Implementación multihebra de aproximación numérica de una integral

Juan Antonio Villegas Recio

Implementar el programa multihebra para aproximar el número π mediante integración numérica del seminario 1 (incluyendo la medición de tiempos de ejecución).

Junto a esta memoria se adjunta el código fuente del programa implementado, llamado `ejemplo9.cpp`. Para compilar el programa se utiliza la orden:

```
g++ -std=c++11 ejemplo9.cpp -o ejemplo9 -lpthread
```

Generando así el archivo ejecutable.

Modo de empleo

Se pedía implementar la integral mediante asignación **cíclica** y mediante **bloques contiguos**. Mi código permite elegir entre ambas opciones utilizando argumentos en la línea de comandos. Si se desea ejecutar empleando asignación cíclica se utiliza:

```
./ejemplo9 -c
```

Y si se desea ejecutar asignación con bloques contiguos:

```
./ejemplo9 -b
```

Cualquier otro conjunto de argumentos se considera incorrecto y el programa finaliza su ejecución mostrando un mensaje de error junto una indicación sobre cómo usarlo.

Implementación

Para permitir este uso, en el código hay declarada una variable booleana global llamada `ciclica`, a la cual se le asigna `true` solo si el segundo argumento es `"-c"`, y se le asigna `false` solo si el segundo argumento es `"-b"`.

```
// Comprobamos argumentos
if( argc != 2 ||
( strcmp(argv[1], "-c" )!=0 &&  strcmp(argv[1], "-b" )!=0 ) ){
      cout << "Error en los argumentos. Uso: " << endl <<
          argv[0] << " -c | -b" << endl <<
          "Use la opción \'-c\' para calcular pi mediante asignación cíclica." <<
          endl <<
          "Use la opción \'-b\' para calcular pi mediante asignación por bloques." <<
          endl;
      exit(-1);
```

```
   }

   //Asignamos true o false a ciclica según los argumentos
   if( !strcmp(argv[1], "-c" ) )
      ciclica = true;
   else
      ciclica = false;
```

En la asignación podemos asignar `false` sin comprobar que el segundo argumento es `"-b"` ya que anteriormente se ha comprobado que necesariamente el segundo argumento es `"-c"` o `"-b"`.

Una vez asignado algún valor a `ciclica`, las hebras ejecutarán la función que asigna iteraciones por bloques o la que asigna iteraciones cíclicas. Lo veremos a continuación.

Cálculo de la integral

La función `main` llama a la función `calcular_integral_concurrente( )`, la cual se encarga de declarar un vector de `n` futuros, en cada uno de los cuales cada hebra depositará el valor que calcule en su ejecución. Se lanzan las `n` hebras y dependiendo del valor de `ciclica` ejecutan una u otra función. Una vez lanzadas, cada hebra está programada para que calcule una suma parcial del valor de la integral, para posteriormente sumar los resultados de cada hebra y obtener así el valor de π.

Para recoger el valor calculado por cada hebra usamos el método `get()` que implementan las variables de tipo `future`. en cada elemento del vector de futuros mencionado anteriormente

```
// calculo de la integral de forma concurrente
double calcular_integral_concurrente( )
{
   future<double> futuros[n];    // Creación de un vector de tantos futuros como hebras
   double resultado = 0.0;       // Inicialización del resultado

   if(ciclica)
      for( int i = 0; i < n ; i++ )
         futuros[i] = async( launch::async, funcion_hebra_ciclica, i );
        // Cada hebra ejecuta funcion_hebra_ciclica

   else
      for( int i = 0; i < n ; i++ )
         futuros[i] = async( launch::async, funcion_hebra_bloques, i );
         // Cada hebra ejecuta funcion_hebra_bloques

   for( int i = 0; i < n ; i++ )
       resultado += futuros[i].get();
       // Cuando cada hebra haya acabado, se suma el resultado de la ejecución

   return resultado;                          // Se devuelve la suma calculada
}
```

Cada hebra calcula el valor de la integral en unas determinadas zonas del intervalo $[0,1]$, dividido en `m` subintervalos iguales de longitud `1/m`. La forma de asignar qué intervalos trata cada hebra se recoge en las siguientes funciones:

Asignación cíclica

Supongamos `n=4` y `m=8`. Si usamos asignación cíclica, cada hebra ejecutaría el cálculo del área sobre los intervalos tal que:

- La hebra 0 calcularía el área sobre los intervalos $[0, \frac{1}{8})$ y $[\frac{4}{8}, \frac{5}{8})$.
- La hebra 1 calcularía el área sobre los intervalos $[\frac{1}{8}, \frac{2}{8})$ y $[\frac{5}{8}, \frac{6}{8})$.
- La hebra 2 calcularía el área sobre los intervalos $[\frac{2}{8}, \frac{3}{8})$ y $[\frac{6}{8}, \frac{7}{8})$.
- La hebra 3 calcularía el área sobre los intervalos $[\frac{3}{8}, \frac{4}{8})$ y $[\frac{7}{8}, 1]$.

Generalizando, la hebra i calcularía el área sobre los intervalos $[\frac{i}{m}, \frac{i+1}{m}], [\frac{i+n}{m}, \frac{i+n+1}{m}], [\frac{i+2n}{m}, \frac{i+2n+1}{m}], \ldots$ mientras el extremo inferior no pasara de 1. Siguiendo este enfoque y teniendo en cuenta que el extremo inferior del intervalo en cada iteración aumenta $\frac{n}{m}$ la implementación de esta función es:

```
double funcion_hebra_ciclica( long i )
{
   double suma_parcial = 0.0;                              // Inicializar suma

   for( double x = (double)i/m; x < 1.0 ; x += (double)n/m )   // Asignación cíclica
      suma_parcial += f(x)/m;          // Añadir f(x)*(1/m)=f(x)/m a la suma parcial

   return suma_parcial;                                    // Devolver suma parcial
}
```

Asignación con bloques contiguos

De nuevo supongamos `n=4` y `m=8`. Si usamos asignación por bloques contiguos, cada hebra ejecutaría el cálculo del área sobre los intervalos tal que:

- La hebra 0 calcularía el área sobre los intervalos $[0, \frac{1}{8})$ y $[\frac{1}{8}, \frac{2}{8})$.
- La hebra 1 calcularía el área sobre los intervalos $[\frac{2}{8}, \frac{3}{8})$ y $[\frac{3}{8}, \frac{4}{8})$.
- La hebra 2 calcularía el área sobre los intervalos $[\frac{4}{8}, \frac{5}{8})$ y $[\frac{5}{8}, \frac{6}{8})$.
- La hebra 3 calcularía el área sobre los intervalos $[\frac{6}{8}, \frac{7}{8})$ y $[\frac{7}{8}, 1]$.

Generalizando, la hebra i calcularía el área sobre los intervalos $[\frac{i}{n}, \frac{i}{n} + \frac{1}{m}], [\frac{i}{n} + \frac{1}{m}, \frac{i}{n} + \frac{2}{m}]$, $[\frac{i}{n} + \frac{2}{m}, \frac{i}{n} + \frac{3}{m}], \ldots$ mientras el extremo inferior no pase de $\frac{i+1}{n}$ (que le correspondería a otra hebra). Siguiendo este enfoque, la implementación sería la siguiente:

```
double funcion_hebra_bloques( long i )
{
   double suma_parcial = 0.0;                              // Inicializar suma

   for( double x = (double)i/n; x < (double)(i+1)/n; x += 1.0/m )
   // Asignación por bloques contiguos
      suma_parcial += f(x)/m;           // Añadir f(x)*(1/m)=f(x)/m a la suma parcial

   return suma_parcial;                                    // Devolver suma parcial
}
```

Resultados

Para probar el programa, a continuación presento un ejemplo de ejecución con $n = 4$ hebras y $m = 1024 \cdot 1024 \cdot 1024$ subintervalos. Lo esperado es que tanto en una asignación con bloques contiguos como cíclica la versión concurrente tarde en torno a un 25% de lo que tarda la versión secuencial en ejecutar el cálculo. Veamos:

```
~$ g++ -std=c++11 ejemplo9.cpp -o ejemplo9 -lpthread
~$ ./ejemplo9 -c
               Asignación cíclica:
Número de muestras (m)   : 1073741824
Número de hebras (n)     : 4
Valor de PI              : 3.14159265358979312
Resultado secuencial     : 3.14159265358998185
Resultado concurrente    : 3.14159265452106196
Tiempo secuencial        : 19436 milisegundos.
Tiempo concurrente       : 4422.1 milisegundos.
Porcentaje t.conc/t.sec. : 22.75%
~$ ./ejemplo9 -b
        Asignación por bloques contiguos:
Número de muestras (m)   : 1073741824
Número de hebras (n)     : 4
Valor de PI              : 3.14159265358979312
Resultado secuencial     : 3.14159265358998185
Resultado concurrente    : 3.14159265452114944
Tiempo secuencial        : 19975 milisegundos.
Tiempo concurrente       : 6163.6 milisegundos.
Porcentaje t.conc/t.sec. : 30.86%
```

Como se puede ver, en ambas versiones el cálculo del valor de π supone una muy buena aproximación y el porcentaje entre concurrente y secuencial es muy cercano al 25%, lo que quiere decir que la función concurrente tarda en ejecutar en torno a la cuarta parte que la secuencial.