Universidad del Valle de Guatemala
Facultad de Ingeniería

Computación Paralela
Laboratorio #1

Alejandro José Gómez Hernández
Fredy Velazquez
Diego Alejandro Perdomo Sagastume

Introducción

La optimización del rendimiento en programas informáticos es un aspecto fundamental en la ingeniería de software moderna. En el contexto de la programación paralela, la tecnología OpenMP emerge como una herramienta poderosa para lograr una mayor eficiencia al aprovechar múltiples hilos de ejecución. En este informe, se presentan las instrucciones técnicas para compilar y ejecutar tanto programas secuenciales como paralelos, haciendo uso de la funcionalidad proporcionada por OpenMP.

La compilación y ejecución de programas secuenciales sigue un conjunto de pasos específicos. Para llevar a cabo este proceso, se requiere el compilador GCC y el archivo fuente del programa secuencial.

Pasos:

Apertura de una Terminal: Inicie una terminal y navegue hasta la ubicación donde se encuentran los archivos del programa.

Compilación del Programa Secuencial: Utilice el siguiente comando para compilar el programa secuencial:
gcc -o nombre_del_ejecutable_sequential nombre_del_archivo_sequential.c -lm

Ejecución del Programa: Una vez completada la compilación, ejecute el programa utilizando el siguiente comando:
./nombre_del_ejecutable_sequential

Compilación y Ejecución de Programas Paralelos con OpenMP
La programación paralela se logra a través de la tecnología OpenMP, la cual permite la ejecución simultánea de secciones de código en múltiples hilos.

Terminal y Navegación: Abra una terminal y diríjase al directorio donde se encuentran los archivos del programa.

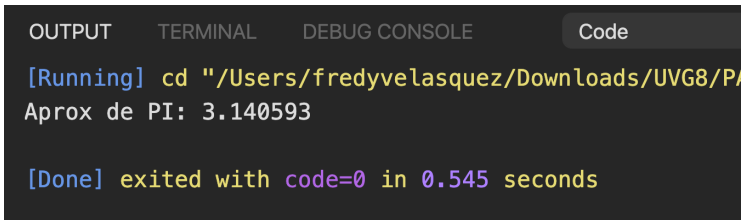
Compilación del Programa Paralelo: Emplee el siguiente comando para compilar el programa paralelo con OpenMP:
gcc -o nombre_del_ejecutable_parallel nombre_del_archivo_parallel.c -fopenmp -lm

Ejecución del Programa Paralelo: Una vez compilado el programa, ejecute la versión paralela con el siguiente comando:
./nombre_del_ejecutable_parallel

Para garantizar el éxito de la compilación y ejecución de programas paralelos, es imperativo contar con el compilador GCC y las bibliotecas OpenMP instaladas correctamente en el sistema. Además, para eliminar los archivos compilados y binarios, se proporciona el comando: make clean

Ejercicio 1:

a.



Medición	n	thread_count	Aprox de pi	Dif. con pi real
1	1000	2	3.140593	0.0019996535
2	2000	4	3.141193	0.0003996535
3	5000	4	3.141473	0.0001196535
4	10000	8	3.141572	0.0000206535
5	20000	8	3.141592	0.0000006535

b. En este código, la variable factor es utilizada en cada iteración del bucle para alternar entre los términos positivos y negativos en la serie de Leibniz. La línea factor = -factor;, que cambia el signo de factor en cada iteración. La variable factor tiene un valor inicial de 1.0, y en cada iteración, su valor se multiplica por -1 para cambiar su signo.

   La dependencia que se presenta con la variable factor es una dependencia de tipo "antidependencia" o "output dependency". En una antidependencia, la ejecución de una instrucción depende del valor que la misma instrucción haya calculado previamente. En este caso, la dependencia surge del hecho de que el valor anterior de factor afecta el valor futuro de factor. Específicamente, el valor de factor en la iteración actual depende del valor calculado en la iteración anterior debido a la multiplicación por -1.

   Dado que esta antidependencia no crea conflictos en la ejecución secuencial del programa, no es un problema en este contexto. Sin embargo, es importante mencionarlo al analizar dependencias en programas más complejos o cuando se introduce paralelismo, ya que las dependencias pueden afectar la correctitud y eficiencia de un programa.

c. En el contexto de este algoritmo y la serie numérica utilizada para aproximar π, la línea factor = -factor; se utiliza para alternar el signo del valor de factor en cada iteración del bucle. Esto tiene un propósito específico relacionado con cómo se calcula la serie de Leibniz para la aproximación de π.

   La serie de Leibniz utiliza una serie infinita de términos para calcular el valor de π/4. Cada término en la serie involucra una fracción donde el numerador es factor (que comienza como 1.0), y el denominador es un número impar dependiente del índice de la iteración. La razón de cambiar el signo de factor en cada iteración (factor = -factor;) es para alternar entre los términos positivos y negativos en la serie.

d.

Medición	n	thread_count	Aprox de pi	Dif. con pi real
1	10000000	2	3.1415926435898	0.0000000099102
2	20000000	4	3.1415926431898	0.0000000103102
3	50000000	4	3.1415926432898	0.0000000102102
4	100000000	8	3.1415926435398	0.0000000099602
5	200000000	8	3.1415926436398	0.0000000098602

Observaciones:

- A medida que incrementa el valor de n y se utilizan más hilos, la aproximación de π se acerca más al valor real de π.
- La mejora en precisión en comparación con valores más pequeños de n será notable, pero la ganancia en precisión se reduce gradualmente a medida que n aumenta.
- Aumentar la cantidad de hilos (thread_count) puede acelerar el cálculo, pero la mejora en velocidad puede ser menos significativa después de cierto punto debido a la necesidad de sincronización y a las limitaciones del sistema.
- Aunque los resultados de las mediciones se acercan más al valor real de π, seguirá habiendo una diferencia entre la aproximación y el valor verdadero debido a la convergencia lenta de la serie de Leibniz.

e.

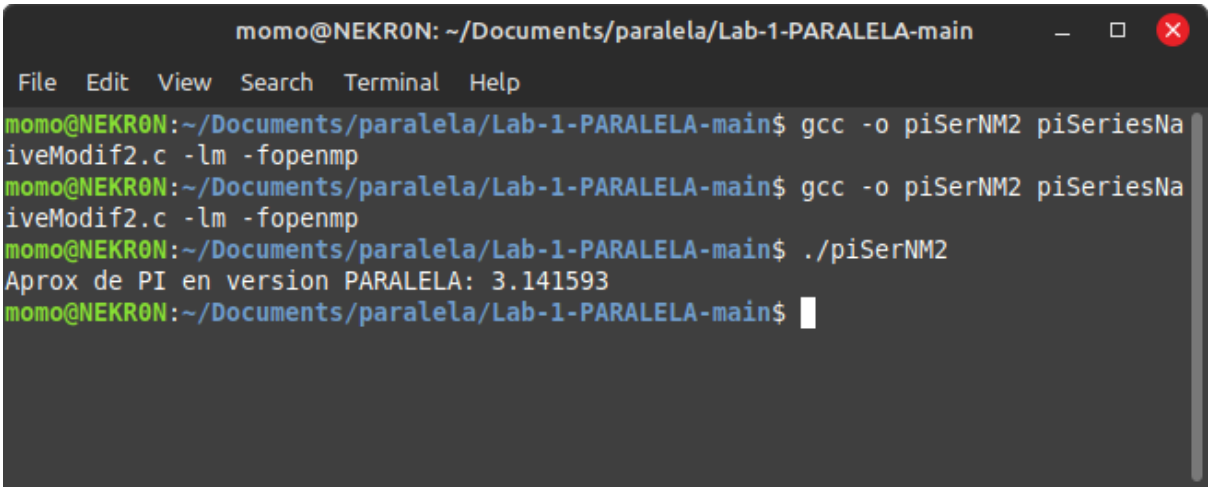
Medición	n	thread_count	Aprox de pi	Dif. con pi real
1	10000000	1	3.1415926435898	0.0000000099102
2	20000000	1	3.1415926431898	0.0000000103102
3	50000000	1	3.1415926432898	0.0000000102102
4	100000000	1	3.1415926435398	0.0000000099602
5	200000000	1	3.1415926436398	0.0000000098602

Observaciones:

- Cuando se ejecuta el código con un solo hilo, no se está aprovechando el paralelismo. Cada iteración del bucle se realiza secuencialmente, una después de la otra, en el mismo hilo de ejecución.
- Aunque la aproximación de π mejora a medida que aumenta n, la convergencia sigue siendo lenta debido a la naturaleza de la serie de Leibniz.
- En comparación con la ejecución en paralelo con múltiples hilos, donde cada hilo calcula y suma términos de la serie de Leibniz de manera concurrente, la ejecución con un solo hilo es más lenta y no aprovecha la capacidad de procesamiento paralelo de la CPU.

f.

Medición	n	thread_count	Aprox de pi	Dif. con pi real
1	10000000	2	3.1415926534823	0.0000000000176
2	20000000	4	3.1415926534735	0.0000000000264
3	50000000	4	3.1415926534883	0.0000000000116
4	100000000	8	3.1415926534877	0.0000000000122
5	200000000	8	3.1415926534806	0.0000000000193



Observaciones:

- A medida que aumenta el valor de n y se utilizan más hilos, la aproximación de π se acerca más al valor real de π.
- La mejora en precisión en comparación con valores más pequeños de n es notable, y la diferencia entre la aproximación y el valor real es cada vez más pequeña.
- Aumentar la cantidad de hilos (thread_count) puede acelerar el cálculo, lo que llevará a una aproximación más rápida del valor de π. Sin embargo, después de cierto punto, la ganancia en velocidad puede ser menos pronunciada debido a la necesidad de sincronización y a las limitaciones del sistema.
- La cláusula private() resuelve el problema de ámbito en la versión paralela, permitiendo que los hilos realicen cálculos independientes de manera segura.

g.

1 thread Secuencial	threads = cores	threads = 2*cores	n=10*n threads = cores
0.005042	0.001318	0.004613	0.011959
0.003711	0.0014229	0.005751	0.009256
0.003319	0.001736	0.002728	0.011857
0.003658	0.001793	0.007999	0.009852
0.004953	0.001833	0.003470	0.012703
0.003350	0.001910	0.003170	0.009167

Tiempo Secuencial:

Promedio Secuencial: 0.004080

Tiempos paralelos:

Promedio Threads = Core: 0.001674

Promedio Threads = 2*cores: 0.004382

Promedio n=10*n, threads = cores: 0.010540

	Threads=Core=4 n = 1000000	Threads=2*Core=8 n = 1000000	Threads=Core=4 n = 10*1000000
SpeedUp	2.43728	0.93108	0.38709
Eficiencia	0.60931	0.11639	0.09677
Escalabilidad fuerte	0.60931	0.11639	0.09677
Escalabilidad débil	9.749*10^-6	7.449*10^-6	1.548*10^-6

h.

Auto

Paralela	Secuencial
0.014534	0.031380
0.013640	0.030832
0.012681	0.033625
0.010399	0.032093
0.018704	0.040953

Static

Paralela			Secuencial		
16	**64**	**128**	**16**	**64**	**128**
0.013750	0.011423	0.015698	0.035530	0.033934	0.041550
0.009364	0.011313	0.010964	0.031342	0.045061	0.035936
0.014572	0.010999	0.012916	0.035664	0.034421	0.050894
0.009109	0.013150	0.010484	0.033297	0.039572	0.033544
0.011519	0.010741	0.010342	0.042618	0.037855	0.038532

Dynamic

Paralela			Secuencial		
16	**64**	**128**	**16**	**64**	**128**
0.010322	0.012319	0.010979	0.034432	0.037526	0.036621
0.013743	0.010322	0.011673	0.045749	0.037849	0.036933
0.010205	0.013628	0.010363	0.033073	0.033929	0.039348
0.013252	0.011024	0.013084	0.040436	0.036879	0.037182
0.011037	0.012334	0.010955	0.047208	0.037847	0.035458

Guided

Paralela			Secuencial		
16	**64**	**128**	**16**	**64**	**128**
0.013364	0.011835	0.010786	0.061256	0.042898	0.035565
0.009866	0.011978	0.009268	0.046706	0.038389	0.030109
0.011496	0.009356	0.009699	0.045049	0.050448	0.035224
0.011615	0.010800	0.0088850	0.046098	0.054203	0.034428
0.013614	0.012082	0.010408	0.068850	0.033080	0.029378

SpeedUps

- Estos se obtienen utilizando el promedio de las columnas de cada caso.

	16	**64**	**128**
Auto	2.414		
Static	3.060174	3.311752	3.318588
Dynamic	3.430694	3.086353	3.252042
Guided	4.102977	3.907477	3.358154

- Se obtuvo un mejor rendimiento con la política de Guided, tanto en términos de SpeedUp y velocidad neta. En el caso de Guided con un block size de 16, se obtuvo el mejor SpeedUp de todos. En cuanto a velocidad neta. Guided con un block size de 128 fue el más rápido de todos, con un tiempo promedio de 0.009809 segundos.

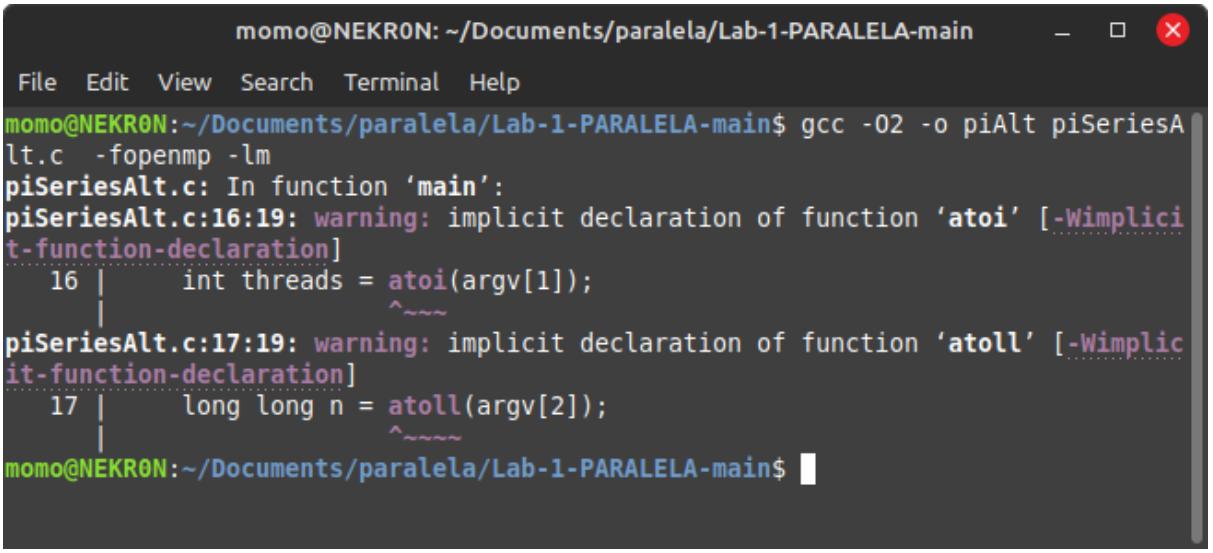
Ejercicio 2:

a.

Medición	n	thread_count	Aprox de pi	Dif. con pi real
1	10000000	2	3.14159275358705514236	0.00000009999726202636
2	20000000	4	3.14159270358709719062	0.00000004999730407462
3	50000000	4	3.14159267358378357926	0.00000001999399046326
4	100000000	8	3.14159266358378275186	0.00000000999398963586
5	200000000	8	3.14159265858252112480	0.00000000499272800880

- Al aumentar el número de hilos de ejecución, se observa una disminución en el tiempo de ejecución gracias a la paralelización efectiva del cálculo, aunque los valores calculados para π varíen ligeramente en cada iteración debido a las imprecisiones inherentes a los cálculos en punto flotante y al orden de finalización de los hilos. A medida que el valor de "n" aumenta, la precisión de π tiende a mejorar gradualmente, acercándose al valor real de 3.14159265358979323846, a pesar de que no se logre una correspondencia exacta debido a las limitaciones de aritmética en punto flotante.

b.



Variables		Tiempo (segundos)	
Threads	n	-O2	Sin optimizar
2	10000000	0.008928	0.023398
4	20000000	0.008276	0.032414
4	50000000	0.025956	0.077151
8	100000000	0.066561	0.142354
8	200000000	0.112977	0.241343

- Tras implementar y compilar el algoritmo con configuraciones estándar y la bandera de optimización -O2 de GCC, se notó una mejora notable en el tiempo de ejecución. La versión optimizada superó significativamente a la no optimizada. Estos resultados demuestran la habilidad del compilador para mejorar secciones del código. Sin embargo, se encontró que depurar el código optimizado era más complejo. Esto destaca el balance crucial entre eficiencia en tiempo de ejecución y facilidad de depuración en el desarrollo de software. Esta mejora es especialmente relevante para aplicaciones a gran escala, donde pequeños ahorros de tiempo pueden reducir significativamente el tiempo total de computación.