Aproximación de Pi utilizando la Serie de Nilakantha y Cómputo Paralelo

Hugo David Franco Ávila (A01654856)
Tecnológico de Monterrey, Campus Querétaro
A01654856@tec.mx

24 de noviembre de 2021

Resumen

En este documento se explora una comparación entre el tiempo de ejecución de un programa escrito de forma secuencial y en paralelo, se utilizó la serie de Nilikantha para la aproximación de Pi como algoritmo. En todas las instancias, el que estaba en paralelo tuvo un mejor (menor) tiempo de ejecución. Otro hallazgo importante es que la mejor opción para el número de procesadores a utilizar en ejecución es igual al número de procesadores lógicos (hilos) que tiene la computadora.

Abstract

In this document, the time of execution of a program written in both sequential and parallel paradigm is compared, the Nilikantha series for the approximation of Pi was used. In every instance, the program running in parallel had a better (lower) time of execution. Another finding was that the best choice when selecting the number of processors for execution, was equal to the number of threads the computer has.

Introducción

Cómputo secuencial

Se le conoce como cómputo secuencial, a aquel en que cada instrucción va después de otra. Si hay una tarea que retrasa la ejecución, todas las demás deben esperar a que esta termine para continuar. Este es el modelo que se solía utilizar para desarrollar software anteriormente, cuando la mayoría de los equipos de cómputo eran de un solo núcleo. Conforme se fueron haciendo más veloces los procesadores, el software fue ganando en el desempeño. A esto se le conoció como el "*free lunch*".

Cómputo paralelo

Es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente. Hay varios tipos: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tarea. Los equipos modernos tienen procesadores multinúcleo (figura

1), es decir, que cuenta con varios procesadores lógicos para realizar procesamiento, cada uno de estos con su propia memoria cache reservada.

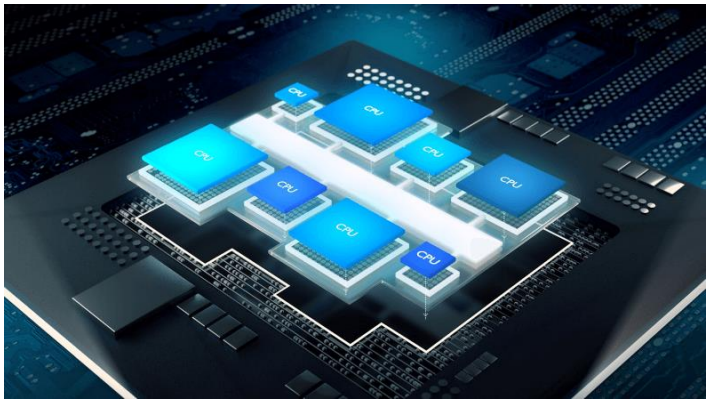

Figura 1. Arquitectura de un procesador multinúcleo

Serie de Nilikantha

La serie de Nilikantha, es una serie infinita con la cual se puede hacer una aproximación a la constante Pi. Esta fue ideada por el matemático Nilikantha Somayaji, y tiene la forma siguiente (figura 2).

$$\frac{\pi}{4} = \frac{3}{4} + \frac{1}{2.3.4} - \frac{1}{4.5.6} + \frac{1}{6.7.8} - \cdots$$

Figura 2. Serie de Nilikantha

Desarrollo

Se decidió trabajar con todas las tecnologías vistas en el semestre ya que, por la estructura del algoritmo, este era fácilmente representado en todos los lenguajes vistos. Primero se corrieron los programas secuenciales en los lenguajes C, C++ y Java, y se anotó su tiempo de ejecución. Se hicieron pruebas posteriores para ver como cambiaba el tiempo de ejecución de acuerdo con el input, y se obtuvo lo esperado, es decir, incrementaba de forma lineal (figura 3). En los tres lenguajes se comportó de la misma manera.

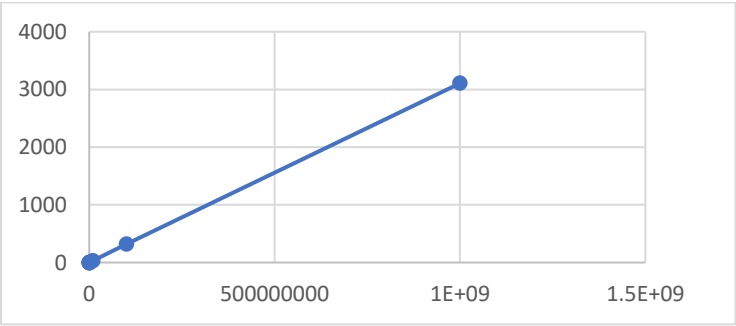

Figura 3. Tiempo de ejecución en C secuencial

Para la ejecución en paralelo, se tomó el tiempo que tardaba con diferente tamaño de input, así como con una cantidad de núcleos disponibles diferentes, tomando como referencia las potencias de dos.

OpenMP

Para el caso de OpenMP, se tomó el programa en C como el de referencia para el secuencial, y se hicieron las pruebas. Se encontró que el desempeño con 1 hilo era muy similar al de C (figura 4), y fue a partir de que suben el número de hilos que se tuvieron resultados interesantes.

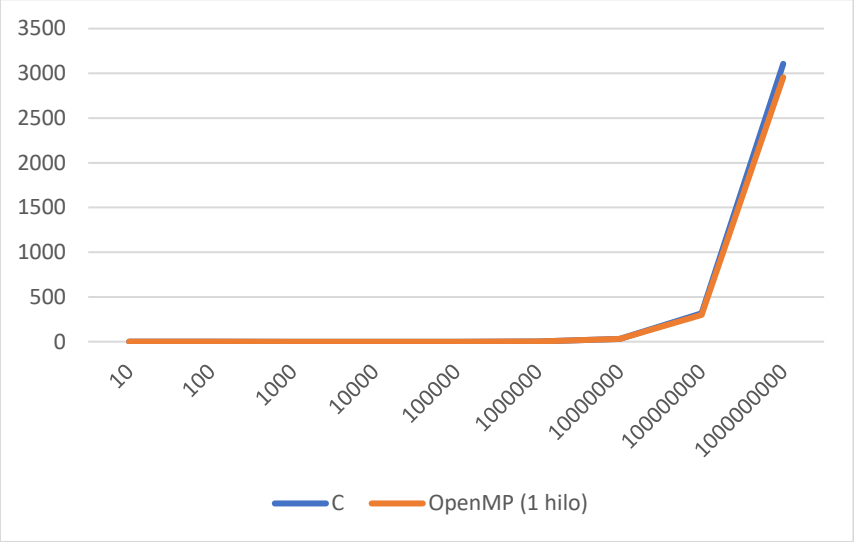

Figura 4. Comparación C vs OpenMP de 1 hilo

En OpenMP, el desempeño fue prácticamente duplicándose cada que aumentaba los hilos en un factor de 2 (figura 5). Siendo esto más visible a medida que crecía el input, cuando era muy pequeño, llegaba a incluso tener peor desempeño a medida que aumentaban los núcleos. A los 16 hilos todavía se encontraban mejoras en el tiempo de desempeño, lo cuál tiene sentido ya que mi computadora tiene un procesador de 16 núcleos. Cuando le puse que 32 el desempeño fue siempre igual o peor que con 16.

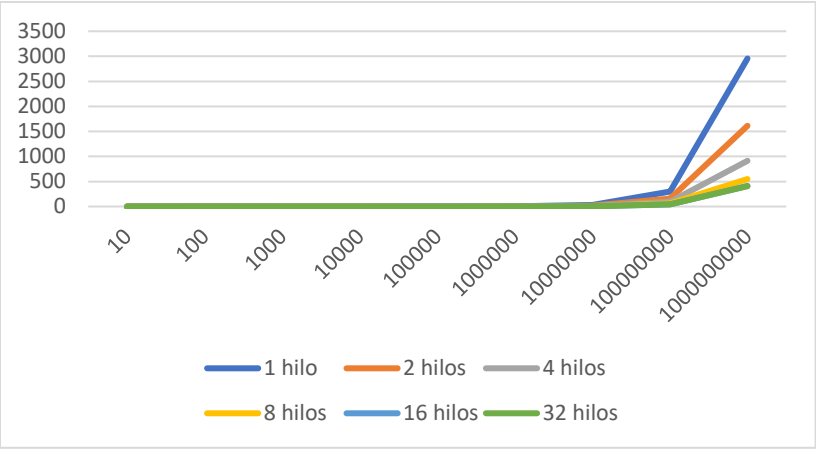

Figura 5. Comparativa por hilos en OpenMP

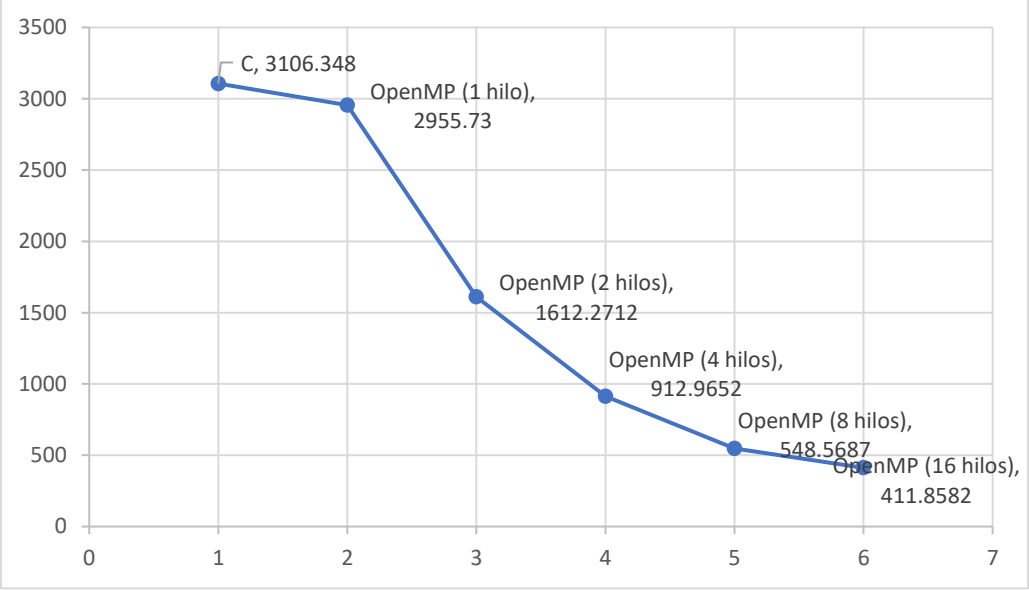

Figura 6. Comparación de los tiempos de ejecución de C vs OpenMP para el input más grande

Java Threads

En el caso de Java Threads, se encontró que la diferencia en tiempo de ejecución entre la versión secuencial y la de 1 sólo thread era prácticamente nula, lo cuál tiene sentido ya que en la forma secuencial se tiene un único hilo de ejecución.

En el caso particular de Java, no se notó mucha diferencia entre los tiempos de ejecución hasta llegar al input de 1 millón de elementos (figura 7). De igual manera, la ejecución óptima era con 16 hilos (figura 8).

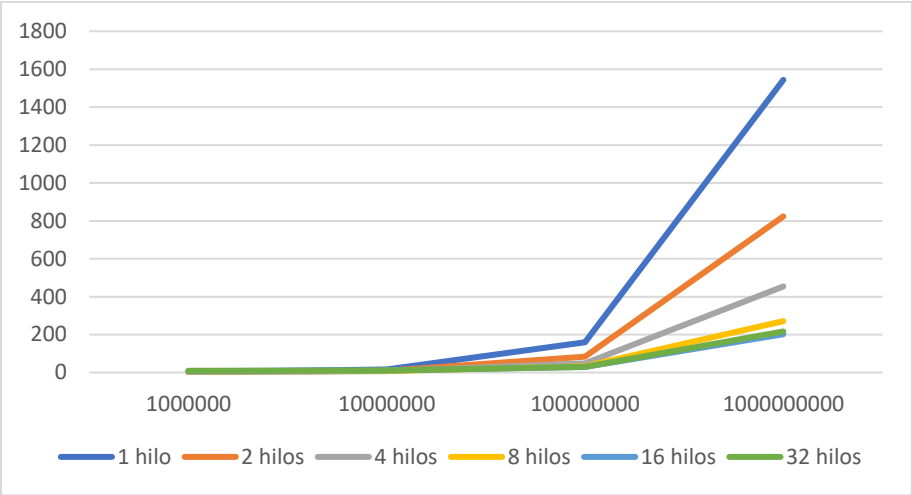

Figura 7. Ejecución de Java Threads para los input de 10 millones en adelante

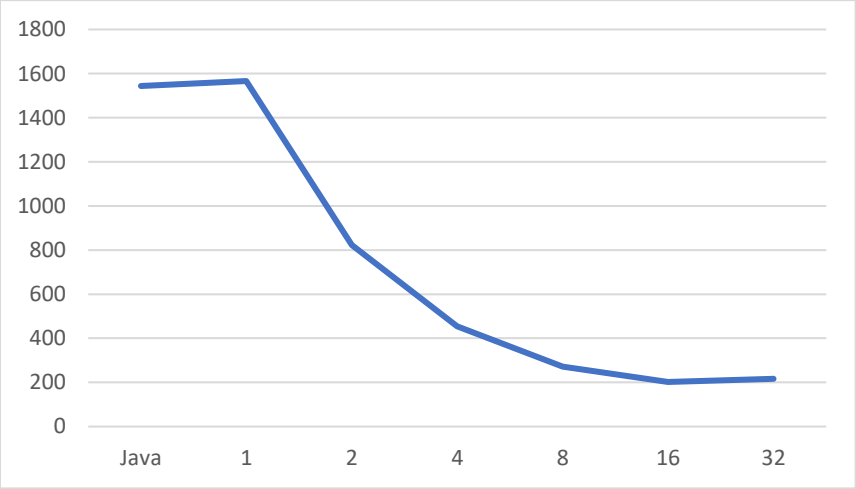

Figura 8. Comparación de los tiempos de ejecución de Java Threads para el input más grande

Java Fork-Join

Fork-Join es un framework de Java, el cual funciona al tener las tareas distribuidas entre un pool de Threads, bajo el paradigma de *work stealing*, esto significa que threads que se encuentren disponibles, van a robar trabajo de aquellos que siguen ocupados

En esta tecnología es en la primera que observé un cambió importante entre los tiempos de ejecución del programa secuencial, y de la tecnología paralela utilizando un solo hilo (figura 9). Esto se debe a la optimización que hace Fork-Join al tener la tarea dividida de forma recursiva. Es en los inputs más grandes donde se puede observar mejor la mejora.

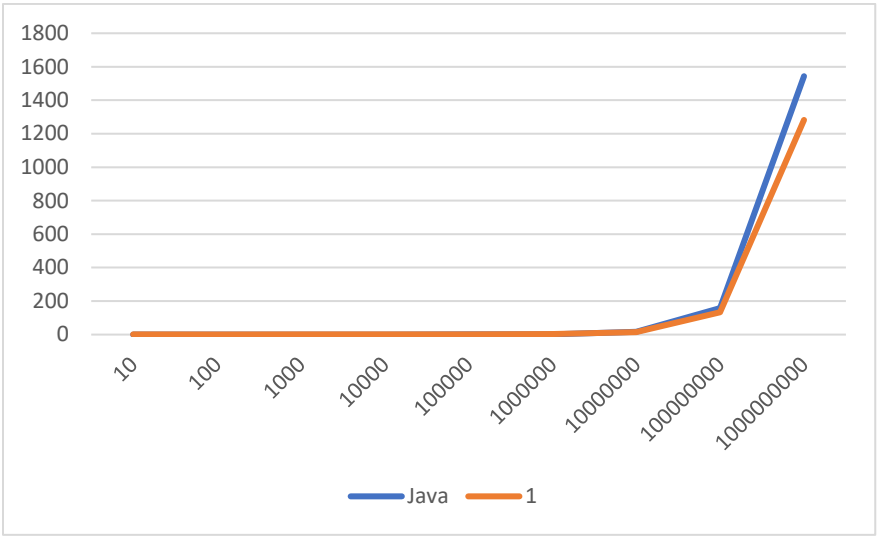

Figura 9. Comparación Java vs Fork-Join de 1 hilo

Se observó una mejora en el tiempo de ejecución a medida que se aumentaba en un factor de 2 el número de threads (figura 10), y fue con 16 threads (figura 11) en donde se encontró el mejor desempeño, que corresponde al número de procesadores lógicos de mi equipo.

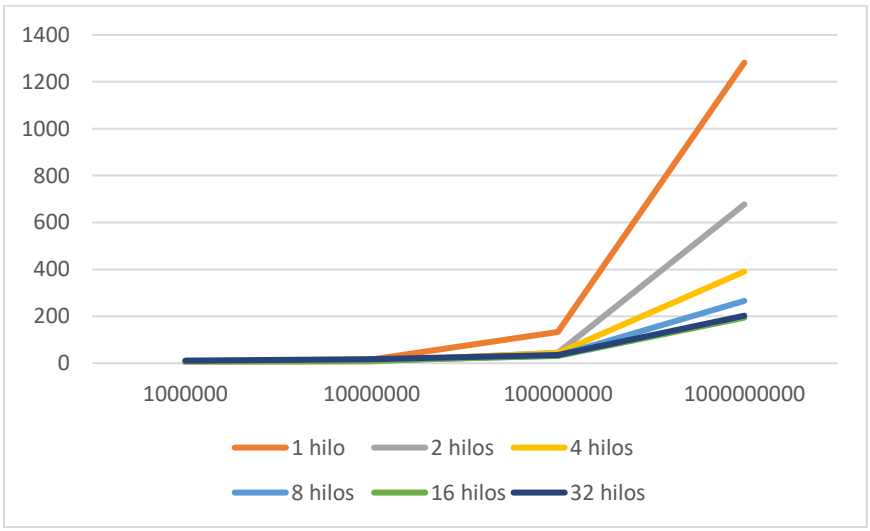

Figura 10. Ejecución de Fork-Join en potencias de 2 threads

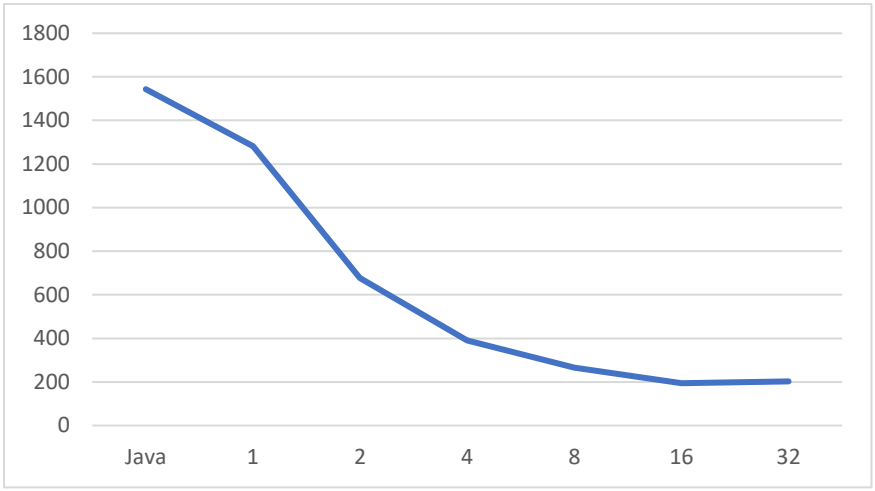

Figura 11. Comparación de los tiempos de ejecución de Fork-Join para el input más grande

Intel Thread Building Blocks

Esta es una librería para C++, que, a diferencia de otro tipo de paralelismo, hace énfasis en la programación de datos-paralelos, lo que permite a múltiples hilos trabajar en diferentes partes de una colección, en lugar de un bloque de tareas determinado.

Por la manera en que funciona la librería, no fue posible hacer la prueba con un número determinado de threads, sin embargo, se realizaron pruebas comparando el tiempo de ejecución del programa secuencial, con los tiempos del paralelo utilizando 16 threads (figura 12), para diferentes tamaños de input. Encontrando que se empieza a notar una enorme diferencia a partir de que se llega a input muy grande (100 millones).

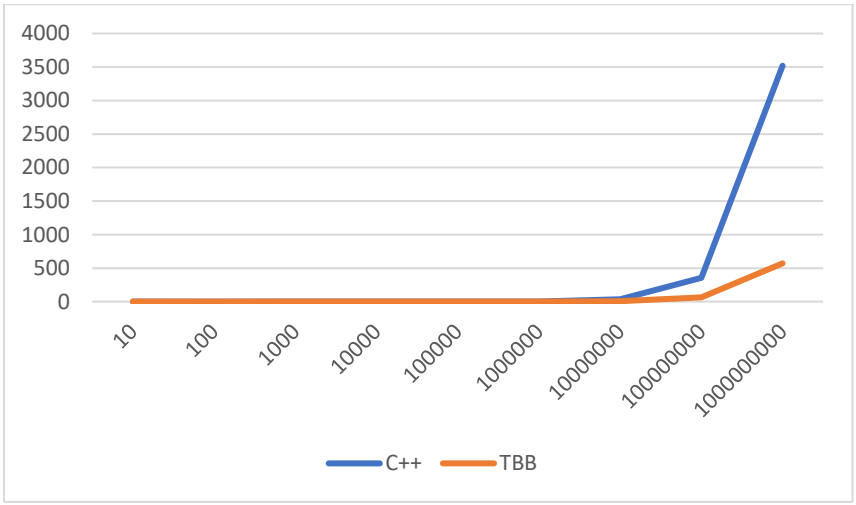

Figura 12. Comparación entre la implementación secuencial y la paralela, de C++ y TBB respectivamente

CUDA C

CUDA es una plataforma de cómputo paralelo desarrollado por NVIDIA para el procesamiento en unidades de gráficos (GPU). El número de hilos de procesamiento disponibles aumenta considerablemente, hasta llegar a los miles, lo que permite que las operaciones se realicen mucho más rápido.

Para las pruebas de CUDA (figura 13) decidí probar con input variante en potencias de 10, igual que con las otras tecnologías, así como con un número potencia de 2 para los hilos a utilizar por bloque. El número de bloques permaneció constante en todas las pruebas.

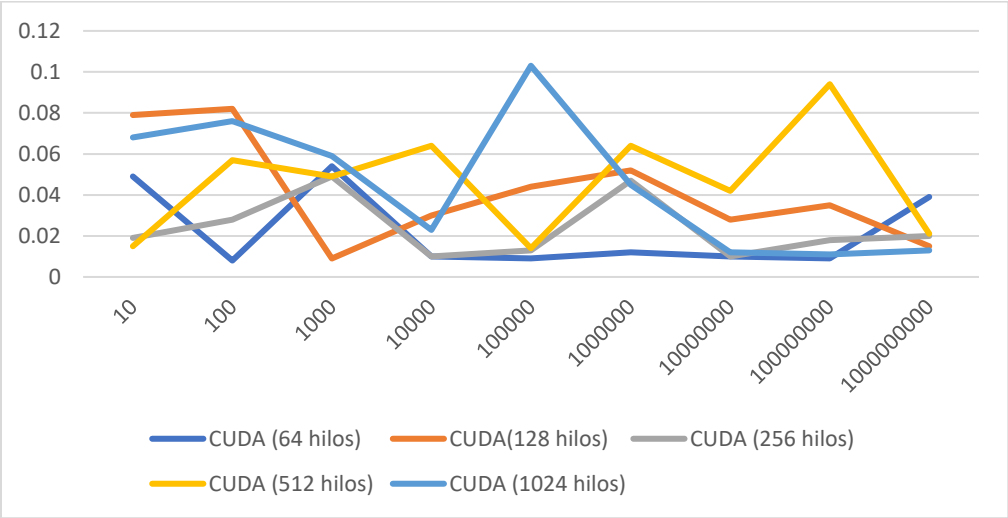

Figura 13. Pruebas con diferente número de hilos en CUDA

Como se puede apreciar en la gráfica, no hay una tendencia clara respecto a los tiempos de ejecución, lo que sí es claro, es que es miles de veces más rápido que la versión secuencial en C. Se realizó una comparación en la que se decidió comparar de con un input entre 10 y 10 millones (figura 14).

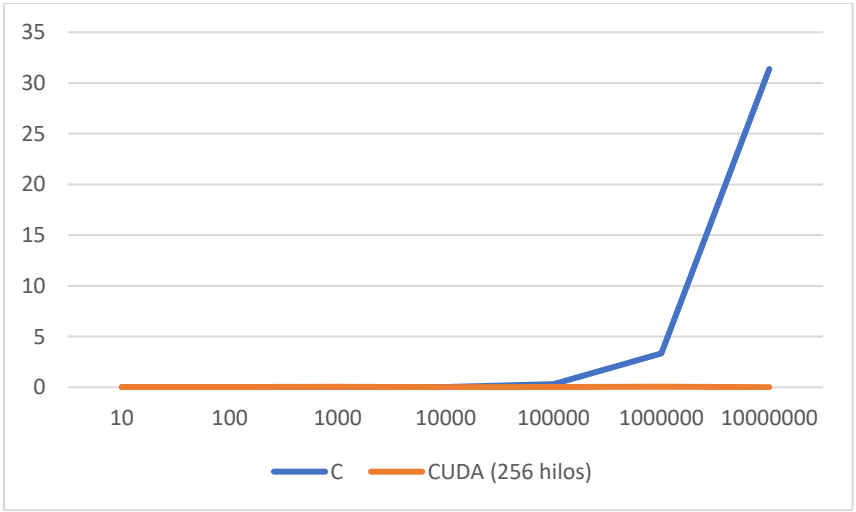

Figura 14. Comparación C secuencial vs CUDA de 256 hilos

Se llenó una tabla (figura 15) con los datos de los tiempos de ejecución para todos los programas de forma secuencial, y todos los paralelos utilizando 16 hilos, y en el caso de CUDA 256 por bloque.

C	C++	Java	Threads	Fork-Join	OpenMP	TBB	CUDA
3106.348	3621.8113	1647.1	177	176.1	403.663	533.5571	0.009

Figura 15. Tabla con todos los tiempos de ejecución

A partir de esta tabla se hizo el cálculo del Speedup (figura 16).

Technology	Original	Time	Speedup
Threads	1647.1	177	9.305649718
Fork-Join	1647.1	176.1	9.353208404
OpenMP	3106.348	403.663	7.69539938
TBB	3621.8113	533.5571	6.788048177
CUDA	3106.348	0.009	345149.7778

Figura 16. Speedup

Conclusiones

El uso del cómputo paralelo nos permite realizar tareas de forma mucho más veloz a como se hacía de forma secuencial. Con los avances en la tecnología de los procesadores disponibles en dispositivos de uso masivo, es claro que sólo los más simples de los programas pueden seguir corriendo en secuencial. Ya no es posible seguir desperdiciando poder de procesamiento que ya se encuentra disponible, cómo programadores debemos encontrar la forma de utilizarlo.

En todas las instancias la ejecución en paralelo del algoritmo de la Serie de Nilikantha fue mayor, siendo en CUDA donde más se vio un incremento, siendo la ejecución 345 mil veces más rápida que en secuencial. Esto tiene sentido ya que CUDA tiene un gran conjunto de hilos y bloques para realizar procesamiento.

También a partir de los resultados obtenidos, se puede concluir que el mejor número de hilos a escoger para paralelizar las tareas es igual al número de procesadores lógicos disponibles en el equipo. Un número menor no haría uso de toda la capacidad del procesador, y uno mayor forzosamente haría esperar a al menos un hilo para empezar a ejecutarse, lo que traería un mayor tiempo de ejecución.

Por último, también se debe considerar el tamaño del problema en cuestión para tomar la decisión de paralelizar un programa. El mayor beneficio a realizar la tarea en paralelo se observó con inputs grandes, mientras que los que eran pequeños no mostraron gran mejoría, y al contrario, a veces era mayor el tiempo de procesamiento incluso comparado con el programa secuencial, ya que el sistema toma más tiempo en dividir la tarea que en realizarla. Por lo que hay casos en los que es mejor no paralelizar el problema.

Agradecimientos

Quisiera agradecer a mi novia Vianey y mi perrito Milaneso por brindarme compañía y dejarme concentrar en el trabajo.

Quisiera agradecer a mis papás por siempre estar apoyándome, y permitiéndome el poder enfocarme en mis estudios.

Quisiera agradecer al Prof. Pedro Pérez por la enseñanza de las tecnologías utilizadas en el trabajo de investigación.

Referencias

Intel. (s. f.). Advanced HPC Threading: Intel® oneAPI Threading Building Blocks. Recuperado 26 de noviembre de 2021, de https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.hmdpoo

NVIDIA. (2021, 26 julio). CUDA Zone. NVIDIA Developer. Recuperado 26 de noviembre de 2021, de https://developer.nvidia.com/cuda-zone

Oracle. (s. f.). Fork/Join (The JavaTM Tutorials > Essential Java Classes > Concurrency). Recuperado 26 de noviembre de 2021, de https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

Orbe, A. (s. f.). Procesamiento secuencial y paralelo. Tareas, ordenadores y cerebro. Sinapsis. Recuperado 25 de noviembre de 2021, de http://sinapsis-aom.blogspot.com/2011/01/procesamiento-secuencial-y-paralelo.html

Sebah P., Gourdon, X. (31 de marzo de 2004). Collection of series for Pi. Recuperado 25 de noviembre de 2021 de http://math.bu.edu/people/tkohl/teaching/spring2008/piSeries.pdf

Apéndices

Código fuente

(a) C secuencial

```
/*------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje C. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
```

```
    El algoritmo está de forma secuencial.
    Hago uso del archivo utils.h generado por el Prof. Pedro Pérez
* Comando para compilar en Linux: gcc nilikantha.c
* Comando para correr en Linux: ./a.out
* NOTA: De especificar otro nombre del ejecutable al compilar
utilizar ./nombreDelEjecutable.out
*-------------------------------------------------------------
*/

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

#define SIZE 1000000000 // 1e9

/**
 * @brief This function implements the Nilikantha series up to a
given n (size), this to
 * give an approximation of Pi. With a larger n, the approximation
is more precise.
 *
 * @param size int
 * @return double
 */
double nilik(int size){
    double result = 3.0;
    double sign, di, denominator, term;
    int i = 1;
    for(i; i <= size; i++){
        sign = i % 2 == 0 ? -1.0 : 1.0;
        di = i*2.0;
        denominator = (di*(di+1)*(di+2));
        term = (4.0*sign)/denominator;
        result += term;
    }
    return result;
}

int main(int argc, char * argv[]){
    int size = SIZE, i = 0;
    double result = 0.0, time = 0.0;
    if(argc >= 2){
        printf("Error: No arguments are allowed\n");
        return -1;
    }
    printf("Starting...\n");
```

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

        result = nilik(size);

        time += stop_timer();
   }
    printf("Pi is approx: %.15lf \n", result);
    printf("Average time elapsed: %.7lf ms\n", (time / N));
    return 0;
}
```

(b) C++ secuencial

```
/*-------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje C++. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está de forma secuencial.
    Hago uso del archivo utils.h generado por el Prof. Pedro Pérez
* Comando para compilar en Linux: g++ nilikantha.cpp
* Comando para correr en Linux: ./a.out
* NOTA: De especificar otro nombre del ejecutable al compilar
utilizar ./nombreDelEjecutable.out
*-------------------------------------------------------------
*/
#include <iostream>
#include <iomanip>
#include "utils.h"

#define SIZE 1000000000 //1e9

using namespace std;

class Nilikantha {
private:
    int size;

    double result;
```

```
public:
    /**
     * @brief Construct a new Nilikantha object
     *
     * @param s int
     */
    Nilikantha(int s) : size(s) {}

    /**
     * @brief Get the result value from the Nilikantha object
     *
     * @return double
     */
    double getResult() {
        return this->result;
    }

    /**
     * @brief This function does the calculation of the Nilikantha
series up to the size defined
     * in the object constructor.
     *
     */
    void calculate() {
        result = 3.0;
        double sign, di, denominator, term;
        int i = 1;
        for (i; i <= size; i++) {
            sign = i % 2 == 0 ? -1.0 : 1.0;
            di = i * 2.0;
            denominator = (di * (di + 1) * (di + 2));
            term = (4.0 * sign) / denominator;
            result += term;
        }
    }
};

int main(int argc, char *argv[]) {
    int size = SIZE, i = 0;
    double result = 0.0, time = 0.0;
    if (argc >= 2) {
        cout << "Error: No arguments are allowed" << endl;
        return -1;
    }
    cout << "Starting..." << endl;
    Nilikantha nik(size);
```

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

        nik.calculate();

        time += stop_timer();
    }
    cout << "Pi is approx: " << setprecision(15) <<
nik.getResult() << endl;
    cout << "Average time elapsed: " << setprecision(15) << (time
/ N) << " ms" << endl;
    return 0;
}
```

(c) Java secuencial

```
/*-------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje Java. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está de forma secuencial.
* Comando para compilar en Linux: javac Nilikantha.java
* Comando para correr en Linux: java Nilikantha
*-------------------------------------------------------------
*/
public class Nilikantha {
    private static final int SIZE = 1_000_000_000;
    private static final int N = 10;
    private double result;

    /**
     * Default constructor for the Nilikantha class
     */
    public Nilikantha(){}

    /**
     * This function receives an integer s, and calculates the
Nilikantha series up to the s-term,
     * which represents an approximation to the term Pi. The
larger the value of S, the better
     * approximation you  get.
```

```
     *
     * @param s int
     */
    public void calculate(int s){
        result = 3.0;
        double sign, di, denominator, term;
        for(int i = 1; i <= s; i++){
            sign = i % 2 == 0 ? -1.0 : 1.0;
            di = i*2.0;
            denominator = (di*(di+1)*(di+2));
            term = (4.0*sign)/denominator;
            result += term;
        }
    }

    /**
     * This function returns the value of result in the Nilikantha
object
     * @return double
     */
    private double getResult(){
        return this.result;
    }

    public static void main(String args[]){
        if(args.length >= 1){
            System.out.println("Error:    No    arguments    are
allowed");
            return;
        }
        long startTime, stopTime;
        double acum = 0;

        Nilikantha n = new Nilikantha();
        acum = 0;
        System.out.printf("Starting...\n");
        for (int i = 0; i < N; i++) {
              startTime = System.currentTimeMillis();

              n.calculate(SIZE);

              stopTime = System.currentTimeMillis();

              acum += (stopTime - startTime);
        }
        System.out.println("Pi is approx: " + n.getResult());
```

```
        System.out.println("Average time elapsed: " + (acum / N)
+ " ms");
    }
}
```

(d) C OpenMp

```
/*------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje C. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está paralelizado, utilizando la tecnología de
OpenMP.
    Hago uso del archivo utils.h generado por el Prof. Pedro Pérez
* Comando para compilar en Linux: gcc nilikantha.c -fopenmp
* Comando para correr en Linux: ./a.out
* NOTA: De especificar otro nombre del ejecutable al compilar
utilizar ./nombreDelEjecutable.out
*-------------------------------------------------------------
*/
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "utils.h"

#define SIZE 1000000000 //1e9

/**
 * @brief The function belows receives a parameter size, which
represents the nth term of the
 * Nilikantha series wanted for the approximation of Pi. The
functions uses the OpenMP directive
 * parallel for, which automatically divides a for statement to
be processed by individual threads.
 * The directive declares the size variable to be shared across
all threads, and an individual copy
 * of the variables: sign, di, denominator and term are created
for each thread. Another declaration
 * added is the reduction keyword, which signals a reduction
operation on the selected variable,
```

```
 * and which operation; in this case it is a sum of every
individual value of result.
 *
 * @param size int
 * @return double
 */
double nilik(int size) {
   double result = 3.0;
   int i;
   double sign, di, denominator, term;
   #pragma omp parallel for shared(size) private(sign, di,
denominator, term) reduction(+:result)
   for(i = 1; i <= size; i++){
        sign = i % 2 == 0 ? -1.0 : 1.0;
        di = i*2.0;
        denominator = (di*(di+1)*(di+2));
        term = (4.0*sign)/denominator;
        result += term;
    }
   return result;
}

int main(int argc, char* argv[]) {
    if(argc >= 2){
        printf("Error: No arguments are allowed\n");
        return -1;
    }
   int i;
   double ms, result;

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

        result = nilik(1000000000);

        ms += stop_timer();
   }
   printf("Pi is approx: %.15lf \n", result);
    printf("Average time elapsed: %.7lf ms\n", (ms / N));

   return 0;
}
```

(e) Java Threads

```
/*-----------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje Java. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está paralelizado, utilizando la tecnología de
Java Threads.
* Comando para compilar en Linux: javac NilikanthaThreads.java
* Comando para correr en Linux: java NilikanthaThreads
*-----------------------------------------------------------------
*/
public class NilikanthaThreads extends Thread {
    private static final int SIZE = 1_000_000_000;
    public      static      final      int      MAXTHREADS      =
Runtime.getRuntime().availableProcessors();
    private static final int N = 10;
    private int start, end;
    private double result;

   /**
    *  Constructor  for  the  NilikanthaThreads  class.  It  will
calculate the Nilikantha
    * series from the start-term to the end-term and save the
result to the variable
    * "result"
    * @param start int - Start point of the calculation
    * @param end int - End point of the calculation
    */
    public NilikanthaThreads(int start, int end) {
        this.start = start;
        this.end = end;
        this.result = 0;
   }

   /**
    * This fuction implements the run function of the Thread class.
Here, the
    * Nilikantha series going from start to end is calculated and
saved to the class variable result.
    */
   @Override
```

```
    public void run(){
        double sign, di, denominator, term;
        if(this.start == 0){
            this.result = 3.0;
            this.start++;
        }
        for(int i = start; i <= end; i++){
            sign = i % 2 == 0 ? -1.0 : 1.0;
            di = i*2.0;
            denominator = (di*(di+1)*(di+2));
            term = (4.0*sign)/denominator;
            this.result += term;
        }
    }

   /**
    * Getter for the class variable result
    * @return double
    */
    private double getResult(){
        return this.result;
    }

    public static void main(String args[]){
        if(args.length >= 1){
            System.out.println("Error:    No    arguments    are
allowed");
            return;
        }
        long startTime, stopTime;
        int block;
        NilikanthaThreads threads[];
        double ms;
        double result = 0.0;

        /**
         * block represents the amount of calculations each thread
is going to process
         */
        block = SIZE / MAXTHREADS;
        /**
         * The number of threads created is the max my local
         * machine allows, which is 16. By doing this, you ensure
you are using all the
         * threads and not assigning extra work to an individual
thread, or leaving threads
```

```
         * without work to do.
         */
        threads = new NilikanthaThreads[MAXTHREADS];

        System.out.printf("Starting   with   %d   threads...\n",
MAXTHREADS);
        ms = 0;
        for (int j = 1; j <= N; j++) {
             /**
              * Threads are created and assigned their start and
end values
              */
             for (int i = 0; i < threads.length; i++) {
                  if (i != threads.length - 1) {
                       threads[i] = new NilikanthaThreads((i *
block), ((i + 1) * block));
                  } else {
                       threads[i] = new NilikanthaThreads((i *
block), SIZE);
                  }
             }

             startTime = System.currentTimeMillis();
             /**
              * The block below starts all the threads, this
means it executes the
              * run() method on every thread.
              */
             for (int i = 0; i < threads.length; i++) {
                  threads[i].start();
             }
             /**
              * The block below signals the main thread to wait
for every individual thread to finish
              */
             for (int i = 0; i < threads.length; i++) {
                  try {
                       threads[i].join();
                  } catch (InterruptedException e) {
                       e.printStackTrace();
                  }
             }
             stopTime = System.currentTimeMillis();
             ms +=  (stopTime - startTime);
             /**
```

```
               * In the last iteration, the result from every
thread gets added to calculate
               * the approximation
               */
              if (j == N) {
                    result = 0;
                    for (int i = 0; i < threads.length; i++) {
                          result += threads[i].getResult();
                    }
              }
        }
        System.out.println("Pi is approx: " + result);
        System.out.println("Average time elapsed: " + (ms / N) +
" ms");
    }
}
```

(f) Java Fork-Join

```
/*-------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje Java. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está paralelizado, utilizando la tecnología de
Fork-Join
* Comando para compilar en Linux: javac NilikanthaThreads.java
* Comando para correr en Linux: java NilikanthaThreads
*-------------------------------------------------------------
*/
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class NilikanthaForkJoin extends RecursiveTask<Double>{
    private static final int SIZE = 1_000_000_000;
    private static final int MIN = 100_000;
    public      static      final      int      MAXTHREADS      =
Runtime.getRuntime().availableProcessors();
    private static final int N = 10;
    private int start, end;
    private double result;
```

```
    /**
    * Constructor for the NilikanthaThreads class. It will calculate the Nilikantha
    * series from the start-term to the end-term and save the result to the variable
    * "result"
    * @param start int - Start point of the calculation
    * @param end int - End point of the calculation
    */
    public NilikanthaForkJoin(int start, int end) {
        this.start = start;
        this.end = end;
        this.result = 0;
   }

    /**
     * This function performs the calculation of the Nilikantha series from the value of start,
     * to the value of end, adding the calculation into the variable result.
     * @return Double
     */
    protected Double computeDirectly(){
        this.result = 0.0;
        double sign, di, denominator, term;
        if(this.start == 0){
            this.result = 3.0;
            this.start++;
        }
        for(int i = start; i <= end; i++){
            sign = i % 2 == 0 ? -1.0 : 1.0;
            di = i*2.0;
            denominator = (di*(di+1)*(di+2));
            term = (4.0*sign)/denominator;
            this.result += term;
        }
        return result;
    }

    /**
     * This function divides the amount of calculations into k processes of
     * n calculations. After reaching the minimum threshold, it will call computeDirectly
     * which will perform the calculation. Until then it divides the amount of calculations by a
```

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

    public static void main(String args[]){
        if(args.length >= 1){
            System.out.println("Error:    No    arguments    are
allowed");
            return;
        }
        long startTime, stopTime;
        double result = 0.0;
        double ms;
        ForkJoinPool pool;

        System.out.printf("Starting   with   %d   threads...\n",
MAXTHREADS);
        ms = 0;
        for (int i = 0; i < N; i++) {
              startTime = System.currentTimeMillis();
             /**
              * The number of threads assigned for the ForkJoin
pool, is the maximum amount of
              * logical threads my machine has
              */
              pool = new ForkJoinPool(MAXTHREADS);
             /**
              * The line belows starts the calculation of the
Nilikantha series
              * from 0 to the value of SIZE
              */
```

```
            result   =   pool.invoke(new   NilikanthaForkJoin(0,
SIZE));

            stopTime = System.currentTimeMillis();
            ms += (stopTime - startTime);
        }
        System.out.println("Pi is approx: " + result);
        System.out.println("Average time elapsed: " + (ms / N) +
" ms");
    }

}
```

(g) C++ Intel Thread Building Blocks

```
/*------------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje C++. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está paralelizado, utilizando la tecnología de
Intel Thread Building Blocks.
    Hago uso del archivo utils.h generado por el Prof. Pedro Pérez
* Comando para compilar en Linux: g++ nilikantha.cpp -ltbb
* Comando para correr en Linux: ./a.out
* NOTA: De especificar otro nombre del ejecutable al compilar
utilizar ./nombreDelEjecutable.out
*-----------------------------------------------------------------
*/
#include <iostream>
#include <iomanip>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>
#include "utils.h"

const int SIZE = 1000000000; //1e9

using namespace std;
using namespace tbb;

class Nilikantha {
private:
```

```
   double result;

public:
   /**
    * @brief Construct a new Nilikantha object. Result is
initialized with 3.0 because this is
    * the first Nilikantha object, which corresponds to the first
term of the
    * Nilikantha series.
    *
    */
   Nilikantha() : result(3.0) {}

   /**
    * @brief Construct a new Nilikantha object. Result is
initialized with 0.0 because every
    * subsequent Nilikantha objects do not need to know about the
beginning of the series.
    * The range for the calculation of Pi begins at the term 1,
not 0.
    *
    * @param x
    */
   Nilikantha(Nilikantha &x, split): result(0.0) {}

   /**
    * @brief Getter for the result member variable
    *
    * @return double
    */
   double getResult() const {
        return result;
   }

   /**
    * @brief Operation definition to be performed on every element
    * within the range defined by . In this case, it is obtaining
the nth
    * element of the Nilikantha series from r.begin to r.end, and
adding it
    * to the class variable result
    *
    * @param r blocked_range<int>
    */
   void operator() (const blocked_range<int> &r) {
        double sign, di, denominator, term;
```

```
        for (int i = r.begin(); i != r.end(); i++) {
              sign = i % 2 == 0 ? -1.0 : 1.0;
            di = i * 2.0;
            denominator = (di * (di + 1) * (di + 2));
            term = (4.0 * sign) / denominator;
            result += term;
        }
   }

   /**
    * @brief Reduce operation to be performed
    *
    * @param x Nilikantha
    */
   void join(const Nilikantha &x) {
        result += x.result;
   }
};

int main(int argc, char* argv[]) {
    if (argc >= 2) {
        cout << "Error: No arguments are allowed" << endl;
        return -1;
    }
   double result = 0.0;
   double ms;

   cout << "Starting..." << endl;
   ms = 0;
   for (int i = 1; i <= N; i++) {
        start_timer();
        /**
         * @brief Creation of the first object
         *
         */
        Nilikantha obj;
        /**
         *  @brief  Using  Intel's  tbb,  parallel  reduction  is
performed from
         * 1  to  SIZE  for  the  object  defined  previously,  this
executes
         * operator() and calls the join function defined in the
class.
         *
         */
        parallel_reduce(blocked_range<int>(1, SIZE), obj);
```

```
        result = obj.getResult();

        ms += stop_timer();
   }
   cout << "Pi is approx: " << setprecision(15) << (double)(result) << endl;
    cout << "Average time elapsed: " << setprecision(15) << (ms / N) << " ms" << endl;

   return 0;
}
```

(h) CUDA C

```
/*-------------------------------------------------------------
--
* Multiprocesadores: Proyecto final
* Fecha: 21-Nov-2021
* Autor: A01654856 Hugo David Franco Ávila
* Descripción: Este código implementa la serie de Nilikantha para
    obtener una aproximación suficientemente precisa de Pi en el
lenguaje CUDA C. A medida
    que incrementan los términos de la serie, la aproximación es
más precisa.
    El algoritmo está paralelizado, utilizando la tecnología de
CUDA de NVIDIA.
    Hago uso del archivo utils.h generado por el Prof. Pedro Pérez
* Comando para compilar en Linux: nvcc nilikantha.cu
* Comando para correr en Linux: ./a.out
* NOTA: De especificar otro nombre del ejecutable al compilar
utilizar ./nombreDelEjecutable.out
*-------------------------------------------------------------
*/
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include "utils.h"

#define SIZE 1000000000 //1e9
#define THREADS    256
#define BLOCKS     MMIN(32, ((SIZE / THREADS) + 1))

/**
 * @brief This function will be run on the device (NVIDIA GPU),
that is indicated by the
 * __global__ keyword. It will calculate the Nikilantha series
for n terms, and save every
```

```
 * partial sum for every block in the cache, which will be further
reduced to be saved in
 * the result array passed as parameter.
 *
 * @param size int
 * @return double
 */
__global__ void nilik(double *result) {
   /**
    * The array declared below, will be shared across all blocks,
and will
    * save the partial sum of the Nilikantha terms calculated in
the block
    */
   __shared__ double cache[THREADS];

   /**
    * The tid is a linearization of the memory in the GPU, and
will be used
    * for the calculation as the nth term
    */
   int tid = threadIdx.x + (blockIdx.x * blockDim.x);
   int cacheIndex = threadIdx.x;

   double acum = 0.0, sign, di, denominator, term;
   /**
    * Special case for the first element in the series
    */
    if(tid == 0){
        acum += 3.0;
        tid += blockDim.x * gridDim.x;
    }

   /**
    * The while loop below obtains every Nilikantha term for every
threadId index in
    * every block.
    */
   while (tid < SIZE) {
        sign = tid % 2 == 0 ? -1.0 : 1.0;
        di = tid*2.0;
        denominator = (di*(di+1)*(di+2));
        term = (4.0*sign)/denominator;
        acum += term;
        tid += blockDim.x * gridDim.x;
   }
```

```
   /**
    * Partial sum is saved in the cache
    */
   cache[cacheIndex] = acum;

   /**
    * The instruction below performs a block level syncronization
barrier, this
    * means it will be called when every thread reaches this line
in their execution pipeline
    */
   __syncthreads();

   /**
    * The code below, performs a reduction by adding the contents
of the element in a power of 2
    * distance to the element at the cacheIndex, which if recalled
is the same as the threadId,
    * it will stop after it reaches the original index.
    */
   int i = blockDim.x / 2;
   while (i > 0) {
         /**
          * The if block belows prevents accidentaly accesing a
non-valid index in the device.
          */
         if (cacheIndex < i) {
               cache[cacheIndex] += cache[cacheIndex + i];
         }
         /**
          * The line belows blocks all threads from advancing until
they reach this line in their
          * execution flow.
          */
         __syncthreads();
         i /= 2;
   }

   /**
    * After the contents in the block are summed up in each index
of the cache,
    * it wil be added to the result array at the position indicated
by its blockId
    */
   if (cacheIndex == 0) {
```

```
        result[blockIdx.x] = cache[cacheIndex];
   }
}

int main(int argc, char* argv[]) {
    if(argc >= 2){
        printf("Error: No arguments are allowed\n");
        return -1;
    }
   int i;
   double *results, *d_r;
   double ms;

   /**
    * Cache allocation in the host memory
    */
   results = (double*) malloc( BLOCKS * sizeof(double) );

   /**
    * Cache allocation in the device memory
    */
   cudaMalloc( (void**) &d_r, BLOCKS * sizeof(double) );

   printf("Starting...\n");
   ms = 0;
   for (i = 1; i <= N; i++) {
        start_timer();
        /**
         * This command calls the code defined in the function
nilik and specifies the number of blocks
         * and threads to be used for the calculation, the
parameter is the cache array
         * allocated in the device.
         */
        nilik<<<BLOCKS, THREADS>>> (d_r);

        ms += stop_timer();
   }

   /**
    * Results from the device are copied onto the host
    */
   cudaMemcpy(results,    d_r,    BLOCKS    *    sizeof(double),
cudaMemcpyDeviceToHost);

   /**
```

```
    * The block below performs a reduction in the results array,
getting the approximation of Pi.
    *  By using the parallel reduction earlier in the nilik
function, it allowed us to only
    * need to perform n-block amount of operations to get the
result, instead of adding the
    * result of every thread.
    */
   double acum = 0;
   for (i = 0; i < BLOCKS; i++) {
        acum += results[i];
   }

   printf("Pi is approx: %.15lf \n", acum);
    printf("Average time elapsed: %.3lf ms\n", (ms / N));

   /**
    * De-allocation of device memory
    */
   cudaFree(d_r);

   /**
    * De-allocation of host memory
    */
   free(results);
   return 0;
}
```

(g) utils.h

```
//
=============================================================
=
//
// File: utils.h
// Author: Pedro Perez
// Description: This file contains the implementation of the
//                 functions used to take the time and perform
the
//                 speed up calculation; as well as functions for
//                 initializing integer arrays.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
```

```
// Quiero agradecer al Prof. Pedro Pérez por implementar las funciones
// dentro de este archivo. Hago uso de este para un propósito no comercial.
// La única modificación que realicé fue eliminar las funciones en arreglos
// porque no las utilicé.
// Atte. Hugo David Franco Ávila
//
//
//============================================================
=

#ifndef UTILS_H
#define UTILS_H

#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>

#define MMIN(a,b) (((a)<(b))?(a):(b))
#define MMAX(a,b) (((a)>(b))?(a):(b))

#define N                    10

struct timeval startTime, stopTime;
int started = 0;

//
//============================================================
=
// Records the initial execution time.
//
//============================================================
=
void start_timer() {
   started = 1;
   gettimeofday(&startTime, NULL);
}

//
//============================================================
=
// Calculates the number of microseconds that have elapsed since
// the initial time.
```

```
//
// @returns the time passed
//
================================================================
=
double stop_timer() {
   long seconds, useconds;
   double duration = -1;

   if (started) {
         gettimeofday(&stopTime, NULL);
         seconds  = stopTime.tv_sec  - startTime.tv_sec;
         useconds = stopTime.tv_usec - startTime.tv_usec;
         duration = (seconds * 1000.0) + (useconds / 1000.0);
         started = 0;
   }
   return duration;
}

#endif
```