SISTEMAS CONCURRENTES Y DISTRIBUIDOS: SEMINARIO 1

Aproximación paralela de π mediante integración numérica

Montserrat Rodríguez Zamorano

5 de octubre de 2015

1. Cálculo numérico de integrales

La programación concurrente puede ser usada para resolver más rápidamente multitud de problemas, entre ellos los que conllevan muchas operaciones con números flotantes.

Un ejemplo típico es el cálculo del valor I de la integral de una función f de variable real (entre 0 y 1, por ejemplo) y valores reales positivos. Para verificar la corrección del método, se puede usar una integral I con valor conocido. A modo de ejemplo, usaremos una función f cuya integral entre 0 y 1 es el número π.

$$\int_0^1 \frac{4}{1+x^2} dx$$

2. Versión concurrente de la integración

Cada hebra calculará la suma parcial de los valores de f, evaluando la función en m/n puntos del dominio. Para distribuir los cálculos entre las hebras existen dos opciones: hacerlo de forma contigua o entrelazada.

2.1. Versión entrelazada

Como se trata de una versión concurrente, cada hebra evaluará la función en m/n puntos del dominio, pero no lo hará en valores $f(x_i)$ contiguos, de ahí la estructura del bucle: no se avanzará el índice sumando uno, sino n, siendo n el número de hebras en total, de manera que cada hebra comenzará en su índice y después avanzará n muestras, dando lugar a la estructura entrelazada.

Código correspondiente a la función que tiene que realizar cada hebra:

```
void * funcion_hebra_entrelazada( void * ih_void )
{
   unsigned long ih = (unsigned long) ih_void ; // indice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   for(unsigned long i=ih; i<m ; i+=n) {
     sumap += f((i+0.5)/m );
   }
```

```
    resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
    return NULL ;
}
```

Para completar la implementación, se usará la siguiente función, que realiza el cálculo concurrente. Para ello se determinará un número de muestras, que será el número de puntos en los que se evaluará la función, y finalmente se aproximará π como la media de todos estos valores.

```
double calcular_integral_concurrente_entrelazada(){
  // crear y lanzar $n$ hebras, cada una ejecuta "funcion\_concurrente"
  pthread_t hebras[n] ;
  for ( unsigned i = 0 ; i < n ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create ( &(hebras[i]), NULL, funcion_hebra_entrelazada, arg_ptr);
  }

  double resultado = 0; //guarda el resultado de cada hebra

  // esperar (join) a que termine cada hebra, sumar su resultado
  for ( unsigned i = 0 ; i < n; i++) {
    pthread_join( hebras[i] , NULL);
  }
  // devolver resultado completo
  for ( unsigned i = 0; i < n; i++) {
    resultado += resultado_parcial[i];
  }
  return resultado/m; //devolver valor promedio
}
```

2.2. Versión contigua

La versión contigua no se diferencia demasiado de la anterior. Tan sólo hay que variar la forma en la que las hebras toman los valores que van a evaluar. Esta vez no se tomarán alternando los $f(x_i)$, sino en bloques.

Esto se traducirá en el código en que habrá que cambiar la forma de recorrer el bucle en la función que tiene que realizar cada hebra. Habrá que desplazar cada hebra al bloque que le corresponda, lo que se conseguirá multiplicando el índice de la hebra por el número de puntos que tiene que evaluar.

```
void * funcion_hebra_contigua( void * ih_void )
{
   unsigned long ih = (unsigned long) ih_void ; // indice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   for(unsigned long i=ih*m/n; i<(ih+1)*m/n ; i++) {
     sumap += f((i+0.5)/m );
   }
   resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
   return NULL ;
}
```

3. Ejecución del programa y análisis de los datos

Se ha realizado la ejecución en un PC con las siguientes especificaciones técnicas:

Procesador: Pentium(R) Dual-Core
Sistema operativo: Ubuntu 14.04 (32 bits)

A la hora de ejecutar el programa se tendrán en cuenta los siguientes puntos:

1. El número de muestras, m, debe ser múltiplo de n. En este caso hemos fijado $m = 1000000$.

2. Los tiempos varían considerablemente incluso aunque no lo hagan n y m. Para que los datos sean lo más objetivos posible, en el main se ha incluido un bucle que repite la ejecución un número determinado de veces (en este caso se han hecho 100 repeticiones) y muestra el promedio de los tiempos de ejecución.

En la figura (1) se puede ver la salida en terminal tras la ejecución del programa.

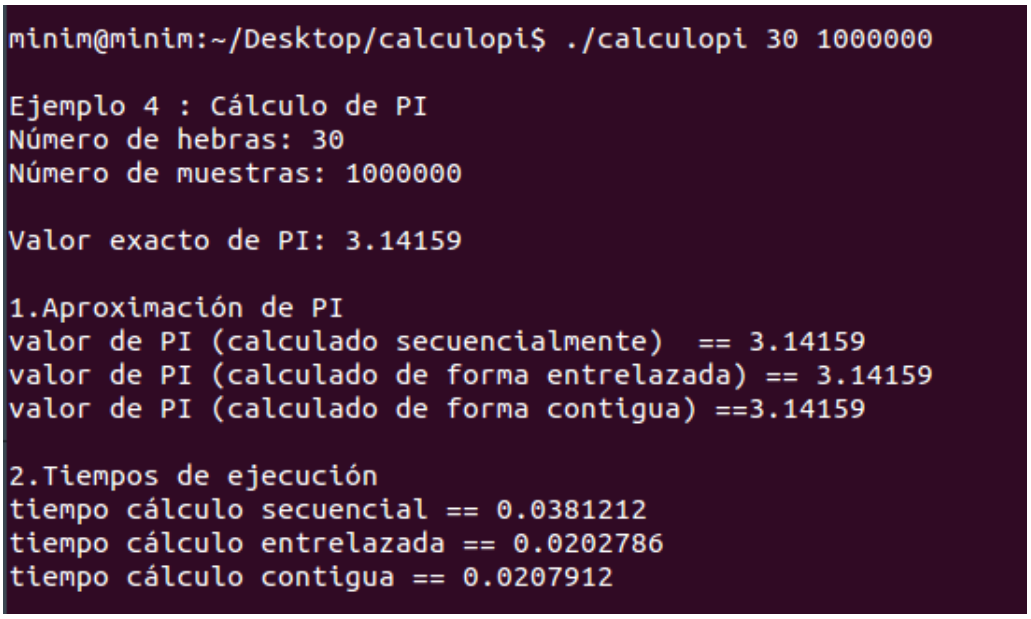

Figura 1: Ejemplo de ejecución del programa

Estudiaremos cómo varían los tiempos de ejecución si aumentamos el número de hebras. Como los tiempos de ejecución son similares en las versiones concurrentes, se tomarán los datos de tan sólo una de ellas.

La tabla (1) recoge los tiempos de ejecución de ambas versiones. Estos datos aparecen también representados en el gráfico (2).

Hebras	Versión secuencial	Versión concurrente
2	0.0289823	0.0169557
4	0.0290449	0.0167735
5	0.0290586	0.0167317
10	0.0289662	0.0164219
20	0.0290133	0.0161194
40	0.0290423	0.0164597
50	0.029057	0.016846
80	0.0290117	0.0172616
100	0.0290421	0.0176222
200	0.0291149	0.0210154
400	0.0291499	0.0253477
1000	0.0292552	0.0414229

Cuadro 1: Tiempos de ejecución dependiendo del número de hebras

Se puede apreciar que en el caso de la versión secuencial, los tiempos permanecen estables aunque no lo

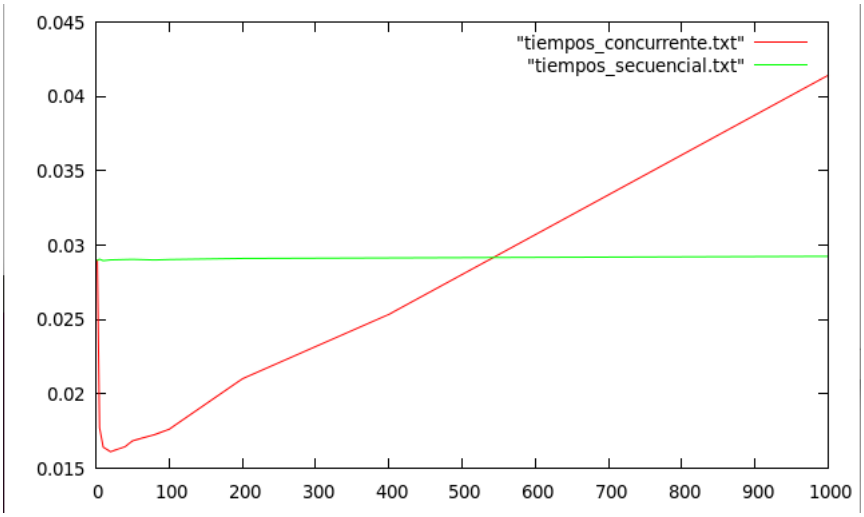

Figura 2: Tiempos de ejecución dependiendo del número de hebras

haga el número de hebras. Es claro que, como la función realiza los cálculos secuencialmente, los tiempos no varían porque su ejecución no depende del número de hebras.

En cuanto a la versión concurrente, se puede ver cómo el tiempo de ejecución disminuye en un primer momento conforme aumenta el número de hebras. Sin embargo, cuando el número de hebras es aproximadamente 40, el tiempo vuelve a aumentar, llegando a superar el tiempo secuencial. Es lo que se conoce como *overhead*: el tiempo de ejecución aumenta porque el coste de la creación y eliminación de hebras llega a superar el coste de realizar los propios cálculos.

En cuanto al número de cores, habría que probar el programa en ordenadores con número de cores distintos para poder comparar los datos. Intuitivamente, cuantos más cores lógicos tenga el ordenador, más hebras podrá lanzar simultáneamente, por lo que los tiempos de ejecución serán menores.

4. Programa concurrente completo

Se incluye aquí el código completo del ejercicio.

```
// ******************************************************************
// SCD. Ejemplos del seminario 1.
//
// Ejercicio calculo PI
// Montserrat Rodriguez Zamorano
// ******************************************************************

#include <iostream>
#include <pthread.h>
#include <stdlib.h>
#include "fun_tiempo.h"

using namespace std ;

// ---------------------------------------------------------------------
//variables globales (compartidas entre hebras)

unsigned long m = 100000 ;  // numero de muestras (un millon)
unsigned n      = 4 ;                       // numero de hebras

double * resultado_parcial; // tabla de sumas parciales (una por hebra)
```

```
// ---------------------------------------------------------------------
// implementa funcion $f$

double f( double x )
{
   return 4.0/(1+x*x) ;      // $~~~~f(x)\,=\,4/(1+x^2)$
}
// ---------------------------------------------------------------------
// calculo secuencial

double calcular_integral_secuencial( )
{
   double suma = 0.0 ;                          // inicializar suma
   for( unsigned long i = 0 ; i < m ; i++ ) // para cada $i$ entre $0$ y $m-1$
      suma += f( (i+0.5)/m );

   return suma/m ;                              // devolver valor promedio de $f$
}
// ---------------------------------------------------------------------
// funcion que ejecuta cada hebra

void * funcion_hebra_contigua( void * ih_void )
{
   unsigned long ih = (unsigned long) ih_void ; // numero o indice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   for(unsigned long i=ih*m/n; i<(ih+1)*m/n ; i++) {
     sumap += f((i+0.5)/m );
   }
   resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
   return NULL ;
}

void * funcion_hebra_entrelazada( void * ih_void )
{
   unsigned long ih = (unsigned long) ih_void ; // numero o indice de esta hebra
   double sumap = 0.0 ;
   // calcular suma parcial en "sumap"
   for(unsigned long i=ih; i<m ; i+=n) {
     sumap += f((i+0.5)/m );
   }
   resultado_parcial[ih] = sumap ; // guardar suma parcial en vector.
   return NULL ;
}

// ---------------------------------------------------------------------
// calculo concurrente

double calcular_integral_concurrente_contigua( )
{
   // crear y lanzar $n$ hebras, cada una ejecuta "funcion\_concurrente"
   pthread_t hebras[n] ;
   for ( unsigned i = 0 ; i < n ; i++ ) {
     void * arg_ptr = (void *) i ; //convertir entero a puntero
     pthread_create ( &(hebras[i]), NULL, funcion_hebra_contigua, arg_ptr);
   }

   double resultado = 0; //guarda el resultado de cada hebra

   // esperar (join) a que termine cada hebra, sumar su resultado
   for ( unsigned i = 0 ; i < n; i++) {
     pthread_join( hebras[i] , NULL);
   }
   // devolver resultado completo
   for ( unsigned i = 0; i < n; i++) {
     resultado += resultado_parcial[i];
   }
   return resultado/m; //devolver valor promedio
}
```

```
double calcular_integral_concurrente_entrelazada(){
  // crear y lanzar $n$ hebras, cada una ejecuta "funcion\_concurrente"
  pthread_t hebras[n] ;
  for ( unsigned i = 0 ; i < n ; i++ ) {
    void * arg_ptr = (void *) i ; //convertir entero a puntero
    pthread_create ( & (hebras[i]), NULL, funcion_hebra_entrelazada, arg_ptr);
  }

  double resultado = 0; //guarda el resultado de cada hebra

  // esperar (join) a que termine cada hebra, sumar su resultado
  for ( unsigned i = 0 ; i < n; i++) {
    pthread_join( hebras[i] , NULL);
  }
  // devolver resultado completo
  for ( unsigned i = 0; i < n; i++) {
    resultado += resultado_parcial[i];
  }
  return resultado/m; //devolver valor promedio
}

// ---------------------------------------------------------------------

int main( int argc , char *argv[])
{
  if(argc == 2)
  {
    n = atoi(argv[1]) ;
  }
  else if(argc == 3)
  {
    n = atoi(argv[1]) ;
    m = atoi(argv[2]) ;
  }
  else if(argc > 3)
  {
    cout << "Demasiados argumentos." << endl;
    exit(0);
  }
   cout << endl << "Ejemplo 4: Calculo de PI" << endl ;
   cout << "Numero de hebras: " << n << endl ;
   cout << "Numero de muestras: " << m << endl;
   resultado_parcial = new double [n];
   double pi_sec = 0.0, pi_conc_ent = 0.0, pi_conc_cont=0.0 ;
   double tiempo_secuencial=0, tiempo_entrelazada=0, tiempo_contigua=0 ;
   int num_rep = 100;

   struct timespec inicio, fin ;

   for (unsigned i=0 ; i < num_rep ; i++ )
   {
     inicio = ahora() ;
     pi_sec  = calcular_integral_secuencial() ;
     fin = ahora() ;
     tiempo_secuencial += duracion(&inicio, &fin);

     inicio = ahora() ;
     pi_conc_ent = calcular_integral_concurrente_entrelazada() ;
     fin = ahora() ;
     tiempo_entrelazada += duracion(&inicio, &fin) ;

     inicio = ahora() ;
     pi_conc_cont = calcular_integral_concurrente_contigua() ;
     fin = ahora() ;
     tiempo_contigua += duracion(&inicio, &fin) ;
    }

   tiempo_secuencial = tiempo_secuencial/num_rep ;
   tiempo_entrelazada = tiempo_entrelazada/num_rep ;
   tiempo_contigua = tiempo_contigua/num_rep ;
```

```
   double pi = 3.14159265358979323846;
   cout << endl << "Valor exacto de PI: " << pi << endl;
   cout << endl << "1. Aproximacion de PI" << endl;
   cout << "valor de PI (calculado secuencialmente)  = " << pi_sec  << endl
        << "valor de PI (calculado de forma entrelazada) = " << pi_conc_ent
        << endl
        << "valor de PI (calculado de forma contigua) = " << pi_conc_cont
        << endl << endl ;

   cout << "2. Tiempos de ejecucion" << endl;
   cout << "tiempo calculo secuencial = " << tiempo_secuencial << endl
        << "tiempo calculo entrelazada = " << tiempo_entrelazada << endl
        << "tiempo calculo contigua = " << tiempo_contigua << endl << endl ;


   return 0 ;
}
// ------------------------------------------------------------------------
```