Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de ciencias de la computación
CC3069 Computación Paralela y Distribuida

Lab 01 - Pi

Evelyn Andrea Amaya Malin 19357
Brandon Josué Hernández Marroquín 19376
Jose Javier Hurarte Hernández 19707

Guatemala, Ciudad de Guatemala 21 de febrero de 2023

Ejercicio 1

a. Implementación del algoritmo y mediciones

Programa secuencial

```
for (int k = 0; k < n; k++) {
    sum  += factor/(2*k+1);
    factor = -factor;
}
```

Programa paralelo

```
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (int k = 0; k < n; k++) {
    sum  += factor/(2*k+1);
    factor = -factor;
}
```

piSeriesNaive

```
PROBLEMS   OUTPUT   GITLENS   ROBOT DOCUMENTATION   ROBOT OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> gcc -o ejer1 .\piSeriesNaive.c -fopenmp
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 1000 2


Con n threads = 2 y con n = 1000

Con tiempo = 0.001000
Con precision = 0.001000
Con pi = 3.140593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA\ejer 1\inciso a> ./ejer1 2000 5


Con n threads = 5 y con n = 2000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = -3.074157
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 3000 7


Con n threads = 7 y con n = 3000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.066266
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 10000 10


Con n threads = 10 y con n = 10000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.175348
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 15000 10


Con n threads = 10 y con n = 15000

Con tiempo = 0.002000
Con precision = 0.001000
Con pi = 3.098403
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> gcc -o ejer1 .\piSeriesNaive.c -fopenmp
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 50000 1


Con n threads = 1 y con n = 50000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141573
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 50000 10


Con n threads = 10 y con n = 50000

Con tiempo = 0.002000
Con precision = 0.001000
Con pi = -3.482582
```

Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846).

Debido a la dependencia con la variable *factor*, a veces el resultado obtenido es negativo (ejecución 2 y 6). Además, el resultado es más preciso al ser secuencial dando como resultado 3.141573 o con una pequeña cantidad de threads (n=2) con un resultado 3.1405. De lo contrario el resultado es impreciso.

piSeriesSeq

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> gcc -o ejer1 .\piSeriesSeq.c -fopenmp
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 1000 2


Con n threads = 1 y con n = 1000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.140593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 2000


Con n threads = 1 y con n = 2000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141093
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 3000


Con n threads = 1 y con n = 3000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141259
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 10000


Con n threads = 1 y con n = 10000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141493
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 15000


Con n threads = 1 y con n = 15000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141526
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 50000


Con n threads = 1 y con n = 50000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141573
```

b. Identifique el tipo de dependencia que se da con la variable factor.

Es una antidependencia debido a que se escribe en factor y luego se lee ese valor en la siguiente iteración para que lo use la variable sum.

c. Observe el algoritmo y la serie numérica. Describa en sus propias palabras la razón por la cual factor = – factor.

Esto se debe a que la serie para calcular pi va intercalando los símbolos. Comenzando por positivo, negativo, positivo, etc..

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

d. Eliminar la dependencia

Programa paralelo

```c
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (int k = 0; k < n; k++) {
    factor = (k % 2 == 0) ? 1.0 : -1.0;
    sum  += factor/(2*k+1);
}
```

piSeriesNaive

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> gcc -o ejer1 .\piSeriesNaive.c -fopenmp
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 1000 2


Con n threads = 2 y con n = 1000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.140593
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA>
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 2000 5


Con n threads = 5 y con n = 2000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141093
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 3000 7


Con n threads = 7 y con n = 3000

Con tiempo = 0.001000
Con precision = 0.001000
Con pi = 2.991310
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 10000 10


Con n threads = 10 y con n = 10000

Con tiempo = 0.001000
Con precision = 0.001000
Con pi = 3.124938
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 15000 10


Con n threads = 10 y con n = 15000

Con tiempo = 0.001000
Con precision = 0.001000
Con pi = 3.153517
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 50000 10


Con n threads = 10 y con n = 50000

Con tiempo = 0.001000
Con precision = 0.001000
Con pi = 3.020061
```

Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846).
A diferencia del primer inciso, ahora todos los resultados son positivos teniendo un menor margen de error. El máximo error obtenido fue de 4.78% en la tercera ejecución dando un resultado pi de 2.9913. Y, la mejor ejecución fue con 5 threads y n=2000 dando un resultado pi de 2.141093 y un margen de error de 0.019%.

e. piSeriesNaive con 1 solo thread

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 1000 1


Con n threads = 1 y con n = 1000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.140593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 2000 1


Con n threads = 1 y con n = 2000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141093
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 3000 7


Con n threads = 7 y con n = 3000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141259
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 10000 1


Con n threads = 1 y con n = 10000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141493
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 15000 1


Con n threads = 1 y con n = 15000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141526
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 50000 1


Con n threads = 1 y con n = 50000

Con tiempo = 0.000000
Con precision = 0.001000
Con pi = 3.141573
```

¿Por qué el resultado es diferente con 1 solo thread?
Con 1 solo thread, al ser secuencial todos los ciclos obtienen el signo correspondiente para el actual resultado y por lo tanto el valor es más preciso a pi. Mientras que con más threads, el signo puede ser incorrecto debido a que otro thread haya modificado la variable factor.

f. Agregar la cláusula private

```c
#pragma omp parallel for num_threads(thread_count) reduction(+:sum) private(factor)
for (int k = 0; k < n; k++) {
    factor = (k % 2 == 0) ? 1.0 : -1.0;
    sum  += factor/(2*k+1);
}
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 100000000 2


Con n threads = 2 y con n = 100000000

Con tiempo = 0.135000
Con precision = 0.001000
Con pi = 3.141593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 100000000 5


Con n threads = 5 y con n = 100000000

Con tiempo = 0.056000
Con precision = 0.001000
Con pi = 3.141593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> gcc -o ejer1 .\piSeriesNaive.c -fopenmp
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 100000000 10


Con n threads = 10 y con n = 100000000

Con tiempo = 0.030000
Con precision = 0.001000
Con pi = 3.141593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 500000000 10


Con n threads = 10 y con n = 500000000

Con tiempo = 0.140000
Con precision = 0.001000
Con pi = 3.141593
```

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 500000000 15


Con n threads = 15 y con n = 500000000

Con tiempo = 0.100000
Con precision = 0.001000
Con pi = 3.141593
```

Un solo thread

```
PS C:\Users\andre\Documents\Trabajo\LAB01-PARALELA> ./ejer1 500000000 1


Con n threads = 1 y con n = 500000000

Con tiempo = 1.324000
Con precision = 0.001000
Con pi = 3.141593
```

g. Cálculos finales

Adjunto el excel en el github con todas las capturas y resultados

cores	8			
N	1	8	16	8
	secuencial n=1.00E08	paralelo threads=cores n=1.00E08	paralelo threads=2*cores n=1.00E08	paralelo n=1.00E9 y threads=cores
t'par	0.454	0.484	0.629	-
tiempo	0.264	0.036	0.037	0.341
speedup	1	7.333333333	7.135135135	0.774193548
eficiencia	1	0.916666667	0.445945946	0.096774194
escalabilidad fuerte	3.787878788	0.916666667	0.445945946	0.096774194
escalabilidad debil	0.581497797	0.074380165	0.058823529	-
tiempo	0.264	0.036	0.024	0.345
speedup	1	7.333333333	11	0.765217391
eficiencia	1	0.916666667	0.6875	0.095652174
escalabilidad fuerte	3.787878788	0.916666667	0.6875	0.095652174
escalabilidad debil	0.581497797	0.074380165	0.038155803	-
tiempo	0.266	0.037	0.021	0.346
speedup	1	7.135135135	12.57142857	0.76300578
eficiencia	1	0.891891892	0.785714286	0.095375723
escalabilidad fuerte	3.759398496	0.891891892	0.785714286	0.095375723
escalabilidad debil	0.585903084	0.076446281	0.033386328	-
tiempo	0.264	0.036	0.022	0.345
speedup	1	7.333333333	12	0.765217391
eficiencia	1	0.916666667	0.75	0.095652174
escalabilidad fuerte	3.787878788	0.916666667	0.75	0.095652174
escalabilidad debil	0.581497797	0.074380165	0.034976153	-
tiempo	0.264	0.035	0.02	0.345
speedup	1	7.542857143	13.2	0.765217391
eficiencia	1	0.942857143	0.825	0.095652174
escalabilidad fuerte	3.787878788	0.942857143	0.825	0.095652174
escalabilidad debil	0.581497797	0.07231405	0.031796502	-

h. Políticas de planificación

Adjunto el excel en el github con todas las capturas y resultados

	static			dynamic			guided			auto
block_size	16	64	128	16	64	128	16	64	128	
tseq	0.454	0.452	0.451	0.572	0.495	0.473	0.457	0.453	0.463	0.451
tiempo	0.066	0.063	0.072	0.194	0.091	0.081	0.067	0.066	0.064	0.083
speedup	6.878787879	7.174603175	6.263888889	2.94845361	5.43956044	5.83950617	6.82089552	6.86363636	7.234375	5.43373494
tiempo	0.064	0.064	0.071	0.196	0.089	0.077	0.076	0.062	0.06	0.075
speedup	7.09375	7.0625	6.352112676	2.91836735	5.56179775	6.14285714	6.01315789	7.30645161	7.71666667	6.01333333
tiempo	0.066	0.068	0.066	0.194	0.089	0.08	0.06	0.064	0.065	0.077
speedup	6.878787879	6.647058824	6.833333333	2.94845361	5.56179775	5.9125	7.61666667	7.078125	7.12307692	5.85714286
tiempo	0.065	0.061	0.068	0.186	0.089	0.077	0.079	0.072	0.065	0.065
speedup	6.984615385	7.409836066	6.632352941	3.07526882	5.56179775	6.14285714	5.78481013	6.29166667	7.12307692	6.93846154
tiempo	0.068	0.062	0.071	0.195	0.088	0.083	0.066	0.064	0.067	0.086
speedup	6.676470588	7.290322581	6.352112676	2.93333333	5.625	5.69879518	6.92424242	7.078125	6.91044776	5.24418605
promedio tiempo	0.0658	0.0636	0.0696	0.193	0.0892	0.0796	0.0696	0.0656	0.0642	0.0772
promedio speedup	6.902482346	7.116864129	6.486760103	2.96477534	5.54999074	5.94730313	6.63195453	6.92360093	7.22152865	5.89737174

dynamic		
16	64	128
0.572	0.495	0.473
0.194	0.091	0.081
2.94845361	5.43956044	5.83950617
0.196	0.089	0.077
2.91836735	5.56179775	6.14285714
0.194	0.089	0.08
2.94845361	5.56179775	5.9125
0.186	0.089	0.077
3.07526882	5.56179775	6.14285714
0.195	0.088	0.083
2.93333333	5.625	5.69879518
0.193	0.0892	0.0796
2.96477534	5.54999074	5.94730313

¿Con cuál política de planificación obtuvo mejores resultados? Los mejores resultados se obtuvieron con un block_size de 16 haciendo uso de la política dynamic. Obteniendo un speedup promedio de 2.964 y un tiempo de ejecución de 0.193 segundos haciendo uso de n=1.00E8 y threads=8

Ejercicio 2

i. **Implementación**

$$\pi = 4\left[\sum_{i\in Even}^{\infty} \frac{1}{2i+1} - \sum_{j\in Odd}^{\infty} \frac{1}{2j+1}\right]$$

```
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (int k = 1; k < n; k +=2) {
    sum  += (1.0/((2.0*(k-1.0)) + 1.0)) - (1.0/((2.0*k) + 1.0));
}
```

Implemente el programa descrito por la ecuación anterior (piSeriesAlt.c) , compílelo y ejecútelo Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846). Haga una comparación con los mismos parámetros (threads, n) de esta versión y su mejor versión del inciso h.

	sequential	Alt_sequential
	2.662	1.382
	2.67	1.383
	2.677	1.388
	2.69	1.403
	2.664	1.378
Promedio	**2.6726**	**1.3868**

Para poder realizar dichas comparaciones se volvió a correr el programa secuencial y se tomaron nuevas medidas debido a que se cambió el ambiente en el que se corrieron los programas del inciso anterior, por lo que se tuvo que hacer esto para tener medidas más verídicas. Además todas las mediciones de este inciso fueron hechas con 10 threads y n=1E9, aumentando este factor para tener medidas más significativas. Además se incluyó una hoja de cálculo con todos los resultados resumidos y las capturas de pantalla de estos en la carpeta del ejercicio 2 en el repositorio

	paralell_dynamic	Alt_parallel	Alt_dynamic
	1.63	0.357	0.795
	1.596	0.355	0.812
	1.597	0.354	0.8
	1.625	0.347	0.8
	1.649	0.357	0.797
Promedio	**1.6194**	**0.354**	**0.8008**
Speedup	**1.650364332**	**3.91751412**	**1.73176823**
Eficiencia	**0.165036433**	**0.39175141**	**0.17317682**

Con estos resultados podemos ver primero que la solución alternativa secuencial es 1.93 veces más rápida que la solución original secuencial, esto se debe principalmente a que a pesar de que ambas soluciones sean O(n),o de complejidad lineal, la solución original utiliza los n números para la sumatoria, mientras que la sumatoria alternativa únicamente se basa en la utilización de un máximo de n/2 números, teniendo así que hacer muchos menos cálculos.
Con respecto a las soluciones paralelas, vemos que la solución más rápida del inciso anterior(parallel_dynamic) tiene en este ambiente un speedup del 1.65 y eficiencia del 0.165, mientras que la solución alternativa paralela (con schedule default) tiene un speedup de 3.92 y eficiencia de 0.39, teniendo la solución alternativa paralela mucha más mejora en tiempo que su versión secuencial, teniendo así más del doble de eficiencia que la versión preferible del inciso anterior.

Además también se comparó que sucede cuando se le coloca la calendarización dynamic a la solución alternativa, obtenemos un speedup del 1.73 y una eficiencia de 0.17, lo cual es ligeramente superior a la versión paralela dinámica. A pesar de esto, al comparar los datos podemos observar que esta variación es 2.26 veces menos eficiente y más lenta que la solución alternativa paralela con calendarización default (estática).

Con respecto al valor de PI medido en todas las ejecuciones de las 5 variaciones del cálculo de PI, podemos ver que siempre obtenemos el siguiente resultado PI= 3.141593 (se puede verificar en la hoja de capturas del excel incluído en el apartado del ejercicio 2 en el repositorio). Teniendo nuestros programas una precisión de 6 decimales y al no variar estos dígitos con ejecución podemos decir que las variables críticas están correctamente manejadas y no dan lugar a condiciones de carrera.

Con todo lo anteriormente mencionado podemos concluir que la solución preferible es la solución alternativa paralela con calendarización default (estática) ya que es superior en tiempo, speedup y eficiencia al resto de soluciones.

j. Implementación

Pruebe compilar su mejor versión al momento pero esta vez agregando la opción de optimización "-O2". Mida varias veces el tiempo de ejecución y compare con la versión sin la bandera de optimización. ¿Qué pudieron observar? Comenten entre el grupo e incluya un resumen de su discusión.

	Alt_parallel	O2 Alt_parallel
	0.357	0.313
	0.355	0.309
	0.354	0.312
	0.347	0.31
	0.357	0.312
Promedio	**0.354**	**0.3112**
Varianza	**1.36E-05**	**2.16E-06**
Speedup	**3.917514124**	**4.456298201**
Eficiencia	**0.391751412**	**0.44562982**

Con las optimizaciones de O2 podemos ver principalmente que el tiempo promedio de ejecución promedio se reduce en un 12.01% lo cual nos es un indicador de que si hay una reducción de tiempo con estas optimizaciones. Además podemos ver a simple vista que los datos con las optimizaciones son menos variados, y al sacar la varianza de ambos lo podemos comprobar que la varianza con las optimizaciones de O2 se reduce 6.3 veces lo cual nos dice que los tiempos de ejecución de este programa con las optimizaciones es mucho más estable. Finalmente al observar los otros cálculos podemos ver que es 1.14 veces más eficiente y posee un 1.14 más speedup con las optimizaciones de O2. Por lo que podemos decir que esta versión es preferible.