

Aproximación de Pi mediante la serie de Euler

Roberto Carlos Guzmán Cortés (A01702388)
Tecnológico de Monterrey, Campus Querétaro
A01702388@tec.mx

30 de Noviembre de 2022

Resumen

El presente artículo de investigación tiene por finalidad aplicar el aprendizaje obtenido en el curso de multiprocesadores durante el semestre Agosto-Diciembre 2022, todo esto se llevará a cabo mediante la resolución de un problema usando el paradigma concurrente. La resolución de dicho problema involucró 4 lenguajes de programación distintos para las implementaciones paralelas y 6 tecnologías distintas para las concurrentes y pueden encontrarse tanto en el [Apéndice](#) de este artículo como en el [repositorio oficial](#).

Las métricas correspondientes al tiempo de ejecución y *speedup* servirán como parámetro clave para evaluar el comportamiento de las diversas tecnologías usadas en la resolución del mismo.

Introducción

Los matemáticos contemporáneos han realizado numerosos debates y aportaciones en torno a lo que hoy conocemos como Pi. Su importancia reside en que es una constante que se halla presente en numerosas áreas de estudio como la astrofísica, geometría, física relativista, teoría de números, entre otros (Stewart, I. 2016).

A pesar de que algunos matemáticos han denominado a Pi como un fractal, lo más que sabemos hoy en día en el ámbito académico es que Pi es considerado como un número irracional trascendental del cual se conocen billones (*trillions*) de sus cifras (Iwao, E. H. 2022).

Existen numerosos algoritmos para calcular aproximaciones de tan famosa constante, en el presente reporte de investigación nos enfocaremos en el que lleva por nombre “La serie de Euler” (también conocida como serie de Leibniz), en la cual se enuncia lo siguiente:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots = \frac{\pi}{4}$$

Ecuación 1. Algoritmo desglosado como sumas sucesivas

La suma infinita de los inversos de todos los números impares en los cuales se intercala el signo positivo y negativo dan como resultado la cuarta parte de Pi (Serie de Leibniz, Wikipedia).

En formato de sumatoria gracias al trabajo de James Gregory (Serie de Leibniz, Wikipedia) quedaría expresado de la siguiente manera:

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

Ecuación 2. Algoritmo expresado en formato de sumatoria

En los siguientes apartados se apreciará una implementación de dicho algoritmo usando diferentes tecnologías asociadas a lenguajes de programación y sus respectivas métricas.

Desarrollo

Dada la naturaleza iterativa del algoritmo a implementar, se optó por usar un ciclo *for* cuyos límites expresados en notación de intervalo serían los siguientes:

$$[0, Size)$$

dónde *Size* representa el número máximo de iteraciones a realizar y puede variar en el siguiente rango expresado también a manera de intervalo:

$$0 \leq Size \leq 100,000,000$$

Se espera que a mayor número de iteraciones se logre mayor exactitud (*accuracy*) y por ende el margen de error (*percent error*) disminuya.

El pseudocódigo de dicha implementación sería el siguiente:

```
PROCEDURE calculatePI(size):
BEGIN
    pi = 0
    FOR (i = 0, i < size, STEP 1):
    BEGIN
        pi = pi + (exp(-1, i))*4/((2*i)+1)
    END
    RETURN pi
END
```

Figura 1. Pseudocódigo de la implementación del algoritmo

Cada tecnología en la cual se eligió implementarlo se ejecutó 10 veces para obtener su tiempo de ejecución promedio y tras analizar los resultados se resaltaron sus pros y contras los cuales se explicarán detalladamente en cada uno de los siguientes apartados.

Implementación en C

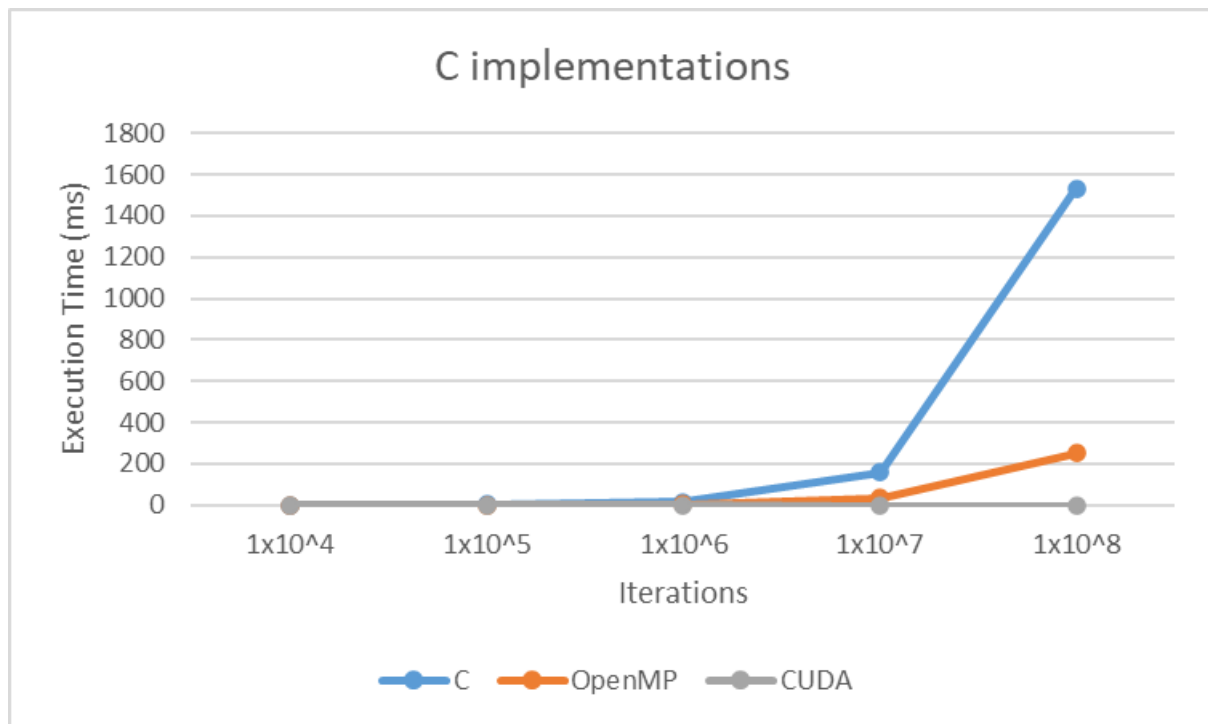
El lenguaje de programación C tiene por características el ser fuertemente tipado, sigue el paradigma estructurado, procedural e imperativo y su sintaxis tiende a parecer más rígida comparada con la de otros lenguajes debido a la amplia cantidad de reglas bajo las cuales se diseñó y que son necesarias para su uso (Lucas, J. 2020).

En la siguiente tabla podemos observar los tiempos de ejecución correspondientes a las implementaciones hechas en dicha tecnología, apreciándose un tiempo de ejecución mayor en la implementación serial (Columna C) y tiempos menores en las implementaciones concurrentes (OpenMP y CUDA)

C					
Iterations	C	OpenMP	CUDA	OpenMP-speedup	CUDA-speedup
1x10⁴	0.4196	0.2921	0.0041	1.436494351	102.3414634
1x10⁵	1.7999	0.4995	0.0043	3.603403403	418.5813953
1x10⁶	15.4674	3.6203	0.004	4.272408364	3866.85
1x10⁷	157.112	35.1837	0.0031	4.465476911	50681.29032
1x10⁸	1,532.69	250.3	0.0029	6.123403116	528513.0345

Tabla 1. Tiempos de ejecución de implementaciones en C (ms)

La siguiente gráfica nos permite apreciar detalladamente las diferencias entre los tiempos de ejecución de las distintas implementaciones, resaltando CUDA por sus tiempos de ejecución los cuales tienden a ser de cero.



Gráfica 1. Comparación de implementaciones en C

Implementación serial

Las ventajas de dicha implementación residen en que es fácil expresar un ciclo *for* y la eficiencia que tiene el lenguaje C en el uso del hardware lo convirtieron en la implementación serial más rápida. La siguiente función realizó el cálculo de Pi de forma iterativa tal como se explicó líneas arriba, añadiendo el uso de un acumulador y retornando dicho valor:

```
double calculate_pi(int size){
    double pi = 0;

    for(int i=0; i<size; i++){
        pi += pow(-1, i)*4 / ((i*2)+1);
    }

    return pi;
}
```

Figura 2. Cálculo de Pi serial en C

Entre las ventajas podríamos mencionar que el uso de apuntadores requiere cierto grado de experiencia en el lenguaje, además de que si no se tiene experiencia con lenguajes fuertemente tipados se corre el riesgo de expresar el valor de Pi como entero (3).

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando $size = 100,000,000$.

```
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$ gcc eulerpi_serial_c.c -lm
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$ ./a.out
Starting...
Calculated value of PI = 3.141593
Real PI = 3.141593
Percent error = 0.000000
avg time = 1532.68780 ms
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$
```

Figura 3. Implementación serial en C

Implementación concurrente usando OpenMP

OpenMP es una librería de código abierto que se puede utilizar en C/C++ y Fortran, su rendimiento es tan eficiente que inclusive tecnologías como ONNX la usan. La implementación realizada es tan similar que sólo fue necesario agregar la línea de código que aparece en color verde; en ella indicamos que todas los cores compartan la variable $size$ y concentren la sumatoria final en la variable pi .

```
double calculate_pi(int size){
    double pi = 0;

    #pragma omp parallel for shared(size) reduction(+:pi)
    for(int i=0; i<size; i++){
        pi += pow(-1, i)*4 / ((i*2)+1);
    }

    return pi;
}
```

Figura 4. Cálculo concurrente de Pi usando OpenMP

Entre las numerosas ventajas de usar esta tecnología encontramos que está incluida en el mismo compilador (gcc) y no requiere de la instalación de componentes adicionales. Por otro lado, al momento de codificar nuestra solución sólo se requirió de una línea de código adicional para paralelizar el ciclo *for*.

Entre las desventajas encontramos que se requiere de un conocimiento previo para obtener un buen *speedup* tales como saber que variables son compartidas (*shared*) o cuáles variables deberán estar involucradas en el proceso de reducción (*reduction*). En el caso contrario, de no tener suficiente experiencia con OpenMP, se corre el riesgo de generar los típicos errores que ocurren al momento de hacer programas concurrentes tales como la generación de cuellos de

botella, incoherencias de caché, *race conditions*, *overhead* provocado por uso inadecuado de hilos, etc. y que terminan provocando un *speedup* cercano a cero.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando $size = 100,000,000$.

```
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/c$ gcc eulerpi_parallel_omp.c -lm -fopenmp
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/c$ ./a.out
Starting...
Calculated value of PI = 3.140593
Real PI = 3.141593
Percent error = 0.000000
avg time = 0.25030 ms
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/c$
```

Figura 5. Implementación concurrente en C usando OpenMP

Implementación concurrente usando CUDA

La característica más resaltante de esta implementación reside en que sus tiempos de ejecución siempre tendieron a cero, además de que comúnmente siempre se tendrá al menos 50 veces más unidades de procesamiento en una GPU comparada con un CPU convencional (640 vs 12 en el caso de mi ordenador). La implementación de la función que calcula Pi repartió el trabajo muchas iteraciones entre muchos hilos de trabajo, además de que usó *caching* para que el resultado de cada bloque de hilos se guardase en un arreglo y al pasar a la CPU todo se simplificó en una sola variable. La apreciación de lo que ocurrió dentro de la GPU lo podemos visualizar en la siguiente imagen:

```

__global__ void calculate_pi(double *result) {
    __shared__ double cache[THREADS];

    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int cacheIndex = threadIdx.x;

    double acum = 0;
    while (tid < SIZE) {
        acum += pow(-1, tid)*4 / ((tid*2)+1);
        tid += blockDim.x * gridDim.x;
    }

    cache[cacheIndex] = acum;

    __syncthreads();

    int i = blockDim.x / 2;
    while (i > 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0) {
        result[blockIdx.x] = cache[cacheIndex];
    }
}

```

Figura 6. Cálculo concurrente de Pi usando CUDA

Una de las grandes desventajas también reside a nivel de hardware, ya que si no se cuenta con una tarjeta gráfica nvidia, no será posible usar CUDA. Por otro lado, si el desarrollador no tiene buenos conocimientos del uso de apuntadores y la manera en que trabajan las localidades de memoria de una GPU, no podrá tener progreso en la resolución de errores de sintaxis ni lógicos, aunado al hecho de que es necesario saber cómo funcionan las operaciones relacionadas con *reduction* y *caching* para hacer concurrencia en sumatorias.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando *size* = 100,000,000.

```

A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$ nvcc eulerpi_parallel_cuda.cu -lm
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$ ./a.out
Starting...
Calculated PI = 3.141593
Real PI = 3.141593
Percent error = 0.000000
avg time = 0.00290 ms
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C$

```

Figura 7. Implementación concurrente en C usando CUDA

Implementación en C++

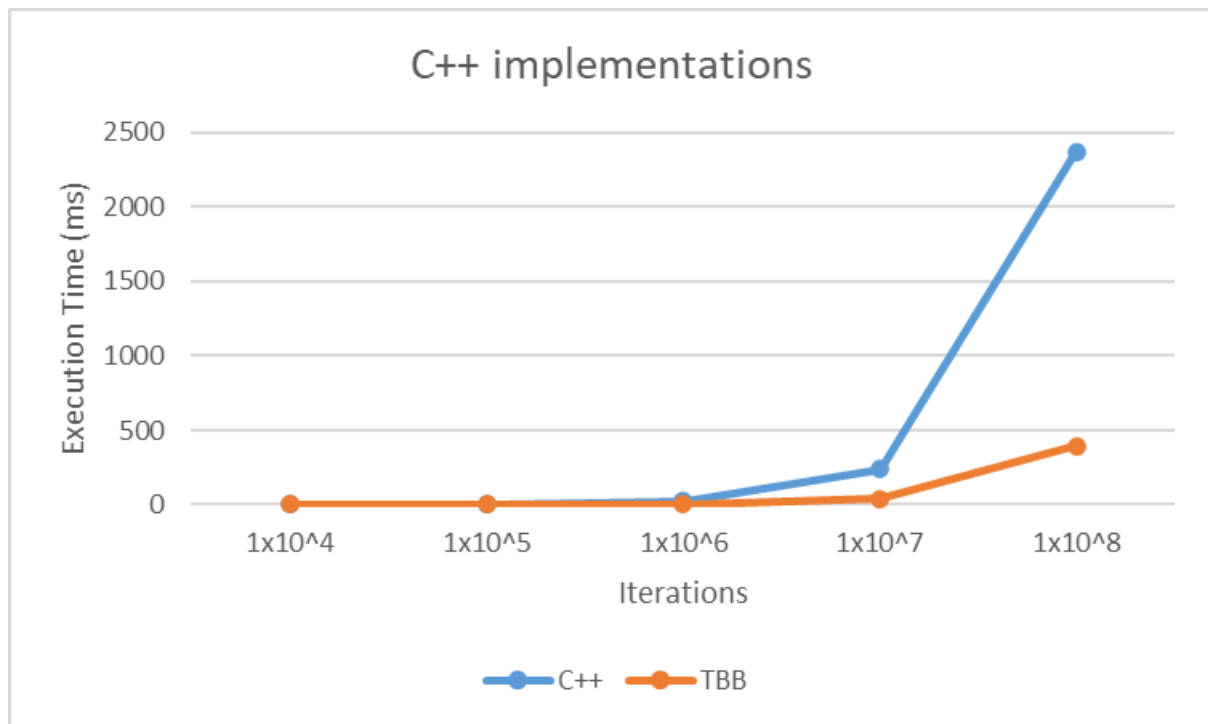
El lenguaje C++ estrictamente hablando no es más que un superconjunto del lenguaje C, en dónde se ha añadido soporte para el paradigma orientado a objetos y demás utilidades, ejemplo de ello es que en algunos compiladores existe herencia múltiple.

En la siguiente tabla podremos apreciar los tiempos de ejecución correspondientes a la implementación serial (C++) y concurrente (TBB).

C++			
Iterations	C++	TBB	TBB-speedup
1x10 ⁴	0.374	0.1663	2.248947685
1x10 ⁵	2.4908	0.582	4.279725086
1x10 ⁶	23.7479	4.3153	5.503186337
1x10 ⁷	236.833	41.7564	5.67177726
1x10 ⁸	2373.42	392.131	6.05262017

Tabla 5. Tiempos de ejecución de implementaciones en C++ (ms)

En el siguiente gráfico podemos apreciar que el *speedup* aumenta conforme aumenta el número de iteraciones, resaltando así los beneficios del uso de tecnologías concurrentes para grandes cantidades de iteraciones.



Gráfica 2. Comparación de implementaciones en C++

Implementación serial

Una de las principales ventajas de usar C++ reside en que si el desarrollador está acostumbrado a el desarrollo en lenguaje C, podrá disfrutar de los beneficios del mismo con un enfoque orientado a objetos. Por otro lado, el desarrollador puede eficientar sus implementaciones tanto como lo desee mediante la creación de tipos de datos abstractos, macros, constantes, uso de memoria dinámica, etc. La estrategia para calcular Pi de manera serial en C++ consistió en reusar el código hecho en C para ahora ponerlo dentro del método de una clase:

```
void calculate(){
    result = 0;

    for(int i=0; i<size; i++){
        result += pow(-1, i)*4 / ((i*2)+1);
    }
}
```

Figura 8. Cálculo serial de Pi usando C++

Entre las desventajas de realizar implementaciones en esta tecnología encontramos que su sintaxis es de las más complejas comparada con los demás lenguajes de programación aquí empleados. Por otro lado, si no se tiene un buen *background* de lenguaje C, se corre el riesgo

de cometer errores en el uso de memoria dinámica, acceso a localidades de memoria no permitidos, etc.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando $size = 100,000,000$.

```
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$ g++ eulerpi_serial_cpp.cpp -lm
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$ ./a.out
Starting...
Calculated value of PI = 3.14159
Real PI = 3.14159
Percent error = 3.18325e-09
avg time = 2373.42 ms
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$
```

Figura 9. Implementación serial en C++

Implementación concurrente usando TBB

Intel Threading Building Blocks es una tecnología que permite ser combinada con OpenMP, sin embargo durante esta investigación decidió no hacerse así debido a que se enfocó más en medir su rendimiento de manera individual. Otra de las ventajas que podemos encontrar en el uso de la misma es que al igual que OpenMP requiere de pocas líneas de código para su implementación y no modifica demasiado la estructura de nuestro programa original. La estrategia para calcular Pi de forma concurrente usando TBB consistió en añadir un método extra que ejecutaría un reduction y el método principal encargado de calcular Pi se tendría que acomodar a manera de notación de rango tal como lo establece TBB:

```
void operator()(const blocked_range<int> &r) {
    for(int i=r.begin(); i!=r.end(); i++){
        result += pow(-1, i)*4 / ((i*2)+1);
    }
}

void join(const CalculatePI &x) {
    result += x.result;
}
```

Figura 10. Cálculo concurrente de Pi usando TBB

Entre las desventajas podemos mencionar que es necesario tener experiencia previa en el uso de la misma debido a que se necesitan funciones adicionales para operaciones como *reduction* y comprender cuáles de sus funciones se acoplan a la naturaleza del problema que queremos resolver.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando $size = 100,000,000$ (Nota: el valor calculado de Pi necesitaba dividirse entre 10 para ajustarlo; esto último ya se realizó en el código fuente original).

```

A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$ g++ eulerpi_parallel_tbb.cpp -lm -ltbb
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$ ./a.out
Starting...
Calculated value of PI = 31.4159
Real PI = 3.14159
Percent error = 9
avg time = 392.131 ms
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/C++$ █

```

Figura 11. Implementación concurrente en C++ usando TBB

Implementación en Java

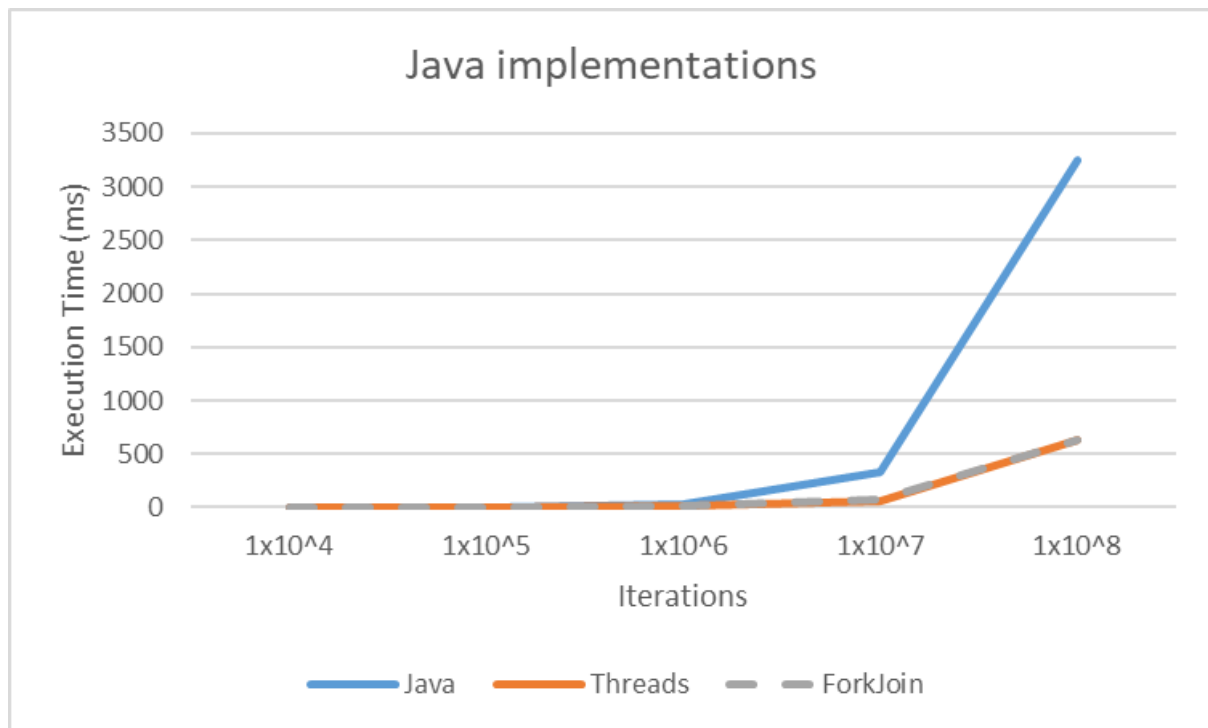
La sintaxis del lenguaje de programación Java surgió a través de un proceso de *cleanup* a la sintaxis del lenguaje C++, todo ello con la finalidad de crear código más “leíble” e intuitivo a la hora de revisarlo. Hoy en día le podemos encontrar en distintos ámbitos que van desde redes y servidores hasta en dispositivos móviles, ya que por su enfoque orientado a objetos y su alta portabilidad tuvo un auge en décadas pasadas.

En la siguiente tabla podemos apreciar cómo las tecnologías concurrentes (Java Thread y ForkJoin) tuvieron un rendimiento muy similar:

Java					
Iterations	Java	Threads	ForkJoin	Threads-speedup	ForkJoin-speedup
1x10 ⁴	0.5	0.9	3.9	0.555555556	0.128205128
1x10 ⁵	3.5	2.5	4.4	1.4	0.795454545
1x10 ⁶	33.1	9.8	10.5	3.37755102	3.152380952
1x10 ⁷	323.4	65.8	70	4.914893617	4.62
1x10 ⁸	3242.9	629.5	631	5.151548848	5.139302694

Tabla 3. Tiempos de ejecución de implementaciones en Java (ms)

La siguiente gráfica nos confirma lo mencionado líneas atrás, Java Threads y ForkJoin tienen rendimientos muy similares:



Gráfica 3. Comparación de implementaciones en Java

Implementación serial

Una de las principales ventajas del lenguaje de esta implementación reside en que java es interpretado y compilado, por lo que su rendimiento se ajustará mucho al uso eficiente del software y hardware del ordenador en que se ejecute, además de que su sintaxis es muy entendible comparado con otros lenguajes de programación orientados a objetos. La manera en como se implementó la sumatoria fue muy parecido a la de C++, se agregó un método a una clase y se iteró al igual que en la implementación que se hizo en C:

```
public void calculatePi(){
    result = 0;

    for(int i=0; i<SIZE; i++){
        result += Math.pow(-1, i)*4 / ((i*2)+1);
    }
}
```

Figura 12. Cálculo serial de Pi usando Java

Entre las desventajas se podría mencionar que es necesario tener conocimiento previo del paradigma orientado a objetos para poder disfrutar de los beneficios que java tiene para los desarrolladores.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando *size* = 100,000,000.

```
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ javac Eulerpi_serial_java.java
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ java Eulerpi_serial_java
Starting...
Calculated value of PI = 3.141592643589326
Real PI = 3.141592653589793
Percentage error = 3.1832475510938223E-7
avg time = 3242.9ms
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$
```

Figura 13. Implementación serial en Java

Implementación concurrente usando Threads

Los threads de Java fueron de las primeras herramientas con las que contaron los desarrolladores para hacer programas concurrentes. La estrategia para realizar el cálculo de Pi consistió en indicar de forma manual cuál sería el índice a partir del cuál cada hilo comenzaría a realizar el cálculo de dicha constante, así como el índice correspondiente al último cálculo. El procedimiento descrito anteriormente se colocó dentro del método que nos obliga la clase Threads a implementar cada que heredamos de ella:

```
public void run() {
    result = 0;
    for (int i = start; i < end; i++) {
        result += Math.pow(-1, i)*4 / ((i*2)+1);
    }
}
```

Figura 14. Cálculo concurrente de Pi usando Java Threads

Las ventajas que el uso de esta tecnología presenta son el brindar libertad al desarrollador para implementar soluciones tan eficientes como lo permita el lenguaje de programación java.

Las desventajas a mencionar residen en el hecho de que se necesita conocimiento del uso de programación orientada a objetos, uso de la interfaz de hilos y conocimiento de la naturaleza del problema a resolver para evitar los típicos problemas que una implementación concurrente puede llegar a tener.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando *size* = 100,000,000.

```

● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ javac Eulerpi_parallel_threads.java
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ java Eulerpi_parallel_threads
Starting with 8 threads...
Calculated value of PI = 3.1415926435898798
Real PI = 3.141592653589793
Percentage error = 3.183071277685476E-7
avg time = 629.5ms
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ █

```

Figura 15. Implementación concurrente en Java usando Threads

Implementación concurrente usando ForkJoin

Debido a que no existía una estandarización en las primeras implementaciones concurrentes realizadas en Java Threads, se optó por un estándar y fue cuando nació la interfaz ForkJoin. La implementación del cálculo de Pi se dividió en dos métodos muy importantes, el primero de ellos `compute()` realiza la división de tareas en entre distintos hilos si el tamaño de iteraciones es mayor a cierta constante, de lo contrario, la tarea será calculada de manera serial por cada uno de los hilos respetando su respectivo rango, para finalmente retornar el valor que acumularon usando `Join()`, todo acorde al estándar de la interfaz `RecursiveTask`.

```

protected Double computeDirectly() {
    double result = 0;
    for (int i = start; i < end; i++) {
        result += Math.pow(-1, i)*4 / ((i*2)+1);
    }
    return result;
}

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

```

Figura 16. Cálculo concurrente de Pi usando Java ForkJoin

Las ventajas que este tipo de tecnología presenta están enfocadas a la automatización del uso de hilos, la cual está delegada a la interfaz que se implementará con la clase, aunado al hecho de que su enfoque “*work-stealing*” permite hacer un uso eficiente del hardware del ordenador en el cual se ejecutará.

Sus desventajas al igual que en el uso de otras tecnologías concurrentes se resumen a que se requiere conocimiento en el uso de la herramienta para poder sacarle el máximo provecho

posible, por otro lado, un desarrollador experimentado se vería limitado a lo que ofrece dicha interfaz.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando $size = 100,000,000$.

```

● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ javac Eulerpi_parallel_forkjoin.java
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$ java Eulerpi_parallel_forkjoin
Starting with 8 threads
Calculated value of PI = 3.1415926435897163
Real PI = 3.141592653589793
Percentage error = 3.183123297424267E-7
avg time = 631.0ms
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Java$

```

Figura 17. Implementación concurrente en Java usando ForkJoin

Implementación en Golang

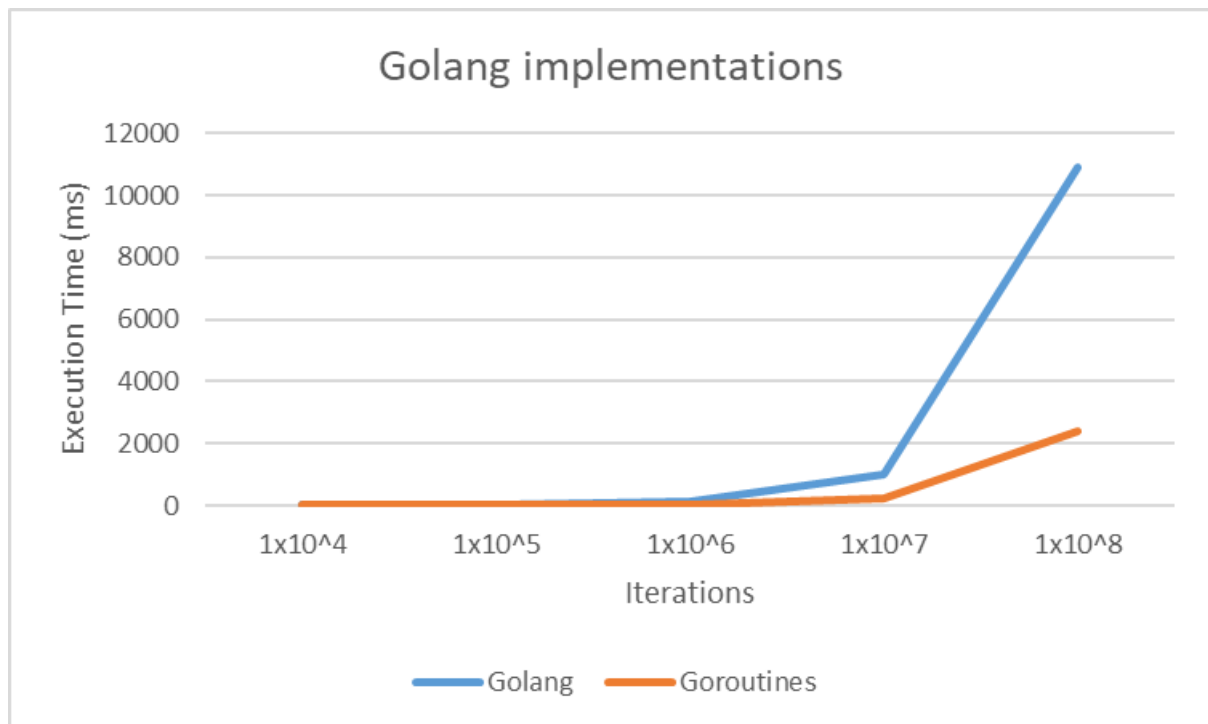
El lenguaje de programación Golang (también conocido como Go) fue creado por Google en 2007 pero liberado hasta 2012 para su uso por parte de la comunidad de desarrolladores. Está enfocado principalmente a la realización de un uso eficiente de los recursos de infraestructura de hardware con los que cuentan las aplicaciones que corren en entornos web tales como servidores propios, servidores en la nube, componentes de software enfocados a microservicios, entre otros (Bai, Y. 2022).

En la siguiente tabla podemos observar los tiempos de ejecución registrados tanto para la implementación serial como la concurrente:

Golang			
Iterations	Golang	Goroutines	Goroutines-speedup
1x10 ⁴	0.9395	0.3344	2.809509569
1x10 ⁵	8.9162	3.136	2.84317602
1x10 ⁶	112.0645	22.6192	4.954397149
1x10 ⁷	1018.3721	226.2636	4.500821608
1x10 ⁸	10897.6999	2404.7984	4.531648017

Tabla 4. Tiempos de ejecución de implementaciones en Golang (ms)

A diferencia de las demás tecnologías concurrentes, las *goroutines* no parecieron tener un *speedup* muy elevado (mayor a 5) tal como veníamos acostumbrados a verlo. La siguiente gráfica nos permitirá apreciar lo expresado en la anterior tabla:



Gráfica 4. Comparación de implementaciones en Golang

Implementación serial

La estrategia que se siguió para realizar la implementación en Golang fue exactamente la misma que en la implementación hecha en lenguaje C, todo esto debido a que se siguió el mismo paradigma y no el orientado a objetos que se logra mediante el uso de estructuras (*structs*). Una particularidad a resaltar en esta implementación es el uso del operador de asignación e inferencia (*type inference*) es cual consta de dos puntos seguido del signo de igual `:=` el cual indica que la variable `pi` tendrá un valor de `0.0` y será de tipo `float64`.

```
func calculatePi(size int) float64{
    pi:=0.0

    for i:=0; i<size; i++){
        pi += math.Pow(-1, float64(i))*4/((float64 (i)*2)+1)
    }

    return pi
}
```

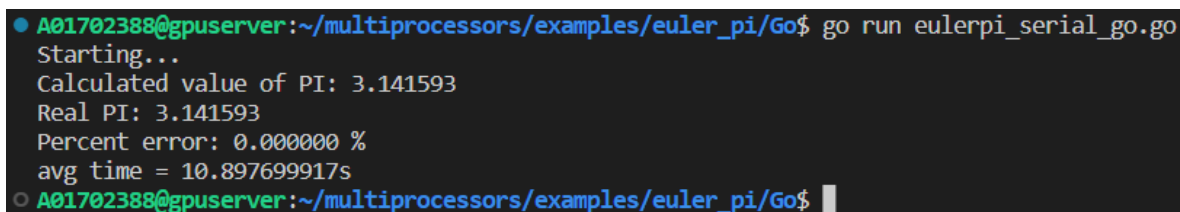
Figura 18. Cálculo serial de Pi usando Golang

Las ventajas del uso del lenguaje de programación Go residen en que retoma elementos de otros lenguajes de programación que han agradado a la comunidad de desarrolladores tales como la sintaxis de java, el uso de apuntadores de C/C++, el retorno de múltiples parámetros

en funciones como lo es en python y perl, etc. Otro elemento a favor de Golang es que tiene gran soporte por parte de la comunidad de desarrolladores y esto se ve reflejado a través de la amplia variedad de paquetes (librerías) de los que se puede disponer.

Entre las principales desventajas podemos mencionar que es necesario aprender algunas reglas adicionales para comprender su sintaxis y que el rendimiento de las soluciones implementadas en dicha tecnología dependerá directamente del grado de expertise que se tenga.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando *size* = 100,000,000.



```
● A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Go$ go run eulerpi_serial_go.go
Starting...
Calculated value of PI: 3.141593
Real PI: 3.141593
Percent error: 0.000000 %
avg time = 10.897699917s
○ A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Go$
```

Figura 19. Implementación serial en Golang running

Implementación concurrente usando Goroutines

La estrategia que se siguió consistió en definir al igual que en la implementación de Java Threads un índice de inicio y uno de final para la operación de cada *Goroutine*, posteriormente de reduciría a una sola variable el contenido el resultado obtenido por cada una de ellas:

```

func calculatePi(size int) float64 {
    n_threads := runtime.NumCPU()
    block := size/n_threads

    ch := make(chan float64)
    for i := 0; i < n_threads; i++ {
        if i != n_threads-1{
            go term(ch, i*block, (i+1)*block)
        } else{
            go term(ch, i*block, size)
        }
    }
    f := 0.0
    for k := 0; k < n_threads; k++ {
        f += <-ch
    }
    return f
}

func term(ch chan float64, start int, end int) {
    acum := 0.0
    for i:=start; i<end; i++){
        acum += math.Pow(-1, float64(i))*4 / ((2*float64(i)) + 1)
    }
    ch <- acum
}

```

Figura 20. Cálculo concurrente de Pi usando Goroutines

Los beneficios del uso de este tipo de tecnología se enfocan en que retoma los beneficios del uso de procesos tales como uso de memoria compartida, intercambio de mensajes (o señales), entre otros. Por lo que se cuenta con una amplia gama de opciones para implementar una solución (paradigma orientado a objetos, estructural, funcional, etc.) y con ello sacarle el máximo provecho a los recursos de hardware en los que se ejecute.

Una de las desventajas reside en que la efectividad de la solución implementada dependerá del grado de experiencia que el desarrollador posea en el dominio de la herramienta, así como su grado de experiencia en el uso de procesos y señales.

En la siguiente imagen podemos apreciar los resultados obtenidos al ejecutar dicha implementación usando *size* = 100,000,000.

```

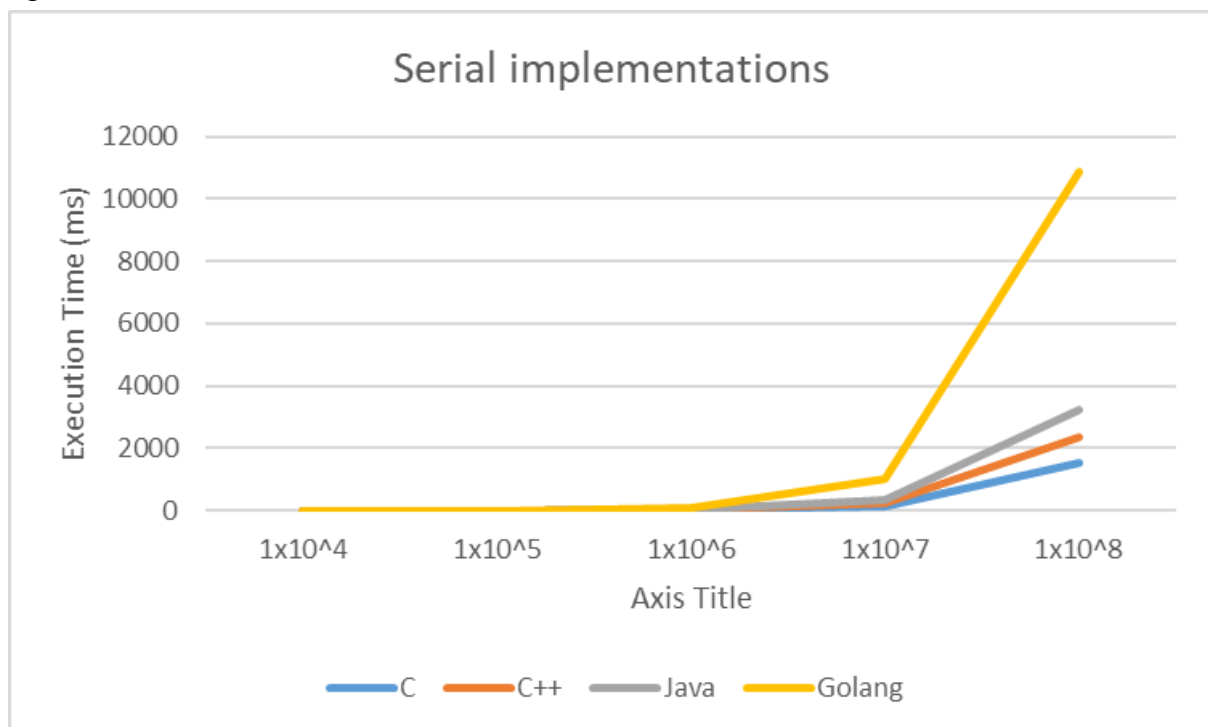
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Go$ go run eulerpi_parallel_goroutines.go
Starting with 8 threads...
Calculated value of PI: 3.141593
Real PI: 3.141593
Percent error: 0.000000 %
avg time = 2.404798427s
A01702388@gpuserver:~/multiprocessors/examples/euler_pi/Go$ 

```

Figura 21. Implementación concurrente en Golang usando Goroutines

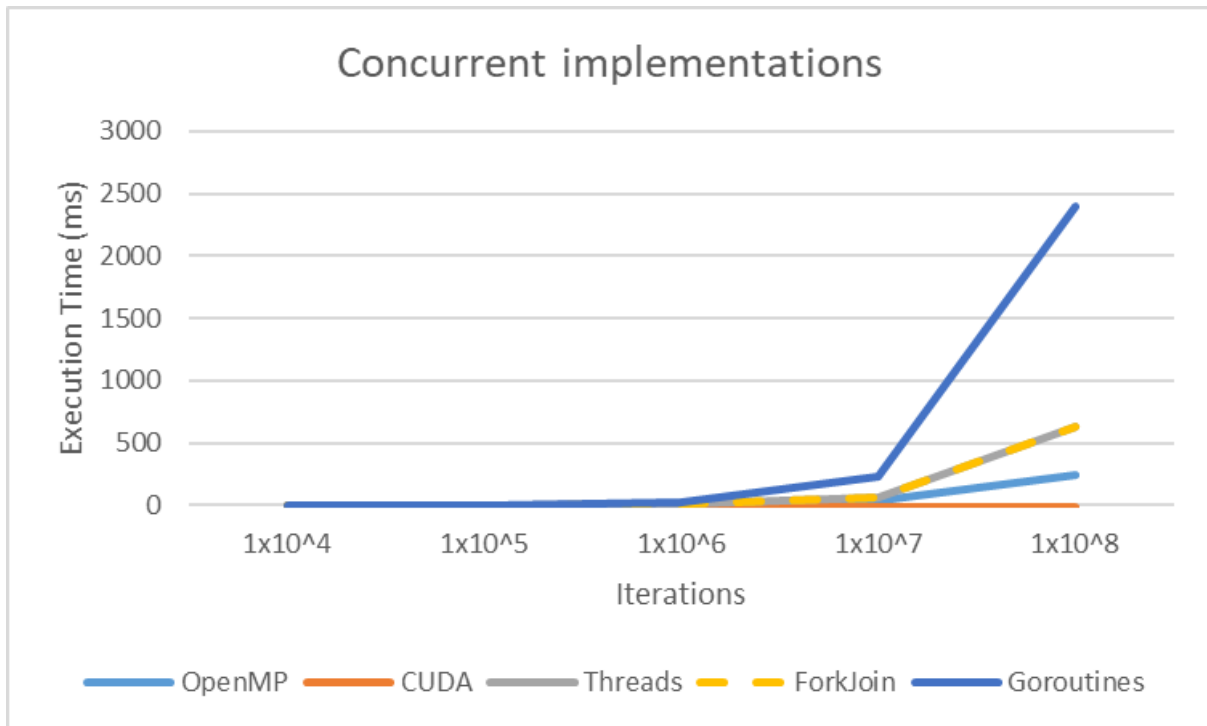
Comparación de implementaciones

De los 4 lenguajes de programación usados en las distintas implementaciones, se cuenta con mayor experiencia en el uso de C/C++. Mientras con el que se tiene menor dominio es en Golang. Los resultados obtenidos tras la ejecución de las soluciones seriales fueron los siguientes:



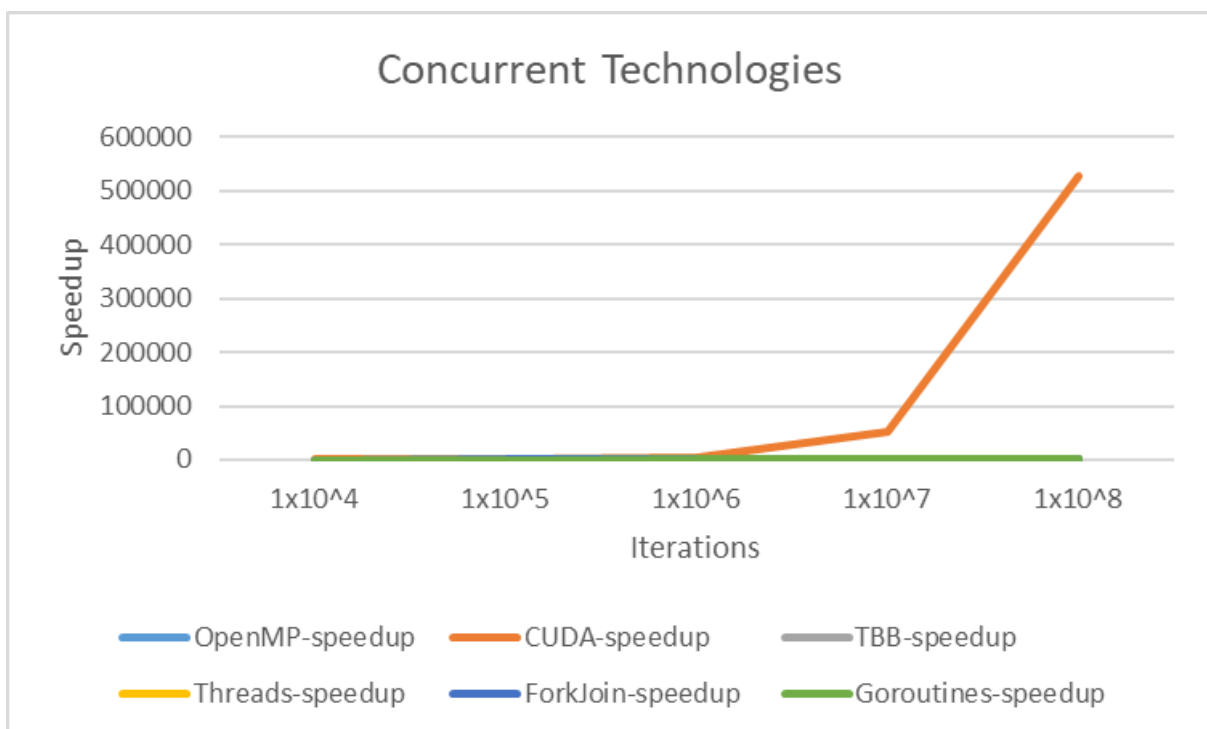
Gráfica 5. Comparación del tiempo de ejecución de todas las implementaciones seriales

Entre las tecnologías concurrentes con la que se tiene mayor experiencia es con el interfaz ForkJoin, mientras que con la que menos se tiene es en el uso de Goroutines. Los resultados obtenidos tras la ejecución de dichas soluciones fueron los siguientes:



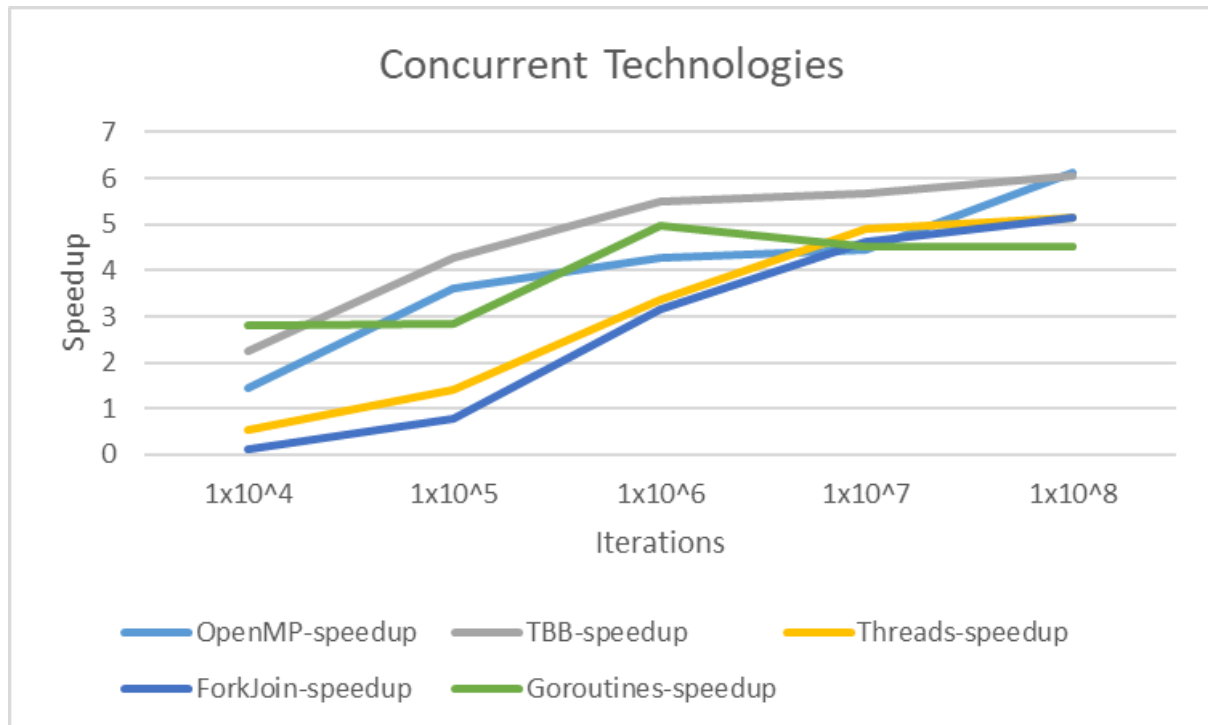
Gráfica 6. Comparación del tiempo de ejecución de todas las implementaciones concurrentes

Al momento de comparar el *speedup* de las diferentes tecnologías podemos observar que el claro ganador es CUDA debido al provecho que obtiene del uso de cientos de cores en la GPU:



Gráfica 7. Comparación del speedup de todas las implementaciones concurrentes

Enfocándonos en tecnologías que hacen uso puramente del CPU podemos notar que aquellas soportadas por las tecnologías C/C++ han logrado superar por 1 unidad a las demás, mientras que aquellas implementadas en Golang no han tenido un *speedup* tan bueno como sus competidores:



Gráfica 8. Comparación del speedup de las implementaciones concurrentes (sin CUDA)

Conclusiones

Una de las lecciones más importantes aprendidas durante la realización de esta investigación fue que el paradigma que usemos para resolver un problema tiene mucha relevancia, debido a que actualmente no sólo contamos con paradigmas de programación sino con paradigmas de uso de hardware. CUDA logró imponerse por un margen abismal sobre las demás tecnologías concurrentes tanto en tiempo de ejecución como en precisión de cálculos, todo esto debido a que hace uso de un recurso de hardware al cual no tienen acceso las demás: la Unidad de Procesamiento Gráfico (GPU).

Por otro lado, cabe destacar que el nivel de experiencia nos permitirá mejorar los resultados en tiempos de ejecución obtenidos en las distintas tecnologías, tanto a nivel serial como concurrente, al contar con mayor experiencia (6 años) en tecnologías relacionadas con C/C++ se pudo reflejar claramente como ciertos detalles han ayudado a ganarle a sus competidores y en cambio aquellos lenguajes en los que apenas se tiene experiencia de 1 año como lo es Golang tuvieron un rendimiento un tanto relativamente bajo pero con muchas áreas de oportunidad para eficientar futuras implementaciones.

Finalmente, este tipo de proyectos permiten que futuros profesionistas tengan un panorama más amplio al momento de resolver problemas que involucren tiempos de ejecución y gestión de los diferentes tipos de memoria del ordenador, ya que al momento de hacer despliegues a producción los usuarios exigirán tiempos más rápidos, mayor consumo de recursos multimedia y márgenes de error lo más cercanos a cero como lo fue en este caso el cálculo de una constante.

Agradecimientos

El principal agradecimiento es a mis padres por todo el apoyo que me han brindado a lo largo de mi carrera como estudiante, a mis profesores por todo el conocimiento que han compartido para con mi persona y contribuyeron a formarme como profesional en el que me he convertido y por último y no menos importante, estoy muy agradecido con mis hermanos y amigos cercanos por todo el apoyo incondicional que me han brindado.

Referencias

- Bai, Y. (n.d.). Best practices: Why use golang for your project. app development company. Retrieved November 30, 2022, from <https://www.uptech.team/blog/why-use-golang-for-your-project>
- Iwao, E. H. (2022, June 8). Calculating 100 trillion digits of pi on google cloud | google cloud blog. Even more pi in the sky: Calculating 100 trillion digits of pi on Google Cloud. Retrieved November 29, 2022, from <https://cloud.google.com/blog/products/compute/calculating-100-trillion-digits-of-pi-on-google-cloud>
- Lucas, J. (2020, September 10). Qué es C: Características Y Sintaxis. OpenWebinars.net. Retrieved November 30, 2022, from <https://openwebinars.net/blog/que-es-c/>
- Stewart, I. (2016). Medida de la circunferencia. Números Increíbles (1a ed., pp. 207–221). Editorial Crítica.
- Wikimedia Foundation. (2022, February 7). Serie de Leibniz. Wikipedia. Retrieved November 29, 2022, from https://es.wikipedia.org/wiki/Serie_de_Leibniz

Apéndices

Esta sección incluye el código fuente de todas las implementaciones.

Implementación serial en C

```
//
=====
===
//
// File: eulerpi_serial_c.c
// Author(s):
//          Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile using math.h
library:
//          gcc eulerpi_serial_c.c -lm
//
// SIZE = 100_000_000
// Serial time: 1,532.68780 ms
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

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

#define SIZE 100000000 //1e8

double calculate_pi(int size){
    double pi = 0;

    for(int i=0; i<size; i++){
        pi += pow(-1, i)*4 / ((i*2)+1);
    }
}
```

```

    }

    return pi;
}

int main(){
    double ms, result = 0;

    printf("Starting...\n");
    ms = 0;
    for(int i=0; i<N; i++){
        start_timer();
        result = calculate_pi(SIZE);
        ms+=stop_timer();
    }

    printf("Calculated value of PI = %lf\n", result);
    printf("Real PI = %f\n", M_PI);
    printf("Percent error = %f\n", abs(result-M_PI)/M_PI *
100);
    printf("avg time = %.5lf ms\n", (ms / N));
    return 0;
}

```


Implementación concurrente en C usando OpenMP

```
//
=====
===
//
// File: eulerpi_parallel_omp.c
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile using math.h
//              and the omp.h libraries:
//              gcc eulerpi_parallel_omp.c -lm -fopenmp
//
// SIZE = 100_000_000
// Serial time: 1,532.68780 ms
// Parallel time: 250.30 ms
// Speedup: 6.123403
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

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

#define SIZE 100000000 //1e8

double calculate_pi(int size){
    double pi = 0;

    #pragma omp parallel for shared(size) reduction(+:pi)
    for(int i=0; i<size; i++){
        pi += pow(-1, i)*4 / ((i*2)+1);
    }
}
```

```

        return pi;
    }

int main(){
    double ms, result = 0;

    printf("Starting...\n");
    ms = 0;
    for(int i=0; i<N; i++){
        start_timer();
        result = calculate_pi(SIZE);
        ms+=stop_timer();
    }

    printf("Calculated value of PI = %lf\n", result);
    printf("Real PI = %f\n", M_PI);
    printf("Percent error = %f\n", abs(result-M_PI)/M_PI *
100);
    printf("avg time = %.5lf ms\n", (ms / N));
    return 0;
}

```

Implementación concurrente en C usando CUDA

```
//
=====
===
//
// File: eulerpi_parallel_cuda.cu
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile using math.h
library:
//              nvcc eulerpi_parallel_cuda.cu -lm
//
// SIZE = 100_000_000
// Serial time: 1,532.68780 ms
// Parallel time: 0.00290 ms
// Speedup: 528,513.0344
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

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

#define SIZE 100000000 //1e8
#define THREADS      256
#define BLOCKS MMIN(32, ((SIZE / THREADS) + 1))

__global__ void calculate_pi(double *result) {
    __shared__ double cache[THREADS];

    int tid = threadIdx.x + (blockIdx.x * blockDim.x);
    int cacheIndex = threadIdx.x;
```

```

double acum = 0;
while (tid < SIZE) {
    acum += pow(-1, tid)*4 / ((tid*2)+1);
    tid += blockDim.x * gridDim.x;
}

cache[cacheIndex] = acum;

__syncthreads();

int i = blockDim.x / 2;
while (i > 0) {
    if (cacheIndex < i) {
        cache[cacheIndex] += cache[cacheIndex + i];
    }
    __syncthreads();
    i /= 2;
}

if (cacheIndex == 0) {
    result[blockIdx.x] = cache[cacheIndex];
}
}

int main(int argc, char* argv[]){
    double *results, *d_r;
    double ms;

    results = (double*) malloc( BLOCKS * sizeof(double));
    cudaMalloc( (void**) &d_r, BLOCKS * sizeof(double) );

    printf("Starting...\n");
    ms = 0;
    for(int i=0; i<N; i++){
        start_timer();
        calculate_pi<<< BLOCKS, THREADS >>>(d_r);
        ms += stop_timer();
    }

    cudaMemcpy(results, d_r, BLOCKS * sizeof(long),
cudaMemcpyDeviceToHost);
    double acum = 0;
    for (int i = 0; i < BLOCKS; i++) {

```

```

        acum += results[i];
    }

    printf("Calculated PI = %lf\n", acum);
    printf("Real PI = %f\n", M_PI);
    printf("Percent error = %f\n", abs(acum-M_PI)/M_PI * 100);
    printf("avg time = %.5lf ms\n", (ms / N));

    cudaFree(d_r);

    free(results);
    return 0;
}

```

Implementación serial en C++

```
//
=====
===
//
// File: eulerpi_serial_cpp.cpp
// Author(s):
//          Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile using math.h
library:
//              g++ eulerpi_serial_cpp.cpp -lm
//
// SIZE = 100_000_000
// Serial time: 2373.42 ms
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

#include<iostream>
#include <math.h>
#include "utils.h"

#define SIZE 100000000 //1e8

using namespace std;

class CalculatePI{
    private:
        int size;
        double result;

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

        double getResult() const{
```

```

        return result;
    }

    void calculate(){
        result = 0;

        for(int i=0; i<size; i++){
            result += pow(-1, i)*4 / ((i*2)+1);
        }
    }
};

int main(){
    double ms, result = 0;

    cout << "Starting..." << endl;
    ms = 0;
    CalculatePI obj(SIZE);
    for(int i=0; i<N; i++){
        start_timer();
        obj.calculate();
        ms+=stop_timer();
    }

    cout << "Calculated value of PI = " << obj.getResult() <<
endl;
    cout << "Real PI = " << M_PI << endl;
    cout << "Percent error = " <<
abs(obj.getResult()-M_PI)/M_PI << endl;
    cout << "avg time = " << (ms / N) << " ms" << endl;
    return 0;
}

```

Implementación concurrente en C++ usando TBB

```
//
=====
===
//
// File: eulerpi_parallel_tbb.cpp
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile using math.h
library:
//              g++ eulerpi_parallel_tbb.cpp -lm -ltbb
//
// SIZE = 100_000_000
// Serial time: 2373.42 ms
// Parallel time: 392.131 ms
// Speedup: 6.0526
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

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

#define SIZE 10000 //1e8

using namespace std;
using namespace tbb;

class CalculatePI{
    private:
        int size;
```



```

        double result;

public:
    CalculatePI(int s) : size(s){}
    CalculatePI(CalculatePI &x, split): result(0) {}

    double getResult() const{
        return result;
    }

    void operator()(const blocked_range<int> &r){
        for(int i=r.begin(); i!=r.end(); i++){
            result += pow(-1, i)*4 / ((i*2)+1);
        }
    }

    void join(const CalculatePI &x) {
        result += x.result;
    }
};

int main(){
    double ms, result = 0;

    cout << "Starting..." << endl;
    ms = 0;
    CalculatePI obj(SIZE);
    for(int i=0; i<N; i++){
        start_timer();
        parallel_reduce(blocked_range<int>(0, SIZE), obj);
        result = obj.getResult();
        ms+=stop_timer();
    }

    cout << "Calculated value of PI = " << result/N << endl;
    cout << "Real PI = " << M_PI << endl;
    cout << "Percent error = " << abs((result/N)-M_PI)/M_PI <<
endl;
    cout << "avg time = " << (ms / N) << " ms" << endl;
    return 0;
}

```

Implementación serial en Java

```
//
=====
===
//
// File: Eulerpi_serial_java.java
// Author(s):
//          Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile:
//              javac Eulerpi_serial_java.java
//
// SIZE = 100_000_000
// Serial time: 3242.9 ms
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

public class Eulerpi_serial_java{
    private static final int SIZE = 100_000_000;
    private double result;

    public Eulerpi_serial_java(){
        result = 0;
    }

    public double getResult(){
        return result;
    }

    public void calculatePi(){
        result = 0;

        for(int i=0; i<SIZE; i++){
            result += Math.pow(-1, i)*4 / ((i*2)+1);
        }
    }
}
```

```

    }

    public static void main (String args[]){
        long startTime, stopTime;
        double acum = 0;

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

            e.calculatePi();

            stopTime = System.currentTimeMillis();

            acum += (stopTime - startTime);
        }
        System.out.println("Calculated value of PI = " +
e.getResult());
        System.out.println("Real PI = " + Math.PI);
        System.out.println("Percentage error = " +
Math.abs(e.getResult()-Math.PI)/Math.PI*100);
        System.out.println("avg time = " + (acum / Utils.N)
+ " ms");
    }
}

```

Implementación concurrente en Java usando Threads

```
//
=====
===
//
// File: Eulerpi_parallel_threads.java
// Author(s):
//           Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//               approximation of pi using the euler series
//               algorithm. How to compile:
//               javac Eulerpi_parallel_threads.java
//
// SIZE = 100_000_000
// Serial time: 3242.9 ms
// Parallel time: 629.5 ms
// Speedup: 5.1515
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

public class Eulerpi_parallel_threads extends Thread{
    private static final int SIZE = 100_000_000;
    private int start, end;
    private double result;

    public Eulerpi_parallel_threads(int start, int end) {
        this.start = start;
        this.end = end;
        this.result = 0;
    }

    public double getResult() {
        return result;
    }

    public void run() {
```

```

        result = 0;
        for (int i = start; i < end; i++) {
            result += Math.pow(-1, i)*4 / ((i*2)+1);
        }
    }

    public static void main(String args[]){
        long startTime, stopTime;
        int block;
        Eulerpi_parallel_threads threads[];
        double ms;
        double result = 0;

        block = SIZE / Utils.MAXTHREADS;
        threads = new
Eulerpi_parallel_threads[Utils.MAXTHREADS];

        System.out.println("Starting with "+ Utils.MAXTHREADS
+" threads...");
        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
Eulerpi_parallel_threads( (i * block), ((i + 1) * block) );
                } else {
                    threads[i] = new
Eulerpi_parallel_threads( (i * block), SIZE);
                }
            }

            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);

        if (j == Utils.N) {
            result = 0;
            for (int i = 0; i < threads.length; i++) {
                result += threads[i].getResult();
            }
        }
        System.out.println("Calculated value of PI = " +
result);
        System.out.println("Real PI = " + Math.PI);
        System.out.println("Percentage error = " +
Math.abs(result-Math.PI)/Math.PI*100);
        System.out.println("avg time = " + (ms / Utils.N) +
" ms");
    }
}

```

Implementación concurrente en Java usando ForkJoin

```
//
=====
===
//
// File: Eulerpi_parallel_forkjoin.java
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile:
//              javac Eulerpi_parallel_forkjoin.java
//
// SIZE = 100_000_000
// Serial time: 3242.9 ms
// Parallel time: 631.0 ms
// Speedup: 5.1393
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class Eulerpi_parallel_forkjoin extends
RecursiveTask<Double>{
    private static final int SIZE = 100_000_000;
    private static final int MIN = 100_000;
    private int start, end;

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

    protected Double computeDirectly() {
        double result = 0;
        for (int i = start; i < end; i++) {
```

```

        result += Math.pow(-1, i)*4 / ((i*2)+1);
    }
    return result;
}

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

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

    System.out.println("Starting with "+ Utils.MAXTHREADS
+" threads");
    ms = 0;
    for (int i = 0; i < Utils.N; i++) {
        startTime = System.currentTimeMillis();

        pool = new ForkJoinPool(Utils.MAXTHREADS);
        result = pool.invoke(new
Eulerpi_parallel_forkjoin(0, SIZE));

        stopTime = System.currentTimeMillis();
        ms += (stopTime - startTime);
    }
    System.out.println("Calculated value of PI = " +
result);
    System.out.println("Real PI = " + Math.PI);
    System.out.println("Percentage error = " +
Math.abs(result-Math.PI)/Math.PI*100);
}

```



```
        System.out.println("avg time = " + (ms / Utils.N) +  
" ms");  
    }  
}
```

Implementación serial en Golang

```
//
=====
===
//
// File: eulerpi_serial_go.go
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile:
//              go run eulerpi_serial_go.go
//
// SIZE = 100_000_000
// Serial time: 10897.6999 ms
//
// Copyright (c) 2020 by Tecnologico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

package main

import (
    "fmt"
    "math"
    "time"
)

const SIZE = 100_000_000
const N = 10

func calculatePi(size int) float64{
    pi:=0.0

    for i:=0; i<size; i++){
        pi += math.Pow(-1, float64(i))*4/((float64 (i)*2)+1)
    }
}
```

```

        return pi
    }

func main(){
    var start time.Time
    var elapsed time.Duration
    var result float64

    fmt.Println("Starting...")
    for i:=0; i<N; i++){
        start = time.Now()
        result = calculatePi(SIZE)
        elapsed = time.Since(start)
    }

    fmt.Printf("Calculated value of PI = %f\n", result)
    fmt.Printf("Real PI = %f\n", math.Pi)
    fmt.Printf("Percent error = %f\n",
math.Abs(result-math.Pi)/math.Pi*100)
    fmt.Printf("avg time = %s\n", elapsed)
}

```

Implementación concurrente en Golang usando Goroutines

```
//
=====
===
//
// File: eulerpi_parallel_goroutines.go
// Author(s):
//         Roberto Carlos Guzmán Cortés A01702388
//
// Description: This file contains the code to calculate an
//              approximation of pi using the euler series
//              algorithm. How to compile:
//              go run eulerpi_serial_go.go
//
// SIZE = 100_000_000
// Serial time: 10897.6999 ms
// Parallel time: 2404.7984 ms
// Speedup: 4.5316
//
// Copyright (c) 2020 by Tecnológico de Monterrey.
// All Rights Reserved. May be reproduced for any
non-commercial
// purpose.
//
//
=====
===

package main

import (
    "fmt"
    "math"
    "runtime"
    "time"
)

const SIZE = 100_000_000
const N = 10

// calculatePi launches n goroutines to compute an
// approximation of pi.
func calculatePi(size int) float64 {
    n_threads := runtime.NumCPU()
```

```

    block := size/n_threads

    ch := make(chan float64)
    for i := 0; i < n_threads; i++ {
        if i != n_threads-1{
            go term(ch, i*block, (i+1)*block)
        } else{
            go term(ch, i*block, size)
        }
    }
    f := 0.0
    for k := 0; k < n_threads; k++ {
        f += <-ch
    }
    return f
}

func term(ch chan float64, start int, end int) {
    acum := 0.0
    for i:=start; i<end; i++){
        acum += math.Pow(-1, float64(i))*4 /
((2*float64(i)) + 1)
    }
    ch <- acum
}

func main(){
    var start time.Time
    var elapsed time.Duration
    var result float64

    fmt.Printf("Starting with %v threads...\n",
runtime.NumCPU())
    for i:=0; i<N; i++){
        start = time.Now()
        result = calculatePi(SIZE)
        elapsed = time.Since(start)
    }

    fmt.Printf("Calculated value of PI = %f\n", result)
    fmt.Printf("Real PI = %f\n", math.Pi)
    fmt.Printf("Percent error = %f\n",
math.Abs(result-math.Pi)/math.Pi*100)
    fmt.Printf("avg time = %s\n", elapsed)
}

```

}