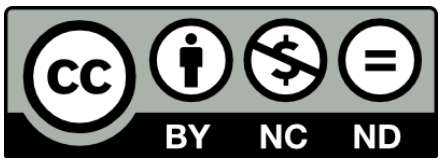
SCD
Seminario I

Los Del DGIIM, losdeldgiim.github.io

Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada



SCD
Seminario I

Los Del DGIIM, `losdeldgiim.github.io`

Arturo Olivares Martos

Granada, 2024-2025

Índice general

1. Cálculo paralelo de una integral

1.1. 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).

1.2. Resolución

Definimos la siguiente función:

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

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
```

```
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()`.

1.2.1. 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.

1.2.2. 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 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.

1.2.3. Análisis de Resultados

En primer lugar, hemos de tener en cuenta que el ordenador personal que vamos a usar para paralelizar tiene un único procesador con dos cores físicos, los cuales a su vez tienen dos cores lógicos cada uno. Por tanto, lanzar más de dos hebras no tendrá sentido en este equipo, ya que no se podrán ejecutar de forma simultánea y se irán alternando el uso del procesador según estime el sistema operativo. En la

```
62 double funcion_hebra(long i){

        double suma_parcial = 0.0 ;
65
        #define OPT_1

        #if defined(OPT_1)
                double muest_hebra = num_muestras/num_hebras;
70              for (long j = i*muest_hebra; j < (i+1)*muest_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()`.

Número de hebras	Tiempo concurrente [ms]	Porcentaje t.conc/t.sec
1	5815.6 ms	112,9 %
2	3069.6 ms	59,44 %
4	3338.4 ms	65,60 %
8	3364.6 ms	66,69 %
16	3347.9 ms	64,14 %

Tabla 1.1: Tiempos para $m = 2^{30}$ y la opción de asignación 1.

Tabla 1.1 se muestran los tiempos manteniendo fijo el número de muestras $m = 2^{30}$ y variando el número de hebras. Como podemos ver, el resultado óptimo se da para dos hebras, ya que es el único caso en el que se pueden ejecutar en paralelo. Para el resto de casos, vemos que el tiempo de ejecución no difiere en gran medida. Un valor resaltable es el porcentaje de tiempo de ejecución concurrente respecto al secuencial en el caso de $n = 1$. Aun esperándose un valor de 100 % al no haber paralelización, este valor es mayor debido a la sobrecarga que supone la creación de la hebra.

A la hora de comparar las dos opciones de asignación de muestras a las hebras, fijamos por tanto el número de hebras en $n = 2$ y el número de muestras en $m = 2^{30}$, obteniendo los resultados de la Tabla 1.2. Como podemos ver, aunque los resultados no difieren en gran cantidad, podemos concluir que la opción 2 (asignación alternada) es ligeramente más eficiente[1].

Por último, es lógico pensar que, conforme más muestras tomemos la aproximación será más precisa. De hecho, intuitivamente la Integral de Riemman se podría entender como tomar un número infinito de muestras. Esto se puede comprobar en

[1]Para un estudio más completo, se podría optar por tomar distintas medidas y obtener la media; pero por simplicidad lo limitaremos a una medida.

Opción de asignación	Tiempo concurrente [ms]	Porcentaje t.conc/t.sec
1	3069.6 ms	59,44 %
2	2695.5 ms	51,46 %

Tabla 1.2: Tiempos para $m = 2^{30}$ y $n = 2$.

Número de muestras	Valor de π secuencial	Valor de π concurrente
–	Valor real (π)	3,14159265358979312
2^{10}	3,14159273306265252	3,14159273306265341
2^{20}	3,14159265358978912	3,14159265358981354
2^{30}	3,14159265358998185	3,14159265359002493

Tabla 1.3: Comparación de valores de π para distintos números de muestras.

la Tabla 1.3, donde vemos que el primer resultado es menos preciso, pero conforme hemos aumentado el número de muestras hemos llegado rápidamente a un resultado muy exacto, pudiendo incluso ser imposible llegar a un resultado más preciso usando los tipos de datos estándares de `C++` debido a su precisión limitada.

Al realizar estas últimas tres mediciones también hemos podido observar que la paralelización en este problema solo tiene sentido para un número de muestras muy elevado, ya que para menos muestras el tiempo de ejecución es prácticamente despreciable y se llega a resultados ilógicos, muchos de ellos causados tal vez por la sobrecarga de la creación innecesaria de 2 hebras. Esto se observa en la Tabla 1.4.

Por último, hemos ejecutado el programa con $n = 2$, $m = 2^{30}$ y la opción de asignación 2, ya que hemos comprobado que es la más eficiente (al tener un porcentaje de tiempo concurrente respecto al secuencial casi del 50 %, que sería el ideal). Esta ejecución la hemos monitorizado con la herramienta `System Monitor` de Linux, cuyo resultado se muestra en la Figura 1.1. Como podemos ver, en un primer momento se emplea uno de los núcleos al 100 %, que consiste en la compilación del programa. Posteriormente, vemos como dicho núcleo (podría haber sido otro) se ejecuta durante cierto tiempo al 100 %, tiempo mientras el cual se ejecuta el programa de forma secuencial. A continuación, se desempeñan todos los núcleos a la vez. Aunque en realidad solo hayamos ejecutado dos hebras, el sistema operativo ha decidido usar los 4 núcleos lógicos disponibles, ya que no había ninguna otra tarea en el sistema. Esto podría hacernos pensar que usar 4 hebras sería mejor, pero ya hemos comprobado anteriormente que no es así.

Nº muestras	T. secuencial [ms]	T. concurrente [ms]	Porcentaje t.conc/t.sec
2^{10}	0.007874 ms	0.22392 ms	2844 %
2^{20}	12.244 ms	9.5976 ms	78,39 %
2^{30}	5187.3 ms	2675.2 ms	51,57 %

Tabla 1.4: Tiempos para $n = 2$ y la opción de asignación 2.

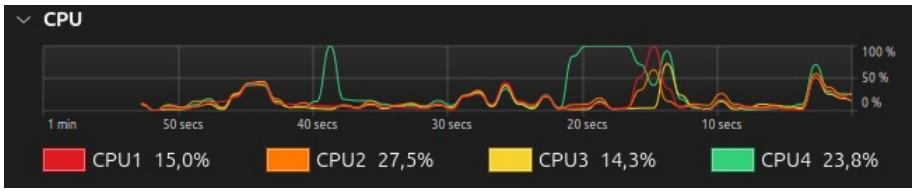


Figura 1.1: Monitorización de la ejecución del programa.