Multiprocesadores

Serie de Nilkantha

Profesor Pedro Oscar Pérez Murueta

Martin Noboa – A01704052

Agosto – Diciembre 2022

Noviembre 30, 2022

Contents

Resumen 2
Introducción 3
Serie de Nilakantha 3
Algoritmo 4
Desarrollo 5
OpenMP 6
TBB 6
Java Threads 7
Fork/Join 8
Conclusiones 9
Resultados 9
Conclusiones 10
Referencias 11
Apéndices 12
Apéndice 1 – C 12
Apéndice 2 – OpenMP 13
Apéndice 3 - C++ 14
Apéndice 4 – TBB 15
Apéndice 5 – Java 17
Apéndice 6 - Java Threads 18
Apéndice 7 - Fork/Join 20
Apéndice 8 – utils.h C 22
Apéndice 9 – utils.h C++ 24
Apéndice 10 – Utils.java Java 26

Resumen

Este artículo de investigación describa el algoritmo de Euler y su implementación con cuatro herramientas de programación en paralelo. Se describe el marco teórico de cada herramienta, el desarrollo de cada implementación secuencial como su implementación en paralelismo y se discuten resultados comparando la mejora (o falta de esta) de ambas implementaciones.

Introducción

Serie de Nilakantha

En Matemáticas, una serie es la suma de una secuencia infinita de números. Denotada de la siguiente manera:

$$S = a_0 + a_1 + a_2 + \cdots = \sum_{k=0}^{\infty} a_k$$

De una manera similar, se puede denotar la suma *parcial* de una serie limitando la secuencia:

$$S_n = \sum_{k=0}^{n} a_k$$

La serie de Nilakantha, nombrada así por el matemático indio del mismo nombre es una serie convergente en PI. PI, al ser un número irracional, es imposible de calcular su valor de manera exacta. Se dice que una serie converge cuando la suma parcial de la serie tiene a un límite. La serie de Nilakantha converge en PI, quiere decir que su suma parcial, mientras tiende al infinito, se aproximará mas y mas al valor de PI.

$$\pi \approx 3.141592$$

$$\pi \approx 3 + \sum_{n¿1}^{\infty} (-1)^{n+1} \left(\frac{4}{2n(2n+1)(2n+2)}\right)$$

Dada esta serie, así se verían los primeros términos:

Término	Aproximación de PI
1	3
2	3.1667
3	3.1333
4	3.1452

Algoritmo

Dado que, entre mayor el número de iteraciones de la serie, mejor la aproximación, la traducción de la serie a un algoritmo sería así:

$$\pi \approx 3 + \left(\frac{4}{2 * 3 * 4}\right) - \left(\frac{4}{4 * 5 * 6}\right) + \left(\frac{4}{6 * 7 * 8}\right) - \cdots$$

Como podemos deducir, la serie tiene los siguientes estándares para considerar:

- ✓ La serie comienza sumando un tres, a partir de eso es una iteración repetitiva de la misma operación.
- ✓ La iteración se suma si es un número de iteración impar, se resta si es impar.
- ✓ Los valores del denominador son secuenciales, y la siguiente iteración comienza en el dígito que se quedó la secuencia anterior.
    - Esto se puede representar de la siguiente manera, de acuerdo a la representación sumatoria de la serie:

$$\frac{4}{n(n+1)(n+2)}, \therefore n = 2 \; y \; n+= 2$$

    - Esto quiere decir que n se inicializa en 2 y se incrementa en 2 cada iteración.

Esto nos permite modelar el algoritmo de la siguiente manera:

```
PI = 3
n = 2
sign = 1
For i in range(n)
     PI = PI + (sign * (4 / ((n) * (n + 1)* (n + 2))))
     sign = sign * (-1)
     n += 2

endFor
```

Este es el algoritmo base que nos permitirá estructurar nuestro código. A partir de esto podemos construir los métodos necesarios.

Desarrollo

Para el desarrollo de esta investigación, se utilizó la metodología vista en clases. Se usaron 4 herramientas de programación en paralelo:

1. OpenMP.
2. Intel Threading Building Blocks.
3. Java Threads.
4. Fork/Join in Java.

De igual manera se implementó secuencialmente en los lenguajes base, para así poder medir el speed up. Esto se hizo usando la librería de **Utils** proporcionada por el profesor Pedro a lo largo del semestre.

```
for (i = 0; i < N; i++) {
        start_timer();
        //implmentación de función u método
        stop_timer();
    }
```

De esta forma medimos el tiempo de ejecución únicamente de la implementación del algoritmo. Se consigue un promedio repitiendo la ejecución ***N*** cantidad de veces, durante esta investigación fueron 10. Otro parámetro para considerar fue el número de iteraciones de la serie que se iban a realizar. Para beneficios de los resultados, se utilizó el mismo número de iteraciones para todas las implementaciones, para todos los lenguajes. Esto debido a que, como es una serie que converge en PI, estaríamos tratando cada vez más y mas con decimales mas grandes a medida que incrementa el número de iteraciones. Se realizaron 10 000 000 de iteraciones en cada implementación.

OpenMP

En base al código generado al Apéndice 1, se implementó la librería de OpenMP. Se usó un **[parallel for]** para realizar la suma de las iteraciones de la serie. Como el lenguaje base es C, no se realizó una clase , sino una función tipo ***Double*** que retornaba la aproximación de PI.

TBB

Para la implementación de Threading Building Blocks en C++, se creó una clase con los métodos de **Join** y **Operator.** De igual manera, hubo dos constructores para la clase, el constructor básico y el respectivo para TBB, que recibía un objeto de la misma clase.

```
class NilkanthaTBB {

private:

     double result, sign,n;

public:

NilkanthaTBB(double s, double a) : sign(s), n(a),result(3) {}

NilkanthaTBB(NilkanthaTBB &obj,split) : sign(obj.s),
n(obj.a),result(3) {}
```

Java Threads

En la implementación de threads en Java, se utilizó la función nativa de Java para encontrar el máximo número de threads disponibles en la computadora:

```
public static final int MAXTHREADS =
Runtime.getRuntime().availableProcessors();
```

Junto con esto, se realizó la división de bloques dividiendo la cantidad de iteraciones entre el número de threads disponible. Se pasaba un start y end a cada instancia del objeto por thread, donde cada start es el end del bloque anterior. Tomado en consideración si la iteración era par o impar se ajustaba el valor inicial para sumar o restar. Se instanció un arreglo del tamaño del número de threads y cada uno tenía su bloque para trabajar, retornando un valor parcial por thread. Con la ayuda de un try – catch se juntaron los threads y se obtuvo el valor final.

```
try {
     for (int j = 0; j < threads.length; j++) {
     threads[j].join();
     res += threads[j].getResult();
     }
} catch (InterruptedException ie) {
     ie.printStackTrace();
}
```

Fork/Join

Para la implementación del Fork/Join, se usó segmentación de bloques de igual manera. Es decir, cada Fork tenía su rango de iteraciones, y al final se juntaban los resultados. La implementación uso RecursiveTask, ya que cada Fork devolvía un valor parcial de tipo **Double**. Declaración de la clase:

```
public class NilkanthaForkJoin extends RecursiveTask<Double>
```

Hay que recordar que la diferencia entre RecursiveTask y RecursiveAction es que el primero permite retornar un valor, mientras que RecursiveAction implementa métodos de tipo *null*. Tomando de punto de inicio el primer valor de n (n = 2) y el end como el número de iteraciones, se fue dividiendo el rango de cada Fork por la mitad hasta llegar al min, que era un valor de 10 000 iteraciones.

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

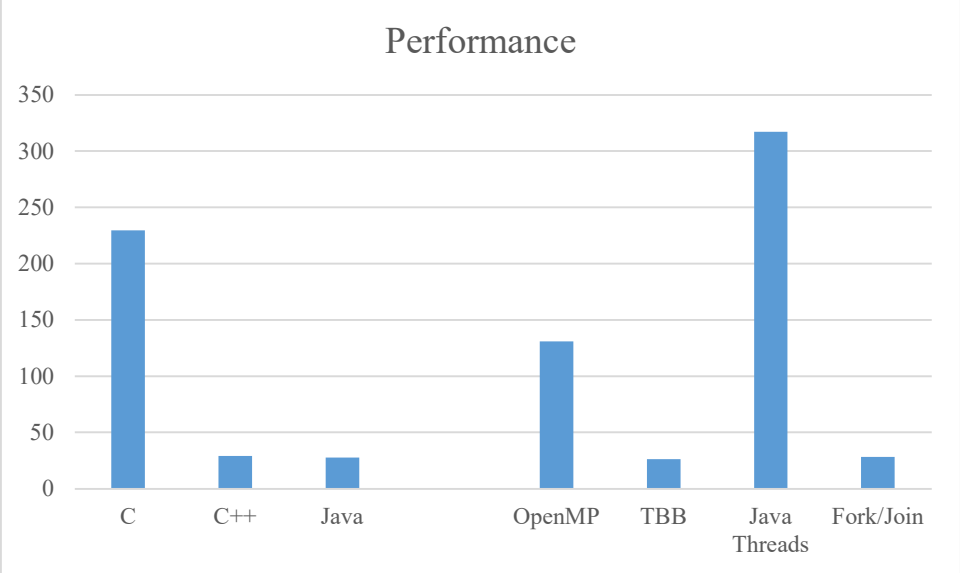
En ese momento, se pasaba del override de compute a computeDirectly, que implementa el algoritmo descrito en la sección de introducción.

```
    public Double computeDirectly() {
        if(this.start % 2 == 0){
            this.sign = this.sign * -1;
        }
        for (int i = this.start; i < SIZE; i++){
            this.result = this.result + (this.sign * (4.0
            / ((this.start) * (this.start + 1) *
            (this.start + 2))));
            this.sign = this.sign * (-1);
            this.start += 2;
        }
            return result;
    }
```

Conclusiones

Resultados

		Speed Up
C	229.4503	
C++	29.2393	
Java	27.7	
OpenMP	130.8258	1.753861241
TBB	26.1601	1.117705972
Java Threads	317.2	0.087326608
Fork/Join	28.2	0.982269504



Conclusiones

A pesar de que no en todos los casos se observo un speedup, podemos llegar a varias referencias de porqué sucedió esto. Primero y más seguro, debido a que se probó de manera local en mi computadora, esto afectó el rendimiento de las implementaciones. Como parte de futuros trabajos, se puede realizar las pruebas de la implementación con el servidor del campus Querétaro para verificar el rendimiento y desempeño.

Como segundo punto de mejora, podemos enfocarnos en una implementación diferente, haciéndolo a través de arreglos. De esta manera la segmentación de tareas entre threads es más directa y puede llegar a ser más eficaz.

Si hay un aprendizaje que me llevo con este trabajo de investigación, es que hay una herramienta para cada situación, y siempre varía de acuerdo a lo que buscas. Por ejemplo, implementar OpenMP es muy sencillo, sin embargo, CUDA (a pesar de que no fue usado en este trabajo de investigación) , es mucho más veloz. Siempre hay una herramienta para cada trabajo, y estar familiarizado con cada herramienta al menos un poco, ayudará a elegir cual es la mejor.

Referencias

B.V., M. (2022). *MAECKES B.V.* Obtenido de Nilakantha's formula for pi: http://www.maeckes.nl/Formule%20voor%20pi%20(Nilakantha)%20GB.html

Brink, D. (2015). *Nilakantha's accelerated series for pi.* ResearchGate.

Khan Academy. (2017). Obtenido de Pascal: Calculating Pi (Nilakantha series): https://www.khanacademy.org/computer-programming/pascal-calculating-pi-nilakantha-series/6592993584152576

Kissell, J. (2009). *Mac Security Bible.* Indiana: O'Rilley.

Lloyd, J. (1986). *Foundations of logic programming.* Berlin.

Apéndices

Apéndice 1 – C

```c
/*------------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
* C
*
*------------------------------------------------------------------*/

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

#define SIZE 10000000

double Nilkantha(double n, double sign) {
  double pi = 0;
  for (int i = 0; i <= SIZE; i++) {
        pi = pi + (sign * (4 / ((n) * (n + 1) * (n + 2))));
        sign = sign * (-1);
        n += 2;
    }
    return pi + 3;
}

int main(int argc, char* argv[]) {
  double n = 2, sign =1;
  int i;
  double ms, result;
  printf("Starting...\n");
  ms = 0;
  for (i = 0; i < N; i++) {
    start_timer();
    result = Nilkantha(n, sign);
    ms += stop_timer();
  }
  printf("Pi =  %0.9lf\n",result);
  printf("avg time = %.5lf ms\n", (ms / N));
  return 0;
}
```

Apéndice 2 – OpenMP

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

#define SIZE 10000000

double Nilkantha(double n, double sign) {
  double pi = 0;
  int i;
  #pragma omp parallel for reduction(+:pi)
  for (i = 0; i <= SIZE; i++) {
        pi = pi + (sign * (4 / ((n) * (n + 1) * (n + 2))));
        sign = sign * (-1);
        n += 2;
    }
    return pi;
}

int main(int argc, char* argv[]) {
  int i;
  double n = 2, sign =1;
  double ms, sum;
  printf("\nOpenMP \n");
  printf("Starting...\n");
  ms = 0;
  for (i = 0; i < N; i++) {
    start_timer();
    sum = Nilkantha(n, sign);
    ms += stop_timer();
  }
  sum += 3;
  printf("count = %.8f\n", sum);
  printf("avg time = %.5lf ms\n", (ms / N));
  return 0;
}
```

Apéndice 3 - C++

```cpp
/*----------------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
C++
*
*----------------------------------------------------------------------*/
#include <bits/stdc++.h>
#include "utils.h"
const int SIZE = 1000000;
using namespace std;

class Nilkantha {
private:
    double result, sign, n;
public:
    Nilkantha(double s, double a) : sign(s), n(a),result(3) {}
    double getResult() const {
        return result;
    }
    void pi () {
        for (int i = 0; i <= SIZE; i++) {
            result = result+(sign * (4/((n)*(n + 1)*(n + 2))));
            sign = sign * (-1);
            n += 2;
        }
    }
};
int main(int argc, char* argv[]) {
    double ms = 0;
    cout << "Starting..." << endl;
    Nilkantha obj(1, 2);
    for (int i = 0; i < 10; i++) {
        start_timer();
        obj.pi();
        ms += stop_timer();
    }
    cout << "pi = " << (double) obj.getResult() << endl;
    cout << "avg time = "<<setprecision(15)<<(ms / N)<< " ms"<<
endl;
    return 0;
}
```

Apéndice 4 – TBB

```
/*-----------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
C++ con TBB
*
*-----------------------------------------------------------------*/
#include <bits/stdc++.h>
#include <tbb/parallel_reduce.h>
#include <tbb/blocked_range.h>
#include "utils.h"
#define SIZE 10000000
using namespace std;
using namespace tbb;
class NilkanthaTBB {
private:
    double result, sign,n;
public:
    NilkanthaTBB(double s, double a) : sign(s), n(a),result(3) {}
    NilkanthaTBB(NilkanthaTBB &obj,split) : sign(obj.s),
n(obj.a),result(3) {}
    double getResult() const {
        return result;
    }
    void operator() (const blocked_range<int> &r){
        for (int i = r.begin(); i != r.end(); i++) {
            result = result+(sign * (4/((n)*(n + 1)*(n + 2))));
            sign = sign * (-1);
            n += 2;
            }
        }
    }
    void Nilkantha () {
        for (int i = 0; i <= SIZE; i++) {
            for (int i = 0; i <= SIZE; i++) {
                result = result + (sign*(4/((n)*(n + 1)* (n + 2))));
                sign = sign * (-1);
                n += 2;
            }
        }
    }
    void join(const NilkanthaTBB &x) {
```

```
        result += x.result;
    }
};
int main(int argc, char* argv[]) {
    double ms;
    double result;
    cout << "Starting..." << endl;
    ms = 0;
    for (int i = 0; i < N; i++) {
        parallel_reduce(blocked_range<int>(1, 2), obj);
        result = obj.getResult();
        NilkanthaTBB obj(1,2);
        start_timer();
        obj.Nilkantha();
        ms += stop_timer();
        result = obj.getResult();
    }
    cout << "pi = " << result << endl;
    cout << "avg time = " << setprecision(15) << (ms / N) << " ms"
<< endl;
    return 0;
}
```

Apéndice 5 – Java

```java
/*-----------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
Java
*
*-----------------------------------------------------------------*/
import java.util.*;
public class Nilkantha {
    private static final int SIZE = 100;
    private double result,n,sign;
    public Nilkantha() {
        this.n = 2;
        this.sign = 1;
        this.result = 3;
    }
    public double getResult() {
        return result;
    }
    public void calculate() {
        for (int i = 0; i <= 1000000; i++) {
            this.result = this.result + (this.sign * (4 / ((this.n)
* (this.n + 1) * (this.n + 2))));
            this.sign = this.sign * (-1);
            this.n += 2;
        }
    }
    public static void main(String args[]) {
        long startTime, stopTime;
        double acum = 0;
        Nilkantha e = new Nilkantha();
        System.out.printf("Starting...\n");
        for (int i = 0; i < 10; i++) {
            startTime = System.currentTimeMillis();
            e.calculate();
            stopTime = System.currentTimeMillis();
            acum += (stopTime - startTime);
        }
        System.out.printf("Pi = %.8f\n", e.getResult());
        System.out.printf("avg time = %.10f ms\n", (acum / 10));
    }
}
```

Apéndice 6 - Java Threads

```
/*-----------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
Java
*
*-----------------------------------------------------------------*/
public class NilkanthaThreads extends Thread {
    private static final int SIZE = 100_000_000;
    private double result,n,sign;
    private int end;
    public NilkanthaThreads(int blockSize, int n,double s) {
        this.n = n;
        this.sign = s;
        this.result = 0;
        this.end = blockSize;
    }
    public double getResult() {
        return result;
    }
    public void calculate() {
        for (int i = 0; i < end; i++){
            this.result = this.result + (this.sign * (4 / ((this.n)
* (this.n + 1) * (this.n + 2))));
            this.sign = this.sign * (-1);
            this.n += 2;
        }
    }
    public void run() {
        calculate();
    }
    public static void main(String args[]) {
        long startTime, stopTime;
        double acum = 0;
        double res = 0;
        double s = 1;
        NilkanthaThreads threads[] = new
NilkanthaThreads[Utils.MAXTHREADS];
        int blockSize = SIZE / Utils.MAXTHREADS;
        acum = 0;
        System.out.printf("Starting with %d threads\n",
Utils.MAXTHREADS);
```

```
        for (int i = 0; i < Utils.N; i++) {
            res = 0;
            startTime = System.currentTimeMillis();
            for (int j = 1; j < threads.length+1; j++) {
                s = 1;
                if ((j-1) % 2 == 0){
                    s = -1;
                }
                threads[j-1] = new NilkanthaThreads(blockSize,
(2+2*j),1);
                threads[j-1].start();
            }
            try {
                for (int j = 0; j < threads.length; j++) {
                    threads[j].join();
                    res += threads[j].getResult();
                }
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
            stopTime = System.currentTimeMillis();
            acum += (stopTime - startTime);
        }
        res += 3;
        System.out.printf("pi = %f\n", res);
        System.out.printf("avg time = %.5f ms\n", (acum / Utils.N));
    }
}
```

Apéndice 7 - Fork/Join

```java
/*-----------------------------------------------------------------
*
* Programación avanzada: Proyecto final
* Fecha: 30-Nov-2022
* Autor: A01704052 Martin Noboa
* Descripción: Implementacion secuencial de la serie de Nilkantha en
Java Fork Join
*
*-----------------------------------------------------------------*/
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
public class NilkanthaForkJoin extends RecursiveTask<Double> {
    private static final int SIZE = 100_000;
    private static final int MIN = 10_000;
    private double result,sign;
    private int start, end;
    public NilkanthaForkJoin(int start, int end) {
        this.start = start;
        this.end = end;
        this.sign = 1;
        this.result = 0;
    }
    public Double computeDirectly() {
        if(this.start % 2 == 0){
            this.sign = this.sign * -1;
        }
        for (int i = this.start; i < SIZE; i++){
            this.result = this.result + (this.sign * (4.0 /
((this.start) * (this.start + 1) * (this.start + 2))));
            this.sign = this.sign * (-1);
            this.start += 2;
        }
        return result;
    }
    @Override
    protected Double compute() {
        if ((end - start) <= MIN) {
            return computeDirectly();
        } else {
            int mid = start + ((end - start) / 2);
            NilkanthaForkJoin lowerMid = new
NilkanthaForkJoin(start, mid);
            lowerMid.fork();
```

```
            NilkanthaForkJoin upperMid = new NilkanthaForkJoin(mid,
end);
            return (upperMid.compute() + lowerMid.join());
        }
    }
    public double getResult() {
        return result;
    }
    public static void main(String args[]) {
        long startTime, stopTime;
        double acum = 0;
        double res = 0;
        ForkJoinPool pool;
        acum = 0;
        System.out.printf("Starting with %d threads\n",
Utils.MAXTHREADS);
        for (int i = 0; i < Utils.N; i++) {
            res = 0;
            startTime = System.currentTimeMillis();
            pool = new ForkJoinPool(Utils.MAXTHREADS);
            res = pool.invoke(new NilkanthaForkJoin(1, SIZE));
            stopTime = System.currentTimeMillis();

            acum += (stopTime - startTime);
        }
        res +=3;
        System.out.printf("pi = %.8f\n", res);
        System.out.printf("avg time = %.5f ms\n", (acum / Utils.N));
    }
}
```

Apéndice 8 – utils.h C

```c
// ================================================================
//
// 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.
//
// ================================================================
#ifndef UTILS_H
#define UTILS_H
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#define N           10
#define DISPLAY     100
#define TOP_VALUE   10000
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
```

Apéndice 9 – utils.h C++

```
// ================================================================
//
// 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.
//
// ================================================================
#ifndef UTILS_H
#define UTILS_H
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#define N           10
#define DISPLAY     100
#define TOP_VALUE   10000
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
```

Apéndice 10 – Utils.java Java

```
// ======================================================================
//
// File  : Utils.java
// Author: Pedro Perez
// Description: This file contains the implementation of the
functions
//              for initializing integer arrays.
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any non-commercial
// purpose.
//
// ======================================================================
import java.util.Random;
public class Utils {
    private static final int DISPLAY = 100;
    public static final int TOP_VALUE = 10_000;
    public static final Random r = new Random();
    public static final int MAXTHREADS =
Runtime.getRuntime().availableProcessors();
    public static final int N = 5;
    //
======================================================================
    // Initializes an array with random values between 1 and
TOP_VALUE.
    //
    // @param array, an array of integer numbers.
    // @param size, the amount of numbers.
    //
======================================================================
    public static void randomArray(int array[]) {
        for (int i = 0; i < array.length; i++) {
            array[i] = r.nextInt(TOP_VALUE) + 1;
        }
    }
    //
======================================================================
    // Initializes an array with consecutive values of 1 and
TOP_VALUE
    // across all locations.
    //
    // @param array, an array of integer numbers.
    // @param size, the amount of numbers.
```

```
    //
=======================================================================
    public static void fillArray(int array[]) {
        for (int i = 0; i < array.length; i++) {
            array[i] = (i % TOP_VALUE) + 1;
        }
    }
    //
=======================================================================
    // Displays the first N locations in the array.
    //
    // @param array, an array of integer numbers.
    // @param size, the amount of numbers.
    //
=======================================================================
    public static void displayArray(String text, int array[]) {
        int limit = (int) Math.min(DISPLAY, array.length);

        System.out.printf("%s = [%4d", text, array[0]);
        for (int i = 1; i < limit; i++) {
            System.out.printf(",%4d", array[i]);
        }
        System.out.printf(", ..., ]\n");
    }
}
```