Aproximación de PI mediante la Serie de Euler

Guillermo Fidel Navarro Vega (A01274191)

Tecnológico de Monterrey, Campus Querétaro

A01274191@tec.mx

1. Resumen:

Este documento, como su título indica, tiene el objetivo de desarrollar una solución al calculo de PI, mediante series infinitas de Euler. Esta solución planteada ha sido programada en 3 lenguajes de programación mediante técnicas secuenciales y paralelas, con el fin de analizar el efecto en el rendimiento que tienen los paradigmas concurrentes que se han desarrollado durante el curso. Las tecnologías que se utilizaron durante la realización de este documento son las siguientes:

- Java Threads
- Fork Join (Java)
- OpenMP (C)
- Intel Threading Building Blocks (C++)

Durante la realización de este documento se comparo el speedup de cada tecnología con respecto a su implementación en el mismo lenguaje, pero de manera secuencial. En base a los resultados obtenidos se determinó que la implementación de concurrencia presenta una gran ventaja, ya que se obtuvo una gran reducción de tiempo de ejecución con un speedup promedio obtenido de 7.45 entre todas las tecnologías utilizadas, siempre y cuando el numero de iteraciones supere 4,000,000 ya que antes de esto, el calculo secuencial produce un resultado con mayor velocidad. Aunque, realmente para obtener un resultado fiable de PI se necesitan utilizar más iteraciones que las mencionadas anteriormente. En conclusión, con este pequeño resumen, se puede determinar que la implementación de concurrencia presenta una gran oportunidad para mejorar el rendimiento de este tipo de cálculos.

2. Introducción

El número Pi es una constante importante en el mundo de la ingeniería ya que su uso es trascendente para un gran número de formulas que describen desde el comportamiento del área de un circulo hasta el de la vibración de hilos. Esto significa que esta constante ha sido objeto de investigación por parte de muchos matemáticos a lo largo de los siglos, además de que muchos problemas matemáticos relacionados a las series infinitas tienen como resultado algún derivado de Pi de alguna forma u otra.

En este documento se usará una de las series infinitas de Euler con el fin de aproximar el valor de Pi de manera iterativa, específicamente se utilizará el problema de Basilea para obtener el valor de Pi. El problema de Basilea fue planteado por Pietro Mengoli en 1650, donde se busca encontrar la suma de los inversos de los cuadrados de los enteros positivos, la definición de la sumatoria es la siguiente:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = (\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \cdots + \frac{1}{n^2})$$

Este problema se mantuvo sin resolver durante 84 años, ya que solo se sabia el valor aproximado de la sumatoria con un valor de 1.64493. Hasta 1734 fue resuelto por Leonhard Euler que encontró la solución exacta a esta sumatoria utilizando las series de Taylor. Euler argumento que las propiedades polinómicas de series finitas aplicaban para series infinitas por lo que llego al siguiente resultado:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \left(\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots + \frac{1}{n^2}\right) = \frac{\pi^2}{6}$$

Este resultado se puede despejar para obtener Pi, de la siguiente manera:

$$\pi = \sqrt{6\left(\sum_{n=1}^{\infty} \frac{1}{n^2}\right)}$$

Como se puede observar, esta seria nos será de gran ayuda para calcular una aproximación precisa del valor de Pi, aunque se necesitará una gran cantidad de iteraciones para obtener un valor mas fidedigno. Por consiguiente, se demostrará el desempeño del algoritmo en las diferentes tecnologías que se vieron durante el curso de Multiprocesadores. Las tecnologías que se utilizaran son las siguientes:

- Java Threads
- Fork Join (Java)
- OpenMP (C)
- Intel Threading Building Blocks (C++)\

Cada una de estas tecnologías será comparada con su equivalente secuencial para demostrar los beneficios que tiene la implementación de la concurrencia en la optimización y aceleración de la obtención de un resultado por parte del algoritmo.

3. Desarrollo

- **Implementación secuencial utilizando Java**

Como se mencionó anteriormente, el algoritmo se implemento de manera secuencial inicialmente, con el fin de obtener una medida base de tiempo para poder compararla con su versión concurrente. El primer paso fue generar la operación que representara la suma de las fracciones. Este fragmento de código representa la suma de $\frac{1}{n^2}$ dentro de un ciclo for.

```
public void calculate() {
          acum = 0;
          // place your code here
          for(int i = 1; i < MAXIMUM; i++){
               acum += 1.0/((double)i*(double)i);
          }
}
```

La variable "MAXIMUM" representa el número máximo de iteraciones que se realizaran, en este caso se llegara hasta 1,000,000,000 para así obtener un valor mas aproximado a Pi y generar una gran carga en el CPU, antes de imprimir el resultado se aplica la operación de raíz cuadrada y multiplicación según la formula obtenida en la introducción, para obtener el valor de Pi.

```
public double getResult() {
          return Math.sqrt(6*acum);
     }
```

El código completo que se utilizo para realizar esta prueba se encuentra en el apéndice *1*, al ser un código secuencial su implementación es bastante trivial, ya que solo se depende de una simple operación matemática.

La velocidad de ejecución obtenida es de 3282.5 ms.

- **Implementación del algoritmo utilizando Java Threads**

Una vez que se obtuvo un tiempo base de manera secuencial se generó el código concurrente utilizando Threads de Java. La operación principal del código se definió de la siguiente manera:

```
public void run() {
          // place yout code here
          acum = 0;
          // place your code here
          for(int i = start; i < end; i++){
               acum += 1.0/((double)i*(double)i);
          }
     }
```

Como se puede observar el único cambio en la función es la utilización de la variable start y end como inicio y fin de i, en lugar de 1 y el valor máximo como se definió en el código secuencial, esto se debe a que al separar los cálculos en threads, se definen rangos de operación para cada uno de los hilos dedicados al cálculo de la sumatoria. De igual manera, una vez terminado el calculo se aplica la operación de raíz cuadrada y multiplicación por 6 que se definió en la introducción para obtener el valor de Pi, además el código completo de esta implementación se encuentra en el apéndice *2*.

Esta implementación obtuvo una velocidad de ejecución de 416.2 ms.

- **Implementación del algoritmo utilizando Fork Join de Java**

La implementación de este algoritmo sigue en concurrencia, pero utiliza un método recursivo para separar las operaciones en Threads con rangos efectivos cada vez más pequeños, llegando hasta un valor mínimo donde se detiene la separación y se ejecuta la operación principal, la que sigue siendo en esencia la misma.

```
public Double computeDirectly(){
          // place yout code here
          acum = 0;
          // place your code here
          for(int i = start; i < end; i++){
               acum += 1.0/((double)i*(double)i);
          }
        return acum;
     }
```

La parte clave del código de Fork Join es la separación de los rangos de manera recursiva, la que se puede ver en el siguiente fragmento de código:

```
@Override
     protected Double compute(){
          if((end - start) <= MIN){
               return computeDirectly();
          }else {
               int mid = start + ((end - start)/2);
               ForkJoin lowerMid = new ForkJoin(start, mid);
               lowerMid.fork();
               ForkJoin upperMid = new ForkJoin(mid, end);
               return upperMid.compute() + lowerMid.join();
          }
     }
```

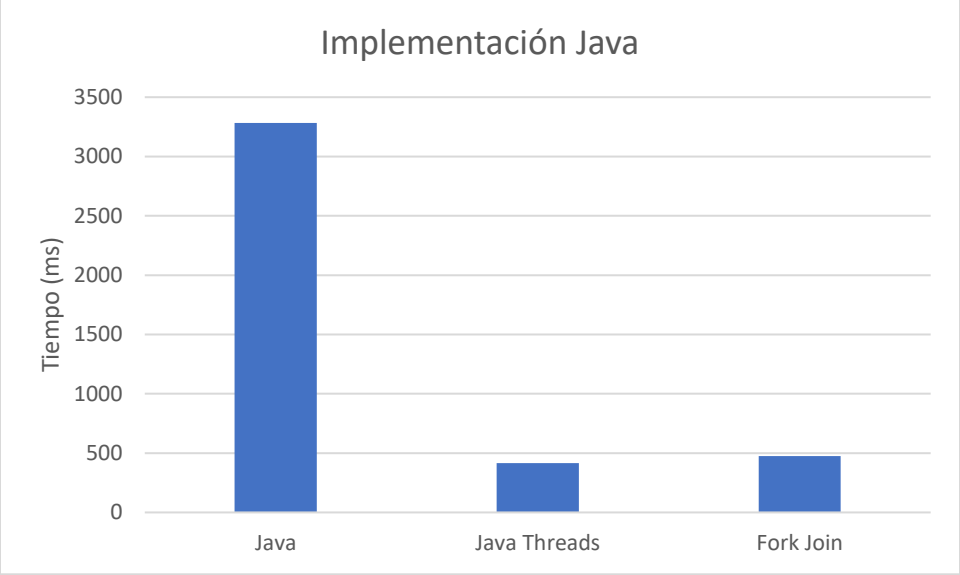
Este código representa la separación del rango de números en threads cada vez mas pequeñas, hasta que se llega a la operación mínima de 1000, donde se detiene y ejecuta la operación que se propuso en el algoritmo. De igual forma se tiene que realizar el paso de raíz cuadrada y multiplicación por 6 para llegar a Pi. Similarmente a los códigos anteriores, este se podrá encontrar en el apéndice *3.*

El resultado de tiempo de ejecución fue de 474.6 ms

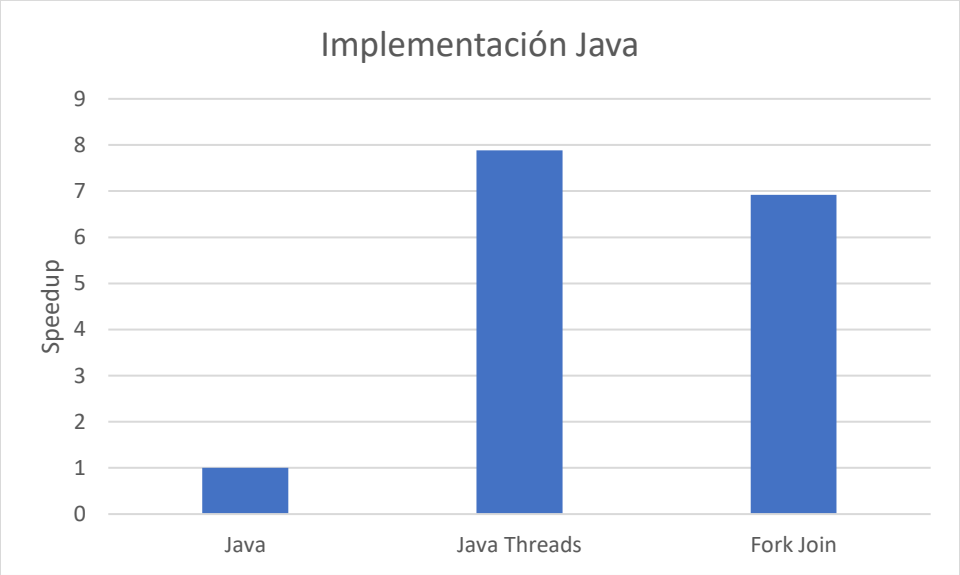
- **Comparación de implementaciones en Java**

Una vez que se obtuvieron los tiempos de ejecución utilizando tecnología secuencial y concurrente se realizó la comparación de tiempos de ejecución de las tecnologías basadas en Java. En base a los resultados se puede llegar a la conclusión que la inclusión de concurrencia en este algoritmo se considera como altamente beneficiosa para su desempeño. Se logro llegar a un speedup con Threads de 7.88 y uno de 6.91 con Fork Join. En la siguiente tabla se puede observar el comparativo en tiempo de ejecución de las 3 implementaciones.

Esta grafica contiene la comparación en términos de tiempo de cada implementación, se observa que la diferencia entre el código secuencial y concurrente es enorme, mientras que entre las tecnologías concurrentes es casi nula debido a que aprovechan los núcleos del sistema



Esta grafica contiene la representación de la mejora de rendimiento en términos de speedup, es decir, en base a producto de la división del tiempo secuencial sobre el tiempo concurrente.



En base a lo descrito anteriormente se observa una aceleración de procesamiento favorable.

- **Implementación del algoritmo utilizando C**

De igual forma se implementó una solución al algoritmo utilizando C de manera secuencial, debido a las similitudes que se tienen con la implementación de Java no se adentrara mucho en la explicación de código. Esta es la función generada en C:

```
double CalcPi(){
     double acum = 0;
     for(int i = 1; i < MAXIMUM; i++){
        acum += 1.0/((double)i*(double)i);
    }
    acum = sqrt(6*acum);
    return acum;
}
```

Similarmente al MAXIMUM de java se utilizaron 1,000,000,000 de iteraciones para el cálculo.

El código completo se encuentra en el apéndice *4.*

La velocidad de ejecución obtenida en C fue de 6220.7178 ms.

- **Implementación de algoritmo utilizando OpenMP (C)**

Para la implementación de concurrencia en C se utilizó OpenMP, esta tecnología se basa en establecer zonas de concurrencia con el fin de optimizar ciertas partes del código como ciclos, etc. Para implementar una zona de concurrencia únicamente se tiene que declarar una directiva con las variables compartidas y las privadas con el fin de que el compilador sepa como crear el código concurrente. El código de la función se optimizó añadiendo las directivas “omp parallel”.

```
double CalcPi(int n){
    // place yout code here
    double acum = 0;
    // place your code here
    #pragma omp parallel for shared(n) reduction(+:acum)
    for(int i = 1; i < n; i++){
        acum += 1.0/((double)i*(double)i);
    }
    return acum;
}
```

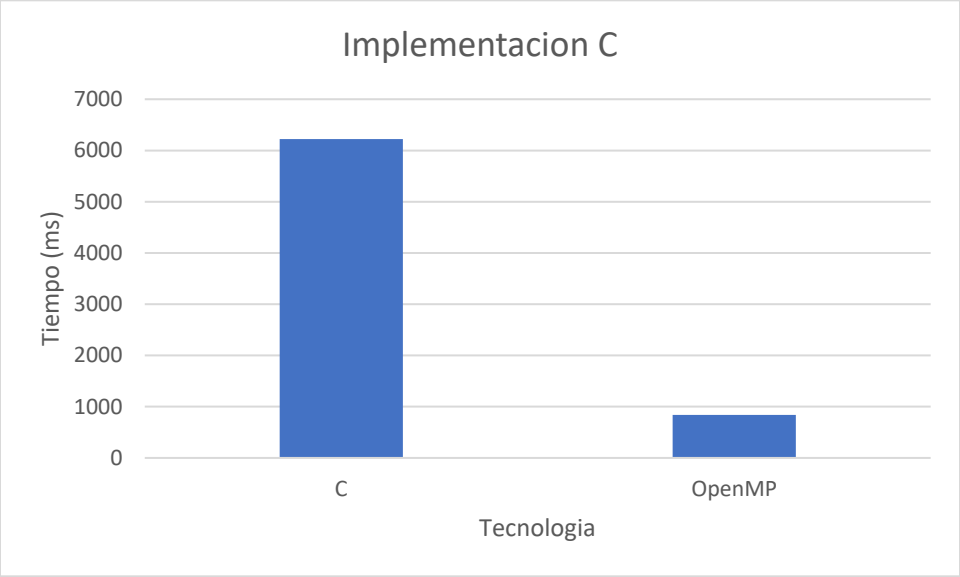
El código completo se encuentra en el apéndice *5*.

El tiempo de ejecución de esta implementación es de 836.3419 ms,

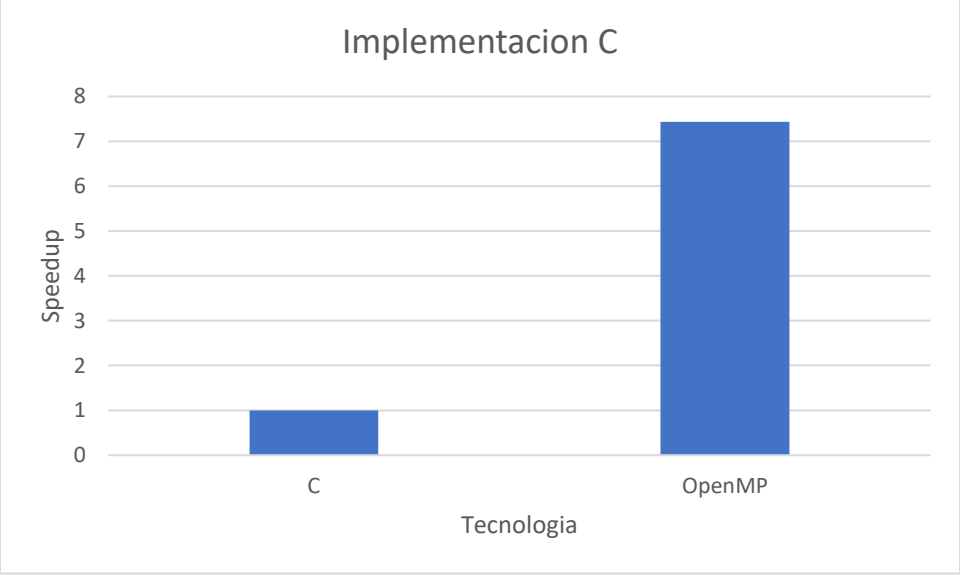
- **Comparación de resultados C**

Al realizar la comparación entre ambos paradigmas se observa como la concurrencia genera ganancias de tiempo altísimas, aunque es el mismo algoritmo, la capacidad de usar todos los hilos del procesador nos da como resultado un tiempo ejecución mucho menor a su equivalente secuencial.

Grafica de tiempo de ejecución:



Otra manera de ver el aumento de eficiencia en tiempo de ejecución es mediante el speedup, donde se obtuvo uno de 7.43. En la siguiente grafica se puede ver el comportamiento del mismo.



- **implementación del algoritmo utilizando C++**

Como ultimo lenguaje de programación para implementar este algoritmo se eligió C++ para fines comparativos con TBB, ya que se basa en este lenguaje, realmente la implementación es muy similar a la de C por lo que no se añadirá el fragmento de código de la función que describe el algoritmo, el código completo en el apéndice *6.*

El numero de iteraciones que se utilizo en este lenguaje sigue siendo el mismo de 1,000,000,000.

El tiempo de ejecución para este código es de 6220.9943 ms, prácticamente el mismo a la implementación en C.

- **implementación del algoritmo utilizando Intel Threading Building Blocks en C++**

Para utilizar esta tecnología se tuvo que modificar la clase que se genero anteriormente con el nombre de los métodos definidos por TBB. Se llego al siguiente codigo:

```
class EulerPi{
     private:
          int max;
        double acum = 0;
     public:
          EulerPi(int s){max = s; acum = 0;}
        EulerPi(EulerPi &x, split) : max(x.max), aum(0) {}
          void operator() (const blocked_range<int> &r) {
            for(int i = r.begin(); i < r.end(); i++){
                 acum += 1.0/((double)i*(double)i);
            }
         }
        void join(const EulerPi &x){
             acum += x.acum;
        }
        double getResult() const {
             return result;
        }

};
```

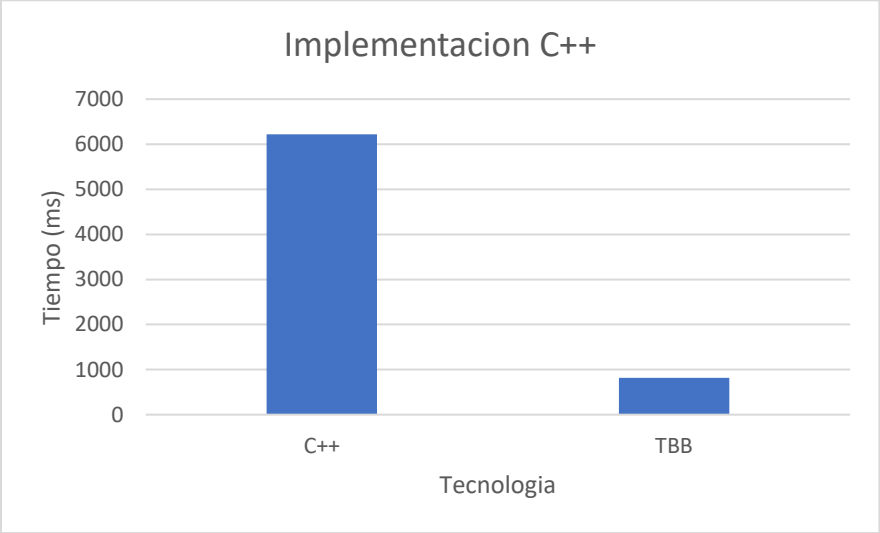
Estos métodos que se utilizaron son los requeridos por TBB para poder generar código concurrente utilizando el método “parallel_reduce”. El codigo completo de esta implementación se encuentra en el apéndice *7.*

El tiempo total de ejecución al utilizar TBB es de 819.56 ms.

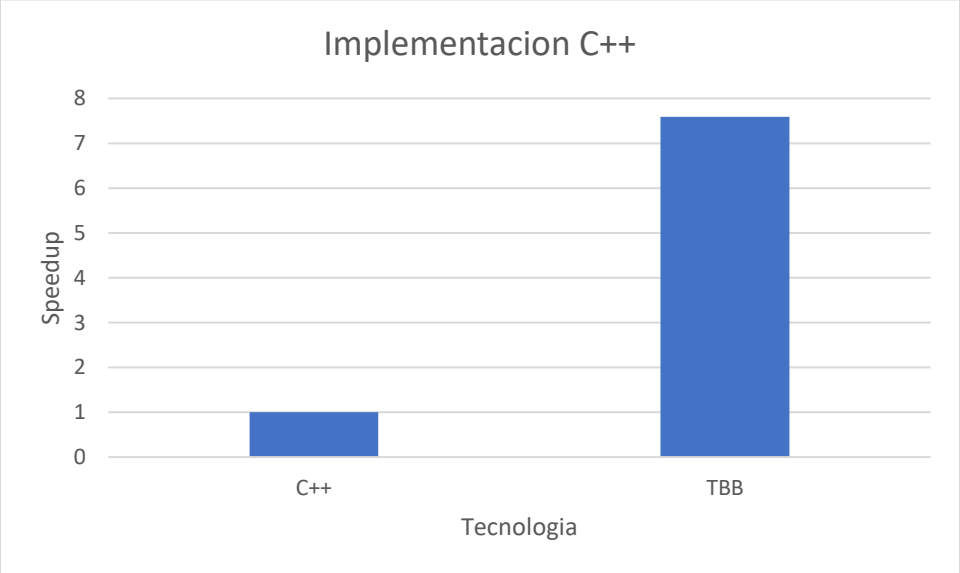
- **Comparación de resultados C++**

Nuevamente la implementación de tecnologías concurrentes en el lenguaje de C++ nos muestra la gran ventaja que tienen estos paradigmas de programación sobre estas tareas iterativas.

Se observa la reducción tiempo de ejecución de 6220.99 a 819.56 ms, lo que considero es un gran logro.



Al ver los resultados en la forma de speedup se alcanza un valor de 7.5906, lo que nuevamente nos da un buen indicativo del desempeño de la concurrencia en este caso.



4. **Conclusiones**

La implementación de este algoritmo tanto de manera secuencial como concurrente me ayudo a dimensionar aun mas las ventajas potenciales que se pueden tener al realizar este tipo de optimizaciones, ya que conforme el tiempo avanza la capacidad de procesamiento en un solo hilo de los procesadores ha dejado de aumentar tan drásticamente, pero la cantidad de núcleos presentes en un CPU solo ha aumentado, por lo que saber optimizar el uso de estos procesadores extra es algo indispensable en el desarrollo de nuevo software. Aun cuando la concurrencia parece la solución a todos nuestros problemas tenemos que saber utilizarla, ya que tenemos que evaluar el algoritmo a utilizar, en este caso la suma no depende de resultados anteriores por lo que es fácilmente separable. Además de que se debe tomar en cuenta el tiempo de procesamiento de cada iteración y la cantidad de estas, ya que hay momentos donde el overhead de separar la información en threads es mucho mayor al tiempo de calculo y el desempeño empeora en comparación del código secuencial.

Para el cálculo de Pi me parece indispensable utilizar concurrencia ya que en promedio con las tecnologías se obtuvo un speedup de 7.45. Java threads fue la tecnología que obtuvo el mejor aumento de desempeño sorprendentemente, ya que yo esperaba que C y OpenMP fueran los más veloces. En general considero que la elección de tecnología concurrente queda a discreción del usuario ya que para este tipo de algoritmos siempre habrá mejoras de desempeño sin importar la tecnología escogida. Realmente considero que este curso me ha ayudado a sentar las bases para que pueda analizar y generar código concurrente de calidad.

5. Referencias

- Grabinksy, G. (2007). *Euler, El Prestidigitador de las Series.* Recuperado de Miscelánea Matemática, sitio web: https://miscelaneamatematica.org/download/tbl_articulos.pdf2.8be6d003eae97820.4775696c6c65726d6f5f672e706466.pdf
- Britannica, (2022). *Pi.* Recuperado de Britannica, sitio web: https://www.britannica.com/science/pi-mathematics
- Sullivan, B. (2013, abril). *The Basel Problem.* Recuperado de Carnegie Mellon University, sitio web: https://www.math.cmu.edu/~bwsulliv/basel-problem.pdf
- Alvy, (2007, noviembre). *Series Infinitas de Euler.* Recuperado de microsiervos, sitio web: https://www.microsiervos.com/archivo/ciencia/series-infinitas.html
- Sanchez, J, M. (2014). *El Problema de Basilea.* Recuperado de Dialnet, sitio web: https://dialnet.unirioja.es/descarga/articulo/7177416.pdf

6. Apéndices

1. Código Java

```
/*-----------------------------------------------------------------
--

*

* Multiprocesadores: Euler Java

* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*

*-----------------------------------------------------------------
*/

public class Euler {
     private static final int MAXIMUM = 1_000_000_000;
     double acum;
     public Euler() {
          this.acum = 0;
     }

     public double getResult() {
          return Math.sqrt(6*acum);
     }
```

```
    public void calculate() {
        acum = 0;
        // place your code here
        for(int i = 1; i < MAXIMUM; i++){
            acum += 1.0/((double)i*(double)i);
        }
    }

    public static void main(String args[]){

        long startTime, stopTime;
        double acum = 0;
        Euler  e = new Euler();
        acum = 0;
        for (int i = 0; i < Utils.N; i++) {
            startTime = System.currentTimeMillis();

            e.calculate();

            stopTime = System.currentTimeMillis();

            acum += (stopTime - startTime);
        }
        System.out.printf("sum = %f\n", e.getResult());
        System.out.printf("avg  time  =  %.5f  ms\n",  (acum  /
Utils.N));
    }
}
```

2. Código Java Threads

```
/*-------------------------------------------------------------
--

*

* Multiprocesadores: Euler Java Threads

* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*

*-------------------------------------------------------------
*/


public class EulerThreads extends Thread {
     private static final int MAXIMUM = 1_000_000_000;
     private double acum;
     private int start;
     private int end;
     public EulerThreads(int start, int end) {
          this.start = start;
          this.end = end;
          this.acum = 0;
     }


     public void run() {
```

```
            // place yout code here
            acum = 0;
            // place your code here
            for(int i = start; i < end; i++){
                  acum += 1.0/((double)i*(double)i);
            }

      }

      public double getResult() {
            return acum;
      }

      public static void main(String args[]) {

            long startTime, stopTime;
            double acum = 0, ms, result;
            int block;
            EulerThreads threads[];
            block = MAXIMUM / Utils.MAXTHREADS;
            threads = new EulerThreads[Utils.MAXTHREADS];



            System.out.println("");
            System.out.println(block);
            System.out.printf("Starting   with   %d   threads...\n",
Utils.MAXTHREADS);
            ms = 0;
            for (int j = 1; j <= Utils.N; j++) {
                  for (int i = 0; i < threads.length; i++) {
```

```
                    if (i != threads.length - 1) {
                         threads[i]            =            new
EulerThreads((i*block)+1,((i + 1) * block));
                    } else {
                         threads[i] = new EulerThreads((i*block),
MAXIMUM+1);
                    }
               }

               startTime = System.currentTimeMillis();
               for (int i = 0; i < threads.length; i++) {
                    threads[i].start();
               }
               for (int i = 0; i < threads.length; i++) {
                    try {
                         threads[i].join();
                    } catch (InterruptedException e) {
                         e.printStackTrace();
                    }
               }
               stopTime = System.currentTimeMillis();
               ms +=  (stopTime - startTime);

          }
          result = 0;
          for (int i = 0; i < threads.length; i++) {
               result += threads[i].getResult();
          }
          result = Math.sqrt(6*result);
          System.out.printf("result = %.5f\n", result);
          System.out.printf("avg time = %.5f\n", (ms / Utils.N));
```

```
    }
}
```

3. Código Java Fork Join

```
/*------------------------------------------------------------------
--

*

* Multiprocesadores: Euler Java Threads

* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*

*------------------------------------------------------------------
*/



import java.util.Arrays;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class ForkJoin extends RecursiveTask<Double> {
    private static final int MAXIMUM = 1_000_000_000;
    private static final int MIN = 1_000;
    private double acum = 0;
    private int start;
    private int end;
```

```
    public ForkJoin(int start, int end){
         this.start = start;
         this.end = end;
    }

    public Double computeDirectly(){
         // place yout code here
         acum = 0;
         // place your code here
         for(int i = start; i < end; i++){
              acum += 1.0/((double)i*(double)i);
         }
       return acum;
    }

    @Override
    protected Double compute(){
         if((end - start) <= MIN){
              return computeDirectly();
         }else {
              int mid = start + ((end - start)/2);
              ForkJoin lowerMid = new ForkJoin(start, mid);
              lowerMid.fork();
              ForkJoin upperMid = new ForkJoin(mid, end);
              return upperMid.compute() + lowerMid.join();
         }
    }
// place your code here
```

```
  public static void main(String args[]){

          long startTime, stopTime;
          double acum = 0, ms = 0;
          double result = 0;
          ForkJoinPool pool;

     for (int i = 0; i < Utils.N; i++) {
          startTime = System.currentTimeMillis();

          pool = new ForkJoinPool(Utils.MAXTHREADS);
          result = Math.sqrt(6*pool.invoke(new ForkJoin( 1,
MAXIMUM)));

          stopTime = System.currentTimeMillis();
          ms += (stopTime - startTime);
     }
     System.out.printf("result = %f\n", result);
     System.out.printf("avg time = %.5f\n", (ms / Utils.N));
}
}
```

4. Código C

```
/*-----------------------------------------------------------------
--

*

* Multiprocesadores: Euler C

* Fecha: 27-Noviembre-2022
```

```
* Autor: Guillermo Fidel Navarro Vega A01274191

*

*------------------------------------------------------------------
*/



#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include <math.h>

#define MAXIMUM 1000000000//5e6

// implement your code

double CalcPi(){
     double acum = 0;
     for(int i = 1; i < MAXIMUM; i++){
        acum += 1.0/((double)i*(double)i);
    }
    acum = sqrt(6*acum);
    return acum;
}



int main(int argc, char* argv[]) {
     int i, *a;
     double ms;
```

```
    double result;
    printf("Starting...\n");
    ms = 0;
    for (i = 0; i < N; i++) {
        start_timer();

        // call the implemented function
        result = CalcPi();
        ms += stop_timer();
    }
    printf("result = %.4f\n", result);
    printf("avg time = %.5lf ms\n", (ms / N));

    free(a);
    return 0;
}
```

5. Código OpenMP C

```
/*-----------------------------------------------------------------
--

*

* Multiprocesadores: Euler C OpenMP

* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*
```

```
*-----------------------------------------------------------
*/



#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "utils.h"

#define MAXIMUM 1000000000 //1e9

double CalcPi(int n){
    // place yout code here
    double acum = 0;
    // place your code here
    #pragma omp parallel for shared(n) reduction(+:acum)
    for(int i = 1; i < n; i++){
        acum += 1.0/((double)i*(double)i);
    }
    return acum;
}

int main(int argc, char* argv[]) {
     int i;
     double ms;
     double result;
     printf("Starting...\n");
     ms = 0;
```

```
     for (i = 0; i < N; i++) {
          start_timer();

          // call the implemented function
          result = CalcPi(MAXIMUM);
          ms += stop_timer();
     }
    result = sqrt(result*6);
     printf("result = %.10f\n", result);
     printf("avg time = %.5lf ms\n", (ms / N));

     return 0;
}
```

6. Código C++

```
/*------------------------------------------------------------------
--

*

* Multiprocesadores: Euler C++

* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*

*------------------------------------------------------------------
*/
```

```
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cmath>
#include <algorithm>
#include "utils.h"

#define MAXIMUM 1000000000 //5e6

using namespace std;

// implement your class here
class EulerPi{
    private:
        double acum;
    public:
        EulerPi(){
            acum = 0;
        }
        void calcPi(int n){
            for(int i = 1; i < n; i++){
                acum += 1.0/((double)i*(double)i);
            }
        }
        double getResult(){
            return acum;
        }
};
```

```
int main(int argc, char* argv[]) {
     int i;
     double ms;
    double result;
     cout << "Starting..." << endl;
     ms = 0;
     // create object here
     for (int i = 0; i < N; i++) {
          start_timer();
        EulerPi e;
        e.calcPi(MAXIMUM);
          // call your method here.
        result = sqrt(e.getResult()*6);
          ms += stop_timer();
     }

     cout << "result = " << setprecision(10) << result << " ";
     cout << "avg time = " << setprecision(15) << (ms / N) << "
ms" << endl;

     return 0;
}
```

7. Código C++ TBB

```
/*------------------------------------------------------------------
--

*

* Multiprocesadores: Euler C++ TBB
```

```
* Fecha: 27-Noviembre-2022

* Autor: Guillermo Fidel Navarro Vega A01274191

*

*------------------------------------------------------------------
*/



//
==================================================================
=

#include <iostream>
#include <iomanip>
#include <cstring>
#include <cmath>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>
#include "utils.h"

const int MAXIMUM = 1000000000; //1e6

using namespace std;
using namespace tbb;

// place your code here
class EulerPi{
     private:
          int max;
```

```
        double acum = 0;
    public:
        EulerPi(int s){max = s; acum = 0;}
       EulerPi(EulerPi &x, split) : max(x.max), acum(0) {}
        void operator() (const blocked_range<int> &r) {
          for(int i = r.begin(); i < r.end(); i++){
               acum += 1.0/((double)i*(double)i);
          }
       }

      void join(const EulerPi &x){
           acum += x.acum;
      }

      double getResult() const {
           return acum;
      }

};



int main(int argc, char* argv[]) {
     int i;
     double ms;
     double result;
     cout << "Starting..." << endl;
     ms = 0;
     // create object here
     for (int i = 0; i < N; i++) {
```

```
          start_timer();
          EulerPi Pi(MAXIMUM);
        parallel_reduce(blocked_range<int>(1,MAXIMUM), Pi);
        result = Pi.getResult();
          // call your method here.

          ms += stop_timer();
     }
    result = sqrt(result*6);
     cout << "result = " << setprecision(10) << result << " ";
     cout << "avg time = " << setprecision(15) << (ms / N) << "
ms" << endl;
     return 0;
}
```