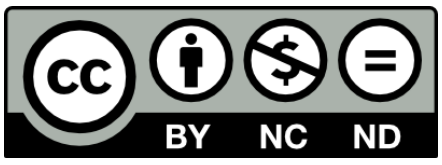
Modelos de Computación Seminario I

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



Modelos de Computación Seminario I

Los Del DGIIM, `losdeldgiim.github.io`

Arturo Olivares Martos

Granada, 2024-2025

Asignatura Modelos de Computación.

Curso Académico 2024-25.

Grado Doble Grado en Ingeniería Informática y Matemáticas.

Grupo 2.

Profesor José Miguel Mantas Ruiz.

Descripción Parcial del Tema 1.

Fecha 25 de septiembre de 2024.

1. Enunciado

Implementar el programa paralelo multihebra para aproximar el número π mediante integración numérica propuesto en el Seminario 1 (incluyendo la medición de tiempos de ejecución).

2. Resolución

Definimos la siguiente función:

$$\begin{aligned} f: \ [0,1] &\longrightarrow \mathbb{R} \\ x &\longmapsto \frac{4}{1+x^2} \end{aligned}$$

El algoritmo consiste en, mediante integración numérica, resolver la siguiente integral:

$$\int_0^1 f(x)\,dx = 4\left[\arctan(x)\right]_0^1 = \pi$$

Esto lo haremos dividiendo el intervalo $[0,1]$ en `num_muestras` (m) muestras equiespaciados, y calculando la suma de Riemman media de la función f en dichos puntos.

$$\pi \approx \frac{1}{m}\sum_{i=0}^{m-1} f\left(\frac{i+0{,}5}{m}\right)$$

Para paralelizar este cálculo, crearemos `num_hebras` (n) hebras que se encargarán cada una de calcular la suma de Riemman en una parte del intervalo $[0,1]$.

Al implementar esto en código, partimos de la plantilla disponible en el Seminario 1, disponible aquí. En el presente documento describiremos los cambios realizados para llegar a la solución final, `S1_Integracion.cpp`, disponible aquí. Este último se puede compilar con la siguiente orden:

```
$ g++ -std=c++11 -pthread S1_Integracion.cpp -o S1_Integracion
```

2.1. Función `calcular_integral_concurrente()`

Esta función se encargará de crear las distintas hebras y de acumular las sumas parciales de cada una de ellas. Se encuentra en el Código Fuente 1. En primer lugar, y tras crear el array correspondiente de futuros, se lanza cada una de las hebras llamándola con su correspondiente función `funcion_hebra()` que describiremos más adelante y su identificador `i`. Posteriormente, recogemos cada una de las sumas parciales llamando al método `get()` de cada uno de los futuros creados.

2.2. Función `funcion_hebra()`

Esta es la función que se ejecutará en cada una de las hebras de forma concurrente. Suponiendo que hay n hebras y m muestras, cada hebra i calculará la suma

```
89 double calcular_integral_concurrente(){
90
        future<double> futuros[num_hebras];

        // Lanzamos cada una de las hebras
        for (long i = 0 ; i < num_hebras ; i++)
95              futuros[i] = async(launch::async, funcion_hebra, i);

        // Acumulamos cada suma parcial
        double suma = 0.0 ;
        for (long i = 0 ; i < num_hebras ; i++)
100             suma += futuros[i].get();

        return suma/num_muestras;
}
```

Código fuente 1: Función `calcular_integral_concurrente()`.

de Riemman de m/n muestras. No obstante, la división de ese número de muestras en n partes (cada una de las hebras) no es trivial, ya que hay distintas formas de hacerlo (como podemos ver en el Código Fuente 2).

Opción 1 Cada hebra i procesa muestras consecutivas. Partiendo de la muestra `i*num_muestras`, procesa las siguientes m/n muestras.

Opción 2 Cada hebra i procesa muestras de forma alternativa. Es decir, partiendo de la muestra `i`, procesa las muestras `i`, `i+n`, `i+2n`, ...

En el análisis de los resultados se compararán ambas opciones.

2.3. Análisis de Resultados

```
62 double funcion_hebra(long i){

        double suma_parcial = 0.0 ;
65
        #define OPT_1

        #if defined(OPT_1)
                double muestras_por_hebra = num_muestras/num_hebras;
70              for (long j = i*num_muestras ; j < i+muestras_por_hebra ; ++j){
                        const double x_j = double(j+0.5)/num_muestras;
                        suma_parcial += f(x_j);
                }
        #elif defined(OPT_2)
75              for (long j = i ; j < num_muestras ; j+=num_hebras){
                        const double x_j = double(j+0.5)/num_muestras;
                        suma_parcial += f(x_j);
                }
        #endif
80
        return suma_parcial;
}
```

Código fuente 2: Función `funcion_hebra()`.