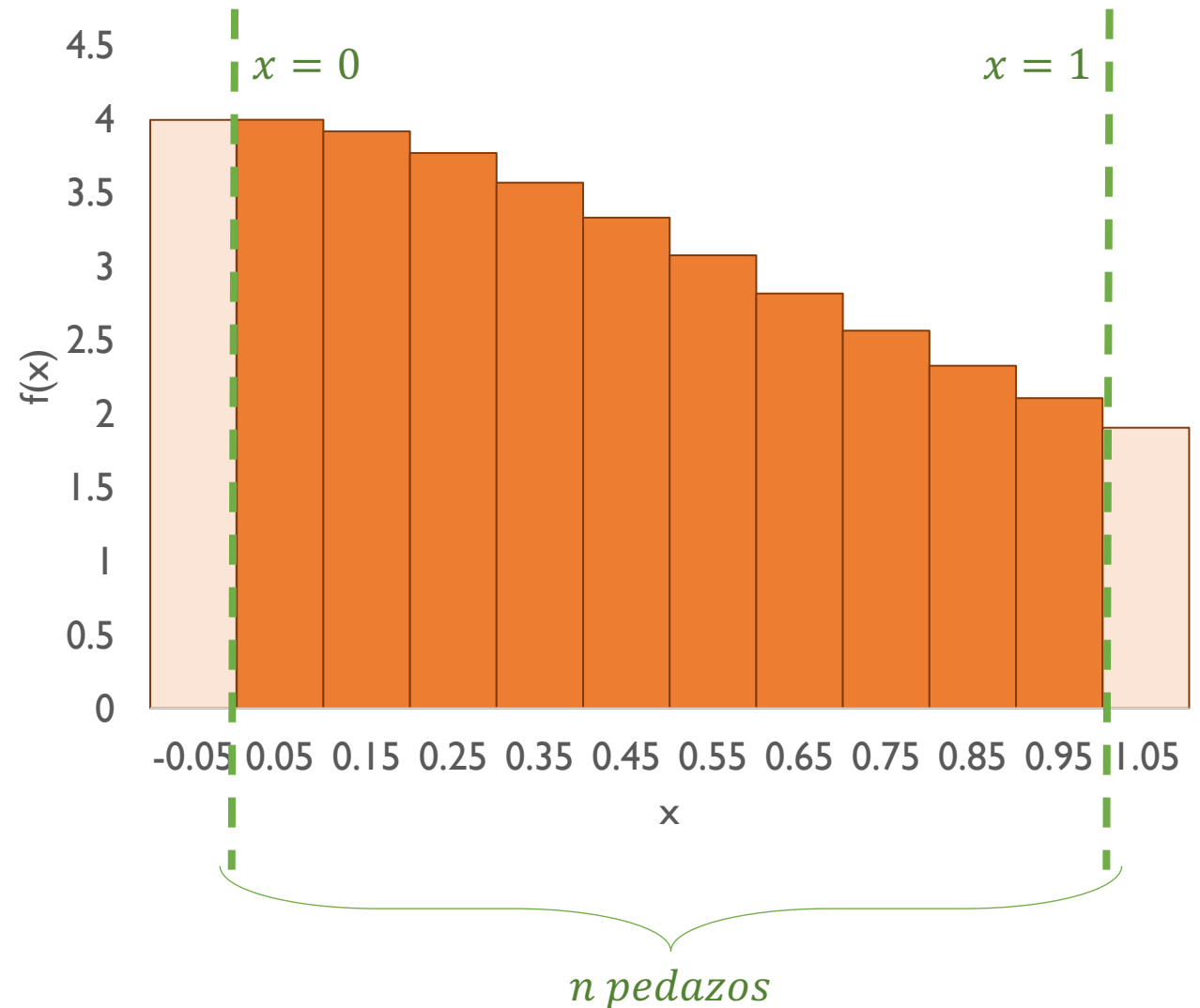


Problema: Calcular π

$$\pi = \int_0^1 \frac{4}{1+x^2} dx \approx \sum_{i=0}^{n-1} f(x_i) * \frac{1}{n}$$

donde

$$x_i = \frac{i + 0.5}{n}$$



Problema: Calcular π / Código Serial

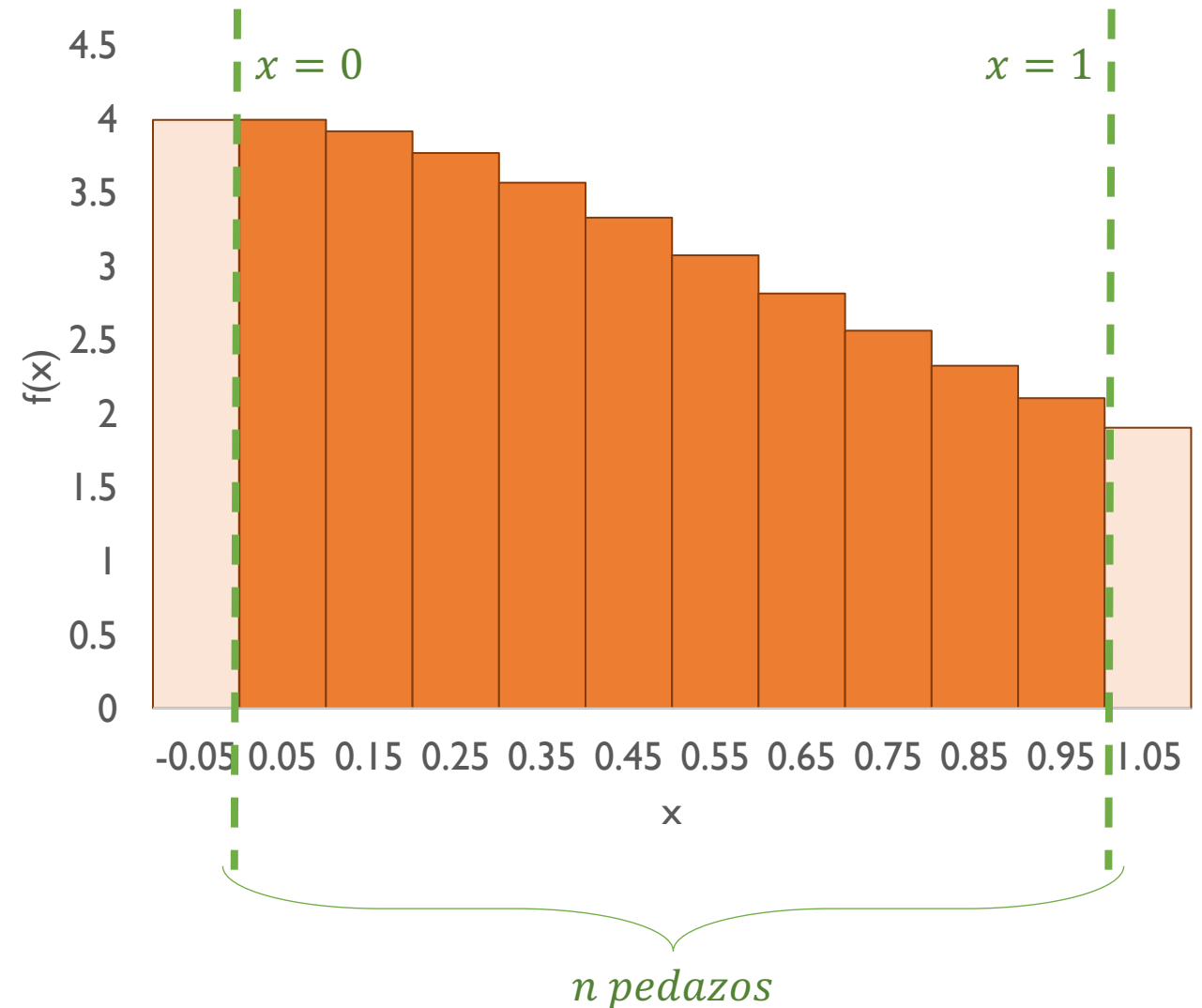
```
#include <stdio.h>
int main(){
    double sum = 0;
    int n = 10;
    double step = 1. / ((double) n);
    double x;
    for (int i = 0; i < n ; ++i){
        x = (i + 0.5) * step;
        sum += 4. / (1. + x * x);
    }
    sum = sum * step;
    printf("El valor de pi es %f",sum);
}
```

$$\sum_{i=0}^{n-1} f\left(\frac{i + 0.5}{n}\right) * \frac{1}{n}$$

Cada iteración es independiente
→ Paralelizar


Problema: Calcular π


Si tenemos $p = 2$ procesadores

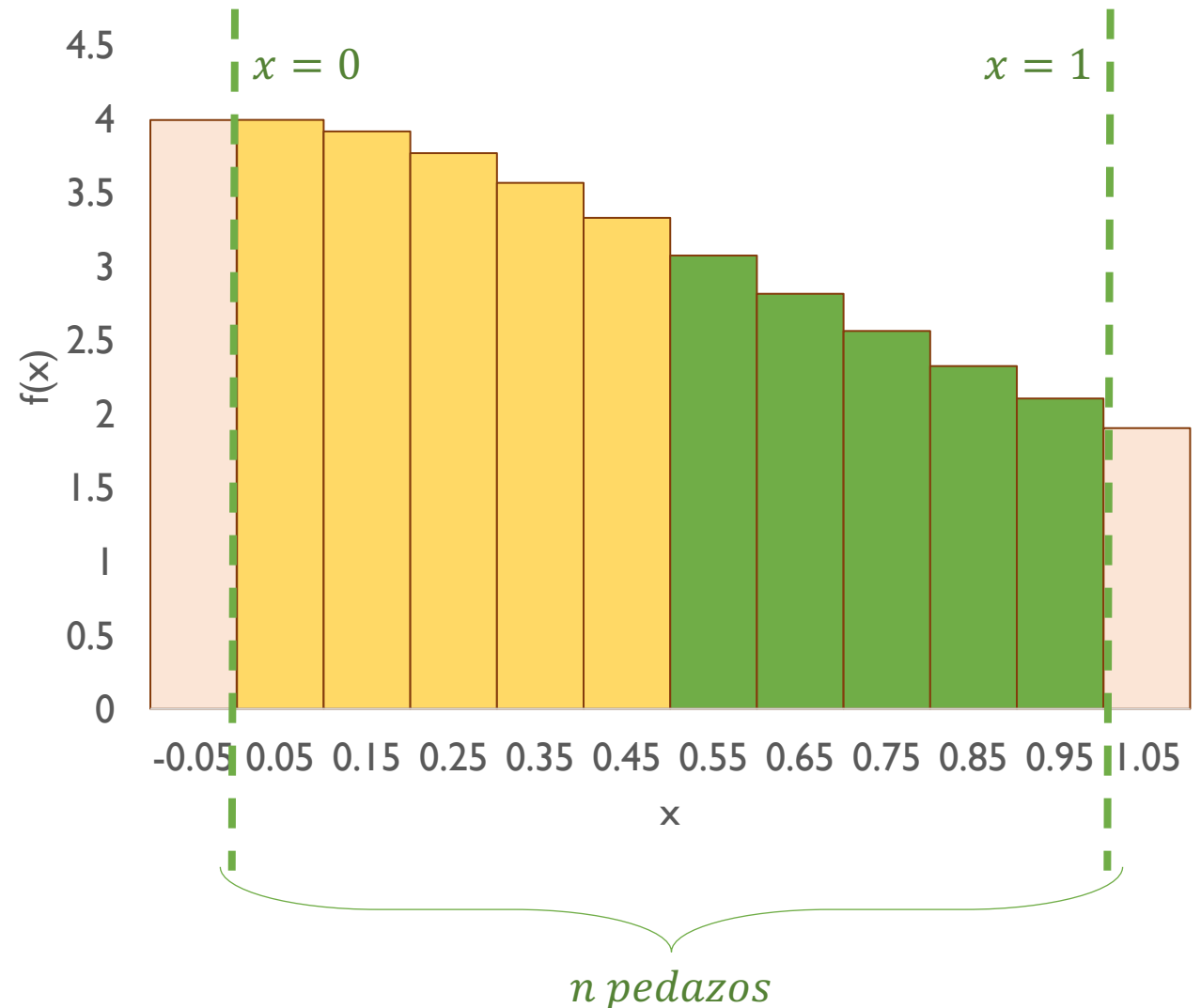


Problema: Calcular π

Si tenemos $p = 2$ procesadores


 Procesador 0 $= \sum_{i=0}^{\frac{n}{2}-1} f(x_i) * \frac{1}{n}$


 Procesador 1 $= \sum_{i=n/2}^{n-1} f(x_i) * \frac{1}{n}$

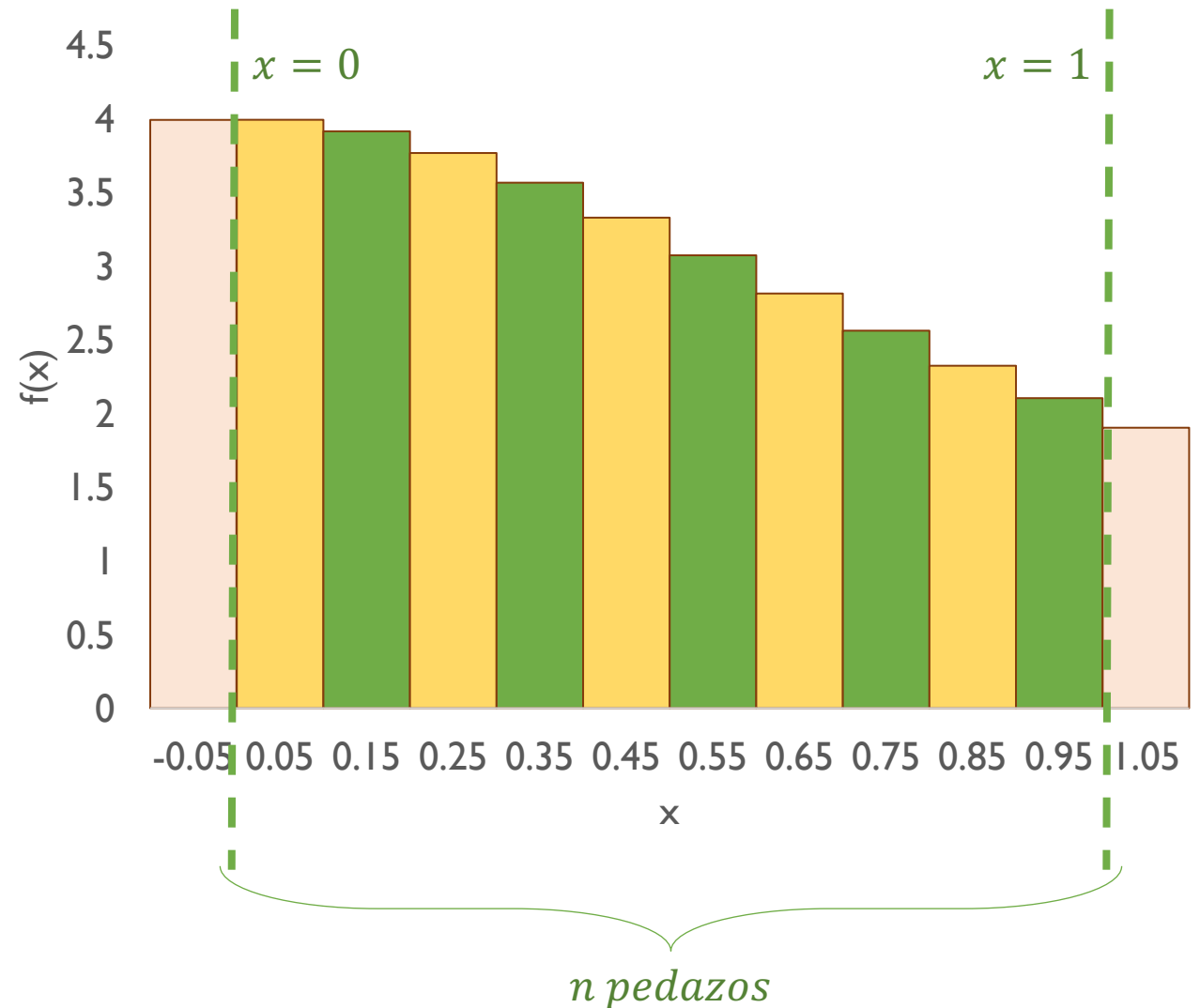


Problema: Calcular π

Si tenemos $p = 2$ procesadores

 Procesador 0 = $\sum_{i=0,2,\dots}^{n-1} f(x_i) * \frac{1}{n}$

 Procesador 1 = $\sum_{i=1,3,\dots}^{n-1} f(x_i) * \frac{1}{n}$



Problema: Calcular π /Código Paralelo, primer intento

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

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

```
#define NUM_THREADS 2
```

```
int main(){
```

```
    omp_set_num_threads(NUM_THREADS);
```

```
    double sum = 0;
```

```
    int n = 10;
```

```
    double step = 1. / ((double) n);
```

```
    #pragma omp parallel  
    {
```

```
        double x;
```

```
        int tid = omp_get_thread_num();
```

```
        int nthreads = omp_get_num_threads();
```

```
        for (int i = tid ; i < n ; i+= nthreads ){
```

```
            x = (i + 0.5) * step;
```

```
            sum += 4. / (1. + x * x);
```

```
        }
```

```
    }
```

```
    sum = sum * step;
```

```
    printf("El valor de pi es %f",sum);
```

```
}
```

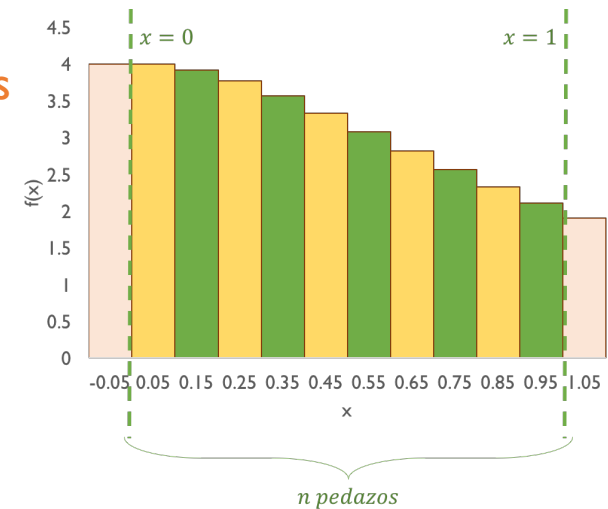
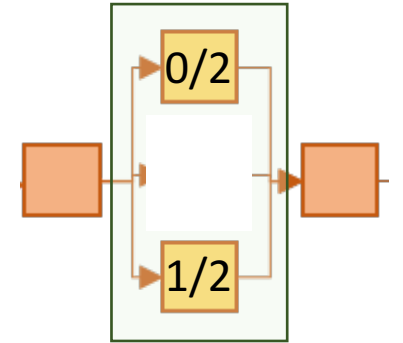
Librería OpenMP

Fijar número de hilos

Iniciar región paralela

Cada hilo identifica ¿quién es?
y ¿Cuántos hay en total?

Subconjunto de iteraciones



Región Paralela: `#pragma omp parallel`

- ✓ Unidad básica de paralelización
- ✓ Genera `NUM_THREADS` hilos
- ✓ Todos los hilos ejecutan la **misma secuencia** de instrucciones, pero se diferencian por un **identificador** = `omp_get_thread_num()`
- ✓ Por defecto, las variables definidas **fuera de la región** son **compartidas** por todos los hilos, y las variables definidas **dentro de la región** son **privadas**

Problema: Calcular π /Código Paralelo, primer intento

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 2
int main(){
    double sum = 0;
    int n = 10;
    double step = 1. / ((double) n);

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x;
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (int i = tid; i < n; i+= nthreads){
            x = (i + 0.5) * step;
            sum += 4. / (1. + x * x);
        }
    }
    sum = sum * step;
    printf("El valor de pi es %f",sum);
}
```

Master

| sum | n | step |
|------|----|------|
| 0 | 10 | 0.1 |
| ¿? | | |
| ... | | |
| 1.72 | | |

Thread 0

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 0 | 2 | |
| 0.05 | | | 0 |

sum+=3.99

Thread 1

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 1 | 2 | |
| 0.15 | | | 1 |

sum+=3.91

Condición de carrera,
¿qué valor agregamos?

¿Cómo evitar la condición de carrera?

Tener más variables privadas

Problema: Calcular π /Código Paralelo, segundo intento

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 2
int main(){
    double sum[NUM_THREADS] = {0};
    int n = 10;
    double step = 1. / ((double) n);

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x;
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (int i = tid; i < n; i+= nthreads){
            x = (i + 0.5) * step;
            sum[tid] += 4. / (1. + x * x);
        }

        for (int j = 1; j < NUM_THREADS; ++j){
            sum[0] += sum[j];
        }

        sum[0] = sum[0] * step;
        printf("El valor de pi es %f", sum[0] );
    }
}
```

Master

| sum [0] | sum [1] | n | step |
|---------|---------|----|------|
| 0 | 0 | 10 | 0.1 |
| 3.99 | 3.91 | | |
| ... | ... | | |
| 16.22 | 15.21 | | |
| 31.42 | | | |
| 3.142 | | | |

Thread 0

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 0 | 2 | |
| 0.05 | | | 0 |

$sum[0] += 3.99$

Thread 1

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 1 | 2 | |
| 0.15 | | | 1 |

$sum[1] += 3.91$

Para $n = 100000$

$T_{\text{serial}} = 21.08 \text{ ms}$

$T_{\text{con 4 threads}} = 12.62 \text{ ms}$

¿ Es posible mejorar este tiempo?

Sí, usando mejor caché

Problema: Calcular π /Código Paralelo, tercer intento

```
#include <stdio.h>
#include <omp.h>
#define NUM_THREADS 2
int main(){
    double sum[NUM_THREADS] = {0};
    int n = 10;
    double step = 1. / ((double) n);

    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x ,mysum;
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        for (int i = tid; i < n; i+= nthreads){
            x = (i + 0.5) * step;
            mysum += 4. / (1. + x * x);
        }
        sum[tid] = mysum;
    }
    for (int j = 1; j < NUM_THREADS; ++j){
        sum[0] += sum[j];
    }
    sum[0] = sum[0] * step;
    printf("El valor de pi es %f", sum[0] );
}
```

Master

| sum [0] | sum [1] | n | step |
|---------|---------|----|------|
| 0 | 0 | 10 | 0.1 |
| 3.99 | 3.91 | | |
| ... | ... | | |
| 16.22 | 15.21 | | |
| 31.42 | | | |
| 3.142 | | | |

Thread 0

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 0 | 2 | |
| 0.05 | | | 0 |

Thread 1

| x | tid | nthreads | i |
|------|-----|----------|---|
| 0 | 1 | 2 | |
| 0.15 | | | 1 |

Para n = 100000

T serial = 21.08 ms

T con 4 threads = 8.80 ms

Inconvenientes de Región Paralela

Soluciones (instrucciones)

Adición de muchas líneas de **código**
para individualizar rutinas

→ Repartir el trabajo

Sincronizar

Fácilmente se pueden tener conflictos
de **Memoria**

→ Manejar Datos

Soluciones para repartir el trabajo

Iteraciones: `#pragma omp for`

Identifica loop para ser ejecutado en paralelo por equipo de threads.

Todas las iteraciones se dividen en pedazos de tamaño (`chunk_size`) y se asignan según el id del thread (`static`) o su disponibilidad (`dynamic`)

`schedule(static/dynamic/auto[chunk_size])`

Secciones: `#pragma omp sections`

Conjunto de secciones no estructuradas para ser repartidas entre threads.

Cada sección se identifica:

`#pragma omp section`

Soluciones para repartir el trabajo

Iteraciones: `#pragma omp for`

¡Sin embargo, no todos los loop pueden ser paralelizables!

```
for (int i = 1; i < n; ++i){  
    a[i] = a[i-1]+1;  
}
```

Se necesita de los anteriores valores para saber el actual, con la directriz seguirá siendo secuencial

Los 2 tipos más usados de for paralelizables son:

Map `b[i]=foo(a[i]);`

Reduce `sum+=a[i];`

`reduction(+/* /max:vars)`

Problema: Calcular π /Código Paralelo, cuarto intento

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

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

Librería OpenMP

```
#define NUM_THREADS 2
```

Fijar número de hilos

```
int main(){
```

```
    omp_set_num_threads(NUM_THREADS);
```

```
    double sum = 0;
```

```
    int n = 10;
```

```
    double step = 1. / ((double) n);
```

```
    double x;
```

```
    #pragma omp parallel for schedule(static,1),private(x),reduction(+:sum)
```

```
    for (int i = 0.; i < n ;++i){
```

```
        x = (i + 0.5) * step;
```

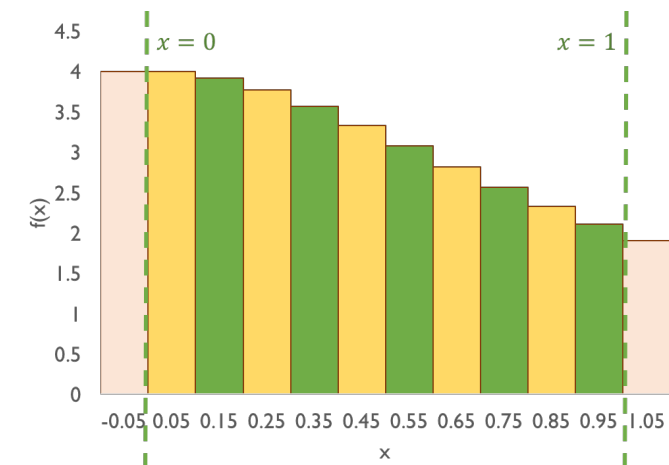
```
        sum += 4. / (1. + x * x);
```

```
    }
```

```
    sum = sum * step;
```

```
    printf("El valor de pi es %f",sum);
```

```
}
```



Soluciones para sincronizar el trabajo

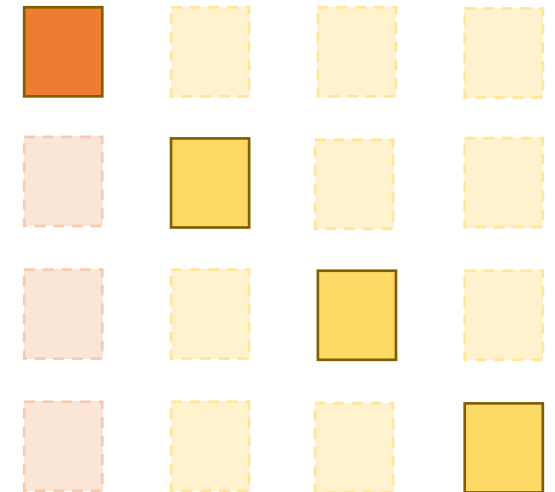
`#pragma omp master` Sólo el master thread ejecuta
código



`#pragma omp single` Sólo uno de los threads ejecuta
código



`#pragma omp critical` Todos los threads ejecutan, pero
sólo uno a la vez



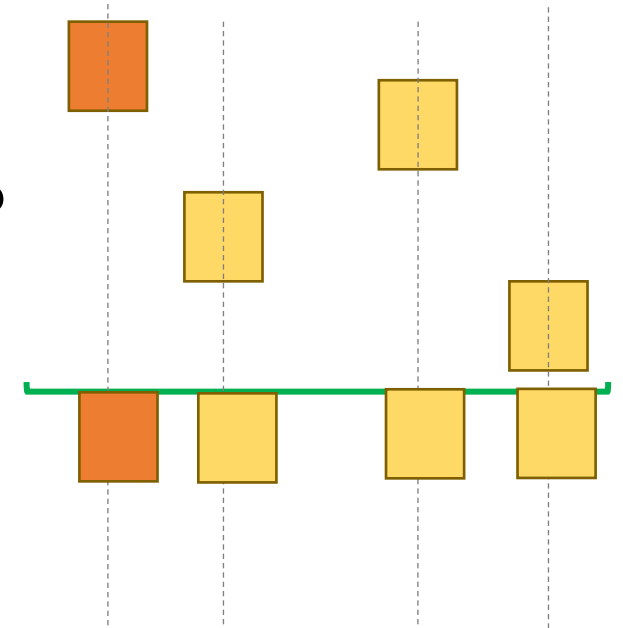
Soluciones para sincronizar el trabajo

`#pragma omp atomic`
código

Solo uno de los threads a la vez modifica estos espacios de memoria

`#pragma omp barrier`

Los threads esperan en este punto hasta que todos lo hayan alcanzado.



Soluciones para manejar la memoria (clausulas)

`shared(vars)`

Valor global compartido por todos los threads

`private(vars)`

Cada thread crea una copia sin inicializar

`firstprivate(vars)`

Cada thread crea una copia y la inicializa con el valor global

`lastprivate(vars)`

Cada thread crea una copia sin inicializar y la variable global toma el valor de la última iteración.

Se adiciona al final de la declaración de región en paralelo

Resumen de instrucciones

- Regiones Paralelas

*Unidades básicas de
paralelización*

`#pragma omp parallel`

- Distribución de trabajo

Asignar tareas a threads

`#pragma omp for`

`#pragma omp sections`

- Sincronización

*En momento t
¿quién hace qué?*

`#pragma omp master`

`#pragma omp single`

`#pragma omp critical`

`#pragma omp atomic`

`#pragma omp barrier`

- Memoria

¿Quién posee la información?

`shared,
private,
firstprivate,
lastprivate`

¿Cómo se usa Open MP?

- Se debe incluir la librería

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

- Para compilar se debe agregar el flag `-fopenmp`

```
gcc -fopenmp hello_world_openmp.c -o hello
```

- Se puede fijar el numero de threads como una variable del sistema

```
export OMP_NUM_THREADS=4
```

- Se corre el programa normalmente

```
./hello
```

Prueba las diferentes soluciones de Pi

- ¿Puedes notar las diferencias que discutimos acerca de memoria y precisión?
- ¿Cómo afecta la cantidad de threads asignados?
- ¿Cómo la cambiarías para cambiar la forma en que se recorren los datos?

