Tecnológico de Monterrey

Articulo de Investigación – Serie de Nilakantha

Martin Noboa - A01704052

Pedro Oscar Pérez Murueta

Agosto Diciembre 2023

noviembre 19, 2023

Contenidos

Introducción

Este artículo de investigación describe la serie de Nilakantha 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.

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.

$$\pi \approx 3.1415592 \ldots$$

$$\pi = 3 + 4\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n^{+3})^3 - (2n+3)} = 3 + 4\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+2)(2n+3)(2n+4)}$$

Dada esta serie, así se verían los primeros términos:

N	Aproximación de Pi
1	3
2	3.1667
3	3.1333
4	3.1452

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 o método
     stop_timer();
}
```

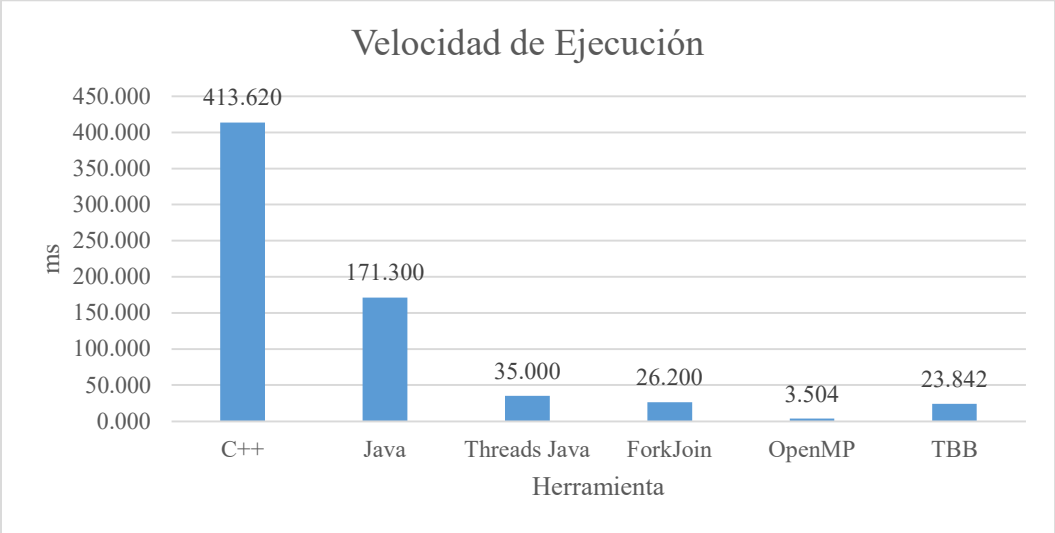
De esta forma medimos el tiempo de ejecución únicamente de la implementación del algoritmo. Se consigue un promedio repitiendo la ejecución N cantidad de veces, durante esta investigación fueron 10. Otro parámetro para considerar fue el número de iteraciones de la serie que se iban a realizar. Para beneficios 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.

Dada la serie especificada en la introducción, el algoritmo implementado para todos los casos fue el siguiente:

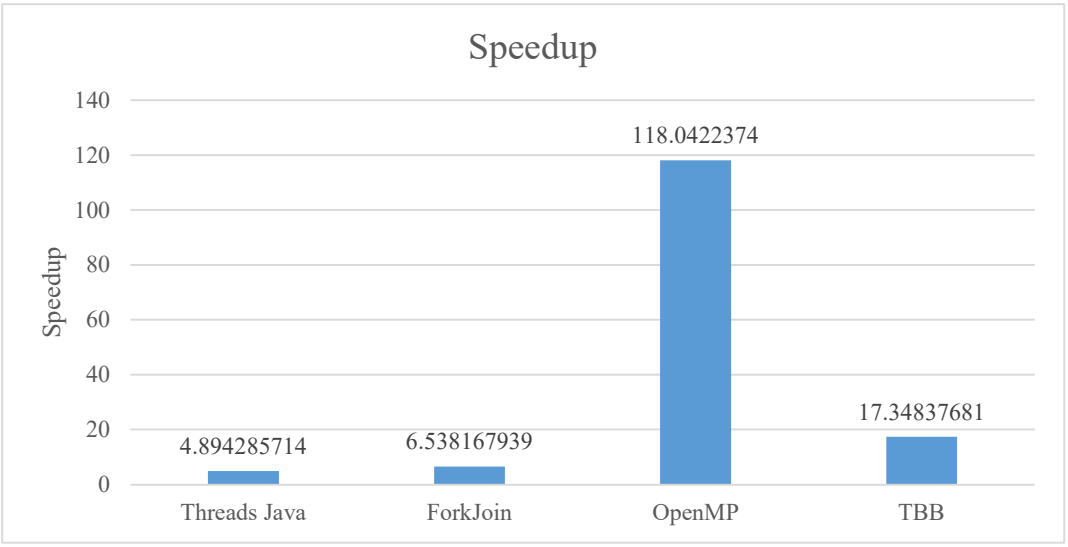
```
sign = 1.0
n = 2.0

for i from 0 to iterations-1:
    pi_approximation = sign * 4.0 / (n * (n + 1) * (n + 2))
    n += 2.0
    sign = -sign
endFor

end
```

Conclusiones

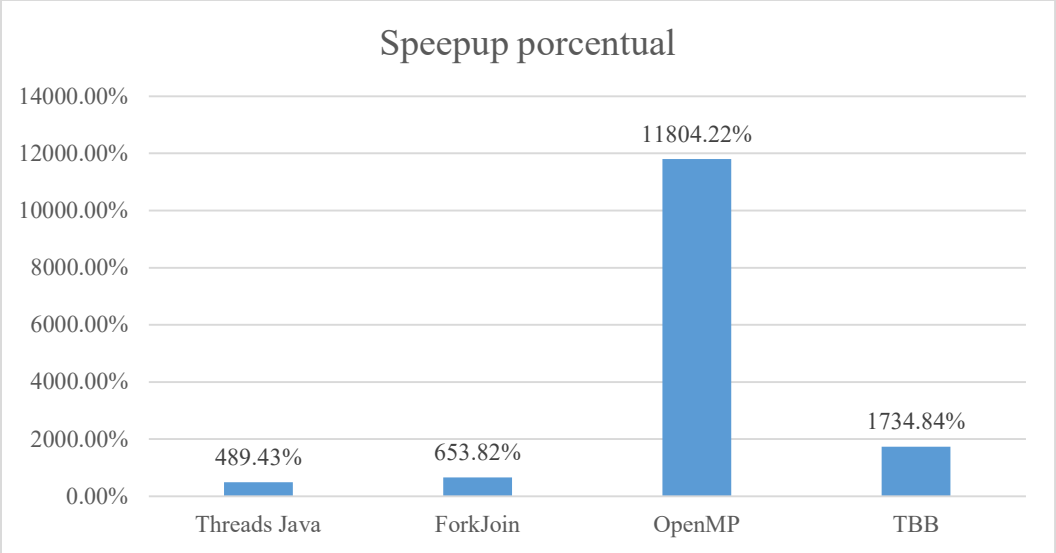
En esta primera grafica comparativa, se encuentra el tiempo de ejecución de los programas secuenciales y los programas con las herramientas para paralelizarlos. Se puede observar una gran diferencia entre los tiempos de ejecución, siendo C++ secuencial el mas tardado y C++ implementando OpenMP el más rápido.



Se puede ver una clara mejora de performance con todas las herramientas. OpenMP muestra el mayo speedup, y esto se debe a que esta herramienta funciona particularmente bien con algoritmos de reducción. Si bien la serie de Nilakantha no es un algoritmo de reducción, sino uno iterativo, la implementación de la serie utiliza la reducción al sumar los resultados parciales de los términos de la sumatoria.



A continuación, podemos ver los speedups en términos porcentuales.



Para finalizar, quiero hacer hincapié en que, si bien una herramienta tuvo mejor desempeño que las demás, fue debido al algoritmo con el que estábamos tratando. Al final del día, cada problema es diferente y habrá diferentes métodos de solución que pueden beneficiarse de una herramienta u otra. El contexto de la solución es muy importante al elegir que herramienta se usara.

Referencias

Kartik Keyan Kant, U. K. (2023). *IQ Open Genius*. Obtenido de Different ways to calculate Pi : https://iq.opengenus.org/different-ways-to-calculate-pi/

Valdebenito, E. (9 de February de 2023). Obtenido de Nilakantha's formula for pi: https://vixra.org/pdf/2302.0056v1.pdf

Apéndices

C Secuencial

```
/*------------------------------------------------------------
* Programación avanzada: Proyecto final
* Fecha: Noviembre 13, 2023
* Autor: A01704052 Martin Noboa
* Archivo: C++ Secuencial
*------------------------------------------------------------*/
#include <iostream>
#include <iomanip>
#include <chrono>
#include <math.h>
#include "utils.h"

using namespace std;
using namespace std::chrono;

#define MAXIMUM 100000000
#define INCREMENT 2

double nilakantha() {
    double piAprox = 0;
    double sign = 1;
    double n = 2;
    for (int i = 0; i < MAXIMUM; i++) {
        piAprox += (sign*4)/((n)*(n+1)*(n+2));
        n+= INCREMENT;
        sign = sign * -1;
    }
    return 3 + piAprox;
}

int main(int argc, char* argv[]) {
     double pi;
     high_resolution_clock::time_point start, end;
     double timeElapsed;

     cout << "Starting...\n";
     timeElapsed = 0;
```

```
for (int j = 0; j < N; j++) {
     start = high_resolution_clock::now();

     pi = nilakantha();

     end = high_resolution_clock::now();
     timeElapsed+=duration<double,std::milli>
                    (end-start).count();
     }
     cout <<"result = "<<fixed<<setprecision(4) << pi << "\n";
     cout    <<"avg    time    =    "<<fixed<<setprecision(3)<<
     (timeElapsed / N) <<  " ms\n";

     return 0;
}
```

TBB

```
/*------------------------------------------------------------
* Programación avanzada: Proyecto final
* Fecha: Noviembre 19, 2023
* Autor: A01704052 Martin Noboa
* Archivo: C++ TBB
*------------------------------------------------------------*/

#include <iostream>
#include <iomanip>
#include <math.h>
#include <chrono>
#include <tbb/blocked_range.h>
#include <tbb/parallel_reduce.h>
#include <tbb/parallel_for.h>
#include "utils.h"

using namespace std;
using namespace std::chrono;
using namespace tbb;

#define MAXIMUM 100000000

// implement your code
class Exercise02{
private:
    double pi;

public:
    Exercise02(): pi(0){}
    Exercise02(Exercise02 &other, split): pi(0){}

    double getResult() const {
        return pi;
    }

    void operator() (const blocked_range<int> &r) {
        double sign = 1.0;
        double n = 2;
        for (int i = r.begin(); i != r.end(); i++) {
            pi += (sign*4)/(i*2+2)/(i*2+3);
            sign = sign * -1;
            n +=2;
        }
```

```
    }

    void join(const Exercise02 &other) {
          pi += other.pi;
     }
};

int main(int argc, char* argv[]) {
     double result;
     // These variables are used to keep track of the execution
time.
     high_resolution_clock::time_point start, end;
     double timeElapsed;

     cout << "Starting...\n";
     timeElapsed = 0;
     for (int j = 0; j < N; j++) {
          start = high_resolution_clock::now();
          Exercise02 obj;
          parallel_reduce(blocked_range<int>(0,
          MAXIMUM),obj);
          result = 3.0 + obj.getResult();

          end = high_resolution_clock::now();
          timeElapsed += duration<double, std::milli>
                         (end - start).count();
     }
     cout<<"result = "<<fixed<<setprecision(5)<<result<< "\n";
     cout<< "avg time = " << fixed << setprecision(3)
           << (timeElapsed / N) <<  " ms\n";

     return 0;
}
```

OpenMP

```
/*-------------------------------------------------------------
-
* Programación avanzada: Proyecto final
* Fecha: Noviembre 19, 2023
* Autor: A01704052 Martin Noboa
* Archivo: OpenMP
*-------------------------------------------------------------*/
#include <iostream>
#include <iomanip>
#include <chrono>
#include <omp.h>
#include <math.h>
#include "utils.h"

using namespace std;
using namespace std::chrono;
#define MAXIMUM 1000000 //1e6
#define INCREMENT 2

double nilakantha(int maximum){
double pi = 0;
     double sign = 1;
     double n = 2;
     #pragma omp parallel for shared(maximum)
     reduction(+:result)
     for (int i = 0; i < MAXIMUM; i++) {
        pi += (sign*4)/((n)*(n+1)*(n+2));
        n+= INCREMENT;
        sign = sign * -1;
    }
    return pi;
}

int main(int argc, char* argv[]) {
     double pi;
     high_resolution_clock::time_point start, end;
     double timeElapsed;

     cout << "Starting...\n";
     timeElapsed = 0;
     for (int j = 0; j < N; j++) {
          start = high_resolution_clock::now();
          pi = 3 + nilakantha(MAXIMUM);
```

```
        end = high_resolution_clock::now();
        timeElapsed += duration<double, std::milli>(end -
start).count();
    }
    cout << "result = " << fixed << setprecision(5) << pi <<
"\n";
    cout << "avg time = " << fixed << setprecision(3) <<
(timeElapsed / N) <<  " ms\n";

    return 0;
}
```

Java Secuencial

```
/*------------------------------------------------------------
-
* Programación avanzada: Proyecto final
* Fecha: Noviembre 13, 2023
* Autor: A01704052 Martin Noboa
* Archivo: Java Secuencial
*------------------------------------------------------------*/

import java.lang.Math;
public class BaseJava {
     private static final int MAX = 100_000_000;
     public BaseJava() {
     }
     public long calculate() {
          long pi = 0.0, sign = 1.0, n = 2.0;
          for (int i = 0; i < this.MAX; i++) {
            pi += (sign * 4)/ (n*(n+1)*(n+2));
            sign = sign * -1;
            n += 2;
          }
          return 3 + pi;
     }
     public static void main(String args[]) {
          long startTime, stopTime;
          long pi=0, elapsedTime;

          BaseJava obj = new BaseJava();
          elapsedTime = 0;
          System.out.printf("Starting...\n");
          for (int i = 0; i < Utils.N; i++) {
               startTime = System.currentTimeMillis();

               pi = obj.calculate();

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

Java Threads

```
/*-------------------------------------------------------------
-
* Programación avanzada: Proyecto final
* Fecha: Noviembre 13, 2023
* Autor: A01704052 Martin Noboa
* Archivo: Java Threads
*-------------------------------------------------------------*/

public class Threads extends Thread {
     private static final int SIZE = 100_000_000;
     private int start, end;
     public double pi, sign, n;

     public Threads(int start, int end) {
        this.start = start;
        this.end = end;
        this.n = start + 2;
        if (start+2 % 2 == 0){
            this.sign = 1.0;
        }else{
            this.sign = -1.0;
        }
        this.pi = 0.0;
     }

    public double getResult() {
        return pi;
    }
```

```
public void run() {
   for (int i = this.start; i < this.end; i++) {
     this.pi += (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 result = 3, elapsedTime;
     int blockSize;
     Threads threads[];
     blockSize = SIZE / Utils.MAXTHREADS;
     threads = new Threads[Utils.MAXTHREADS];

     elapsedTime = 0;
     System.out.printf("Starting...\n");
     for (int i = 0; i < Utils.N; i++) {
          startTime = System.currentTimeMillis();
          for (int j = 0; j < threads.length; j++) {
               threads[j] = new Threads(j * blockSize,
               (j + 1) * blockSize);
          }
          for (int j = 0; j < threads.length; j++) {
           threads[j].start();
       }
```

```
            double sign = 1.0;
            for (int j = 0; j < threads.length; j++) {
                try {
                    threads[j].join();
                    result += sign* threads[j].getResult();
                    sign = sign * -1;
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }

               stopTime = System.currentTimeMillis();

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

Fork Join

```
/*-----------------------------------------------------------
* Programación avanzada: Proyecto final
* Fecha: Noviembre 13, 2023
* Autor: A01704052 Martin Noboa
* Archivo: Java Fork Join
*-----------------------------------------------------------*/

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

public class ForkJoin extends RecursiveTask<Double> {
     private static final int SIZE = 100_000_000;
     private static final int MIN = 10_000;
     private int start, end;
    public double sign, n;

     public ForkJoin(int start, int end) {
          this.start = start;
          this.end = end;
        this.start = start;
        this.end = end;
        this.n = start + 2;
        if (start+2 % 2 == 0){
            this.sign = 1.0;
        }else{
            this.sign = -1.0;
        }
     }

     public double calculate() {
          double pi = 0;
        for (int i = this.start; i < this.end; i++) {
            pi        +=        (this.sign        *        4)/
(this.n*(this.n+1)*(this.n+2));
            this.sign = this.sign * -1;
            this.n += 2;
        }
        return pi;
    }


     @Override
```

```
     protected Double compute() {
          if ( (end - start) <= MIN ) {
               return calculate();
          } 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();
          }
     }

     public static void main(String args[]) {
          long startTime, stopTime;
          double result = 0, elapsedTime;
          int blockSize;
          ForkJoinPool pool;

          elapsedTime = 0;

          System.out.printf("Starting...\n");
          for (int i = 0; i < Utils.N; i++) {
               startTime = System.currentTimeMillis();
            pool = new ForkJoinPool(Utils.MAXTHREADS);
               result = 3 + pool.invoke(new ForkJoin(0, SIZE));

               stopTime = System.currentTimeMillis();

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