Actividad de programación Multiprocesadores:

Calculando Pi con Threading Building Blocks (TBB)

José Ramón Romero Chávez (A01700318)
Tecnológico de Monterrey, Campus Querétaro
A01700318@itesm.mx

28 de octubre de 2018

Resumen

Este documento se explicara de manera puntual la implementación del algoritmo de cálculo de PI usando el método de integración haciendo uso de Paralelismo y su contraparte serial en C++ haciendo uso de la librería Threading Building Blocks, que posteriormente serán analizados y evaluados con respecto a su velocidad.

1. Introducción

π (**pi**) es la relación entre la longitud de una circunferencia y su diámetro en geometría euclidiana. Es un número irracional y una de las constantes matemáticas más importantes. Se emplea frecuentemente en matemáticas, física e ingeniería.

El método de cálculo por integración de pi, está basado en la siguiente fórmula:

Definite integral:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi \approx 3.1416$$

Para calcular una aproximacion del area debajo de la curva de la función (Figura 1) , es necesario calcular el área de los rectángulos que conforman dicha curva, entre mayor sea la cantidad de rectángulos, la diferencia será menor y nos podrá otorgar un resultado más acertado (Figura 2)

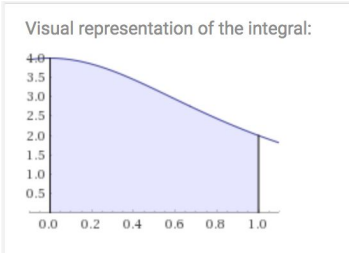

Figura 1

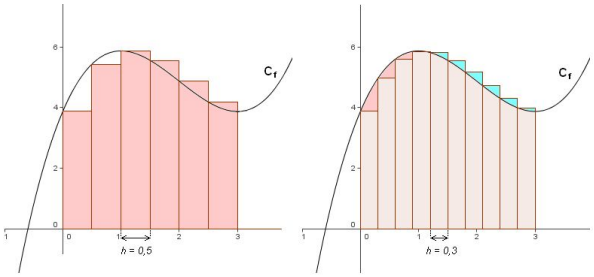

Figura 2

Ahora bien, dado este algoritmo, es posible realizarse de manera serial o paralela, para ello haremos uso de la clase Threading Building Blocks existente en C++ para posteriormente explicar y comparar ambas implementaciones con fin de exponer los puntos importantes de cada uno con respecto a su contraparte.

2. Implementación

La implementación fue realizada en una computadora con las siguientes características:

- MacBook Pro (Retina, 13-inch, Early 2015)
  - macOS Mojave 10.14 (18A391)
  - Processor Name: Intel Core i5 o Processor Speed: 2.7 GHz
  - Number of Processors: 1
  - Total Number of Cores: 2 with hyperthreading
  - L2 Cache (per Core): 256 KB
  - L3 Cache: 3 MB
  - Memory: 8 GB 1867 MHz DDR3
  - Apple LLVM version 10.0.0 (clang-1000.11.45.2)
  - Target: x86_64-apple-darwin18.0.0
  - Thread model: posix

Versión Single-Thread

En esta primera implementación vemos todo el cálculo implementado en una sola clase SinglePi.

El algoritmo va de lo siguiente, una vez determinado el número de rectángulos en los cuales de planea calcular el área debajo de la curva, es necesario calcular el ancho de los rectángulos con respecto al intervalo de integración [0,1] y el número de los mismos, y una vez con ello evaluar con respecto a la función para encontrar la altura que iremos sumando y al final usaremos para encontrar el área.

Es importante recalcar que la parte fundamental, es el ciclo iterativo que va desde el cero hasta el número de rectángulos deseado.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>
#include "utils/cppheader.h"

using namespace std;
using namespace tbb;

const int SIZE = 100000000;
const int GRAIN = 100000;

class SinglePi {
public:
     int size;
```

```
    double result;

public:
    SinglePi(int s) : size(s){}

    long getResult() const {
        return result;
    }

    void calculate() {
        double mid, height,width;
        double sum  = 0;
        width = 1.0/size;

        for (double i = 0; i < size ; i++) {
                mid=(i+.5)*width;
                height = 4.0 / (1.0 + mid* mid);
                double a=sum;
                sum+= height;
        }

        result = width * sum;

    }
};
```

```
int main(int argc, char* argv[]) {
     Timer t;
     double ms;
     long result;
     int *a = new int[SIZE];
          SinglePi sa(SIZE);
          cout << "Starting..." << endl;
          ms = 0;
          for (int i = 0; i < N; i++) {
               t.start();
               sa.calculate();
               ms += t.stop();
          }
          cout << "\nPi: " << sa.result << " size: "<< SIZE
<< " Times "<< N << endl;
          cout << "avg time = " << (ms/N) << " ms\n" << endl;



          return 0;
     }
```

Versión Multi-Thread

En la versión multi-thread existen los siguientes puntos claves dentro de la implementación:

El algoritmo sigue siendo el mismo, sin embargo, es posible implementar la técnica de divide y conquista dentro de la iteración principal [0, SIZE], permitiendo segmentar dicho rango..

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <tbb/task_scheduler_init.h>
#include <tbb/parallel_for.h>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>
#include "utils/cppheader.h"

using namespace std;
using namespace tbb;

const int SIZE = 100000000;
const int GRAIN =  10*1000;

class ParallelPi {

public:
     double size;
     double suma;

     ParallelPi(double s) : suma(0.0),  size(s) {}
```

```
     ParallelPi(ParallelPi &x, split)
          : suma(0.0), size(x.size) {}

     void operator() (const blocked_range<int> &r) {
    double mid, height;
          for (int i = r.begin(); i < r.end(); i++) {
               mid = (double)(i + 0.5)* size;
      height = 4.0 / (1.0 + (mid * mid));
      suma += height;

          }
     }

     void join(const ParallelPi &x) {
          suma += x.suma;
     }
};

int main(int argc, char* argv[]) {
     Timer t;
     double ms;
     double result;

     cout << "Starting..." << endl;
     ms = 0;
  double w = 1.0 /(double) SIZE;
```

```
     for (int i = 0; i < N; i++) {
          t.start();
          ParallelPi obj(w);
          parallel_reduce(         blocked_range<int>(0,SIZE,
GRAIN),obj );
          result = obj.suma;
          result = result*w;
          ms += t.stop();
     }
     cout << "\nPi: " << result << " size: "<< SIZE << "
Times "<< N << endl;
     cout << "avg time = " << (ms/N) << " ms\n" << endl;


     return 0;
}
```

3. Resultados.

MacBook-Pro-de-Ramon-2:pi-tbb ramonromero$ g++ singlePiTBB.cpp -o single.out

MacBook-Pro-de-Ramon-2:pi-tbb ramonromero$ g++ -ltbb multiPiTBB.cpp -o multi.out

MacBook-Pro-de-Ramon-2:pi-tbb ramonromero$ ./single.out

Starting...

Pi: 3.14159 size: 100000000 Times 10

avg time = 457.504 ms

MacBook-Pro-de-Ramon-2:pi-tbb ramonromero$ ./multi.out

Starting...

Pi: 3.14159 size: 100000000 Times 10

avg time = 273.718 ms

La evaluación promedio de 10 ejecuciones del caso arrojó el siguiente speedup:

$S = \frac{T_s}{T_m}$ $S = \frac{457.504\ s}{273.718\ s}$ $S = 1.671$

Es evidente la mejora que existe, ya que es fácil observar como la ejecución de la tarea paralelizada logra compensar su tiempo de creación. *Sin embargo, siempre hay que tener en cuenta la factibilidad de una mejora tomando en consideración aspectos como la cantidad de cores de nuestro equipo, además de los tipos y la longitud de los datos que ocupamos para iterar.*

3. Referencias

[1]"Pi", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Pi. [Accessed: 17- Sep- 2018].

[2] [BRESHEARS] pp. 31 & 32.

[3]"Amdahl's law", En.wikipedia.org, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed: 15- Sep- 2018].

[4]"Computing Pi in Parallel with Java", Science.smith.edu, 2018. [Online]. Available: http://www.science.smith.edu/dftwiki/index.php/CSC352:_Computing_Pi_in_Parallel_with_Java. [Accessed: 15- Sep- 2018].

[5]"Wolfram|Alpha", M.wolframalpha.com, 2018. [Online]. Available: http://m.wolframalpha.com/. [Accessed: 15- Sep- 2018].