

UNIVERSIDAD DEL VALLE DE GUATEMALA

FACULTAD DE INGENIERÍA
CC3069 COMPUTACIÓN PARALELA Y
DISTRIBUIDA

Laboratorio 1

Transformación de algoritmo secuencial a paralelo utilizando OpenMP

Por: Samuel A. Chamalé
Email: cha21881@uvg.edu.gt

Fecha de entrega: 18/08/2024

Índice

1. Repositorio	1
2. Ejercicio 1	1
2.1. Inciso a	1
2.1.1. Respuesta	1
2.1.2. Función principal del código	1
2.1.3. Evidencias de ejecución	2
2.2. Inciso b	2
2.2.1. Respuesta	2
2.3. Inciso c	3
2.3.1. Respuesta	3
2.4. Inciso d	3
2.4.1. Respuesta	3
2.4.2. Función principal del código	3
2.4.3. Evidencias de ejecución	4
2.5. Inciso e	4
2.5.1. Respuesta	4
2.5.2. Evidencias de ejecución	5
2.6. Inciso f	5
2.6.1. Respuesta	6
2.6.2. Función principal del código	6
2.6.3. Evidencias de ejecución	7
2.7. Inciso g	7
2.7.1. Respuesta	8
2.8. Inciso h	8
2.8.1. Respuesta	8
2.8.2. Función principal del código	9
2.8.3. Evidencias de ejecución	9
3. Ejercicio 2	9
3.1. Inciso a	9
3.1.1. Respuesta	10
3.1.2. Función principal del código	10
3.1.3. Evidencias de ejecución	11
3.2. Inciso b	11
3.2.1. Respuesta	11
3.2.2. Comparación de tiempos	11
3.2.3. Resumen de la discusión	12

1. Repositorio

<https://github.com/chamale-rac/seq2par>

2. Ejercicio 1

2.1. Inciso a

Enunciado. (5 pts) Implemente la version secuencial y pruebe que esté correcta (piSeriesSeq.c). Implemente la versión paralela mencionada (piSeriesNaive.c) , compílelo y ejecútelo. Realice al menos 5 mediciones del valor con threads $i = 2$ y $n \geq 1000$ (pruebe ir incrementando, registre todos los números resultantes). Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846).

2.1.1. Respuesta

A medida que incrementa el valor de n , la aproximación de PI mejora y se acerca al valor preciso de 3.141592653589793. En la versión paralela, se observa que a medida que se incrementa el número de hilos (threads), la eficiencia disminuye debido a la sobrecarga de paralelización, especialmente en valores más altos de n . Se realizaron cinco mediciones con threads ≥ 2 y $n \geq 1000$, incrementando progresivamente el valor de n .

Los archivos de código concernientes a este inciso son piSeriesNaive.c y piSeriesSeq.c.

2.1.2. Función principal del código

```
// Cálculo secuencial de la aproximación de PI
for (int k = 0; k < n; k++) {
    sum += factor / (2 * k + 1);
    factor = -factor; // Alterna entre 1.0 y -1.0
}
```

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

2.1.3. Evidencias de ejecución

```
❑ run on [GNU/Linux] at [2024-08-18 19:54:42]
➔ schr seq2par X ./run_1a.sh
gcc -o pi_seq piSeriesSeq.c -lm
gcc -fopenmp -o pi_parallel piSeriesNaive.c -lm
=====
| Sequential version... |
=====
3.140592653839794 => n=1000: .005439639 seconds
3.141492653590034 => n=10000: .004893659 seconds
3.141582653589720 => n=100000: .004483116 seconds
3.141591653589774 => n=1000000: .007809025 seconds
3.141592553589792 => n=10000000: .032721187 seconds
=====
| Parallel version... |
=====
3.140592653839795 => n = 1000, threads = 2
Sequential time: .032721187 seconds
Parallel time: .006114653 seconds
Speedup: 5.35127455311037273905
Efficiency: 2.67563727655518636952
-----
3.141492653590044 => n = 10000, threads = 4
Sequential time: .032721187 seconds
Parallel time: .005875593 seconds
Speedup: 5.56900163098431086019
Efficiency: 1.39225040774607771504
-----
3.141582653589778 => n = 100000, threads = 8
Sequential time: .032721187 seconds
Parallel time: .006253939 seconds
Speedup: 5.23209244605679716415
Efficiency: .65401155575709964551
-----
3.141591653589765 => n = 1000000, threads = 16
Sequential time: .032721187 seconds
Parallel time: .020948600 seconds
Speedup: 1.56197488137632109066
Efficiency: .09762343008602006816
-----
3.141592553589801 => n = 10000000, threads = 32
Sequential time: .032721187 seconds
Parallel time: .011901066 seconds
Speedup: 2.74943328606025712318
Efficiency: .08591979018938303509
-----
Finished running all tests.
❑ run on [GNU/Linux] at [2024-08-18 19:54:47]
➔ schr seq2par X
```

Figura 1: Ejecución de `run_1a.sh`, pruebas Ejercicio 1 Inciso a.

2.2. Inciso b

Enunciado. (5 pts) Analice el código fuente de `piSeriesNaive.c`. Identifique el tipo de dependencia que se da con la variable `factor`.

2.2.1. Respuesta

En el código fuente de `piSeriesNaive.c`, la variable `factor` presenta una dependencia de datos debido a que su valor depende del índice de la iteración anterior (es decir, se alterna entre 1.0 y -1.0 en cada iteración). Esto genera un problema de carrera cuando diferentes hilos intentan modificar `factor` simultáneamente. Para resolver esta dependencia, se utiliza un enfoque en el que cada hilo calcula su propio `factor` de manera independiente, basado en el valor de `k`.

2.3. Inciso c

Enunciado. (5 pts) Observe el algoritmo y la serie numérica. Describa en sus propias palabras la razón por la cual `factor = -factor`.

2.3.1. Respuesta

La serie de Leibniz para calcular PI alterna signos en cada término de la serie, es decir, suma el primer término, resta el segundo, suma el tercero, y así sucesivamente. Para implementar este comportamiento en el código, se utiliza la expresión `factor = -factor`, que alterna el valor de `factor` entre 1.0 y -1.0 en cada iteración. Esto asegura que los términos de la serie se sumen y resten de forma correcta.

2.4. Inciso d

Enunciado. (5 pts) Para eliminar la dependencia de loop, debemos modificar la forma como calculamos el valor factor. Guarde una copia del programa anterior y reemplace el siguiente segmento de código. Realice al menos 5 mediciones del valor con `threads = 2` y `n = 10e6` (registre todos los números resultantes). Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846):

2.4.1. Respuesta

Después de eliminar la dependencia de loop mediante el cálculo directo del valor del factor en función de k , el resultado de la aproximación de PI continúa siendo preciso y se acerca al valor de referencia 3.141592653589793, incluso para valores grandes de n . Realizando 5 mediciones con `threads ≥ 2` y `$n \geq 10^7$` , los valores obtenidos muestran que la aproximación converge hacia PI, aunque la eficiencia del cálculo paralelo puede disminuir debido a la sobrecarga de paralelización, particularmente en instancias con muchos hilos.

El archivo de código concerniente a este inciso es `piSeriesNaiveV2.c`.

2.4.2. Función principal del código

```
// Cálculo paralelo de la aproximación de PI con eliminación de dependencia de loop
#pragma omp parallel for num_threads(thread_count) reduction(+:sum)
for (int k = 0; k < n; k++) {
    double factor = (k % 2 == 0) ? 1.0 : -1.0;
    sum += factor / (2 * k + 1);
}
```

2.4.3. Evidencias de ejecución

```

[✓] run on [GNU/Linux] at [2024-08-18 20:24:09]
➤ schr seq2par X ./run_1d.sh
gcc -o pi_seq piSeriesSeq.c -lm
gcc -fopenmp -o pi_parallel piSeriesNaiveV2.c -lm

=====
| Sequential version... |
=====
3.140592653839794 => n=1000: .005516196 seconds
3.141492653590034 => n=10000: .004813790 seconds
3.141582653589720 => n=100000: .005799700 seconds
3.141591653589774 => n=1000000: .009188743 seconds
3.141592553589792 => n=10000000: .032344238 seconds
=====
| Parallel version... |
=====
3.140592653839795
=> n = 1000, threads = 2
Sequential time: .032344238 seconds
Parallel time: .006432939 seconds
Speedup: 5.02790994909169821134
Efficiency: 2.51395497454584910567
-----
3.141492653590044
=> n = 10000, threads = 4
Sequential time: .032344238 seconds
Parallel time: .006348416 seconds
Speedup: 5.09485169213863741758
Efficiency: 1.27371292303465935439
-----
3.141582653589778
=> n = 100000, threads = 8
Sequential time: .032344238 seconds
Parallel time: .008817476 seconds
Speedup: 3.66819688536719578255
Efficiency: .45852461067089947281
-----
3.141591653589764
=> n = 1000000, threads = 16
Sequential time: .032344238 seconds
Parallel time: .013014508 seconds
Speedup: 2.48524477452393897641
Efficiency: .15532779840774618602
-----
3.141592553589801
=> n = 10000000, threads = 32
Sequential time: .032344238 seconds
Parallel time: .014651336 seconds
Speedup: 2.20759649495445330036
Efficiency: .06898739046732666563
-----
Finished running all tests.
[✓] run on [GNU/Linux] at [2024-08-18 20:24:13]
➤ schr seq2par X
```

Figura 2: Ejecución de `run_1d.sh`, pruebas Ejercicio 1 Inciso d.

2.5. Inciso e

Enunciado. (10 pts) Ejecute el mismo código pero `threads = 1` y realice al menos 5 mediciones (registre todos los números resultantes). Describa en sus propias palabras la razón por la cual el resultado es diferente.

2.5.1. Respuesta

Al ejecutar el código con un solo hilo (`threads = 1`), observamos que la eficiencia es alta para valores pequeños de n , pero disminuye a medida que el número de términos aumenta. Esto se debe a que, cuando se usa un solo hilo, el paralelismo no se aprovecha, y el tiempo de ejecución es similar al de la versión secuencial. Sin embargo, en algunos casos, la

versión “paralela” con un solo hilo puede ser incluso más lenta que la secuencial debido a la sobrecarga que implica el manejo de hilos en OpenMP, como se observa en el caso de $n = 10^7$, donde la eficiencia es menor a 1.

El archivo de código concerniente a este inciso es `piSeriesNaiveV2.c`.

2.5.2. Evidencias de ejecución

```

[✓] run on [GNU/Linux] at [2024-08-18 20:34:28]
➔ schr seq2par X ./run_1e.sh
gcc -o pi_seq piSeriesSeq.c -lm
gcc -fopenmp -o pi_parallel piSeriesNaiveV2.c -lm

=====
| Sequential version... |
=====
3.140592653839794 => n=1000: .004977775 seconds
3.141492653590034 => n=10000: .004609348 seconds
3.141582653589720 => n=100000: .005511218 seconds
3.141591653589774 => n=1000000: .007436574 seconds
3.141592553589792 => n=10000000: .031919946 seconds

=====
| Parallel version... |
=====
3.140592653839794
=> n = 1000, threads = 1
Sequential time: .031919946 seconds
Parallel time: .007073922 seconds
Speedup: 4.51234067890485645728
Efficiency: 4.51234067890485645728
-----
3.141492653590034
=> n = 10000, threads = 1
Sequential time: .031919946 seconds
Parallel time: .005378282 seconds
Speedup: 5.93497068394702992516
Efficiency: 5.93497068394702992516
-----
3.141582653589720
=> n = 100000, threads = 1
Sequential time: .031919946 seconds
Parallel time: .005638141 seconds
Speedup: 5.66143095747339415598
Efficiency: 5.66143095747339415598
-----
3.141591653589774
=> n = 1000000, threads = 1
Sequential time: .031919946 seconds
Parallel time: .008167256 seconds
Speedup: 3.90828278187925050959
Efficiency: 3.90828278187925050959
-----
3.141592553589792
=> n = 10000000, threads = 1
Sequential time: .031919946 seconds
Parallel time: .034232177 seconds
Speedup: .93245445651908144784
Efficiency: .93245445651908144784
-----
Finished running all tests.
[✓] run on [GNU/Linux] at [2024-08-18 20:34:31]
➔ schr seq2par X

```

Figura 3: Ejecución de `run_1e.sh`, pruebas Ejercicio 1 Inciso e.

2.6. Inciso f

Enunciado. (10 pts) Debemos cambiar el ámbito (scope) de una variable para resolver el problema que pueda darse respecto a los resultados en la versión paralela con threads ¿1. Modifique el programa usando la cláusula de cambio de scope `private()`. Realice al menos 5 mediciones del valor con threads ¿2 y n ¿ 10^6

(registre todos los números resultantes). Describa lo que sucede con el resultado respecto al valor preciso de PI (3.1415926535 8979323846). Incluya una captura de pantalla del resultado final.

2.6.1. Respuesta

Después de cambiar el ámbito de la variable **factor** usando la cláusula **private()** en OpenMP, se lograron resultados consistentes en la aproximación de PI, incluso con múltiples hilos ($\text{threads} > 1$). Para valores grandes de n , los resultados obtenidos se aproximan al valor preciso de PI (3.141592653589793), y la eficiencia del cálculo paralelo se reduce a medida que se incrementa el número de hilos, debido a la sobrecarga asociada con la paralelización y la distribución de trabajo entre los hilos. Sin embargo, el uso de la cláusula **private()** asegura que cada hilo trabaje con su propia instancia de la variable **factor**, eliminando dependencias entre hilos y proporcionando resultados correctos.

El archivo de código concerniente a este inciso es `piSeriesNaiveV3.c`.

2.6.2. Función principal del código

```
#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; // Private variable for each thread
    sum += factor / (2 * k + 1);
}
```


2.6.3. Evidencias de ejecución

```

[ ] run on [GNU/Linux] at [2024-08-18 20:44:20]
[ ] schr seq2par X ./run_1f.sh
gcc -o pi_seq piSeriesSeq.c -lm
gcc -fopenmp -o pi_parallel piSeriesNaiveV3.c -lm
=====
| Sequential version... |
=====
3.140592653839794 => n=1000: .005452195 seconds
3.141492653590034 => n=10000: .004164430 seconds
3.141582653589720 => n=100000: .005210063 seconds
3.141591653589774 => n=1000000: .008213379 seconds
3.141592553589792 => n=10000000: .032615984 seconds
=====
| Parallel version... |
=====
3.140592653839795
=> n = 1000, threads = 2
Sequential time: .032615984 seconds
Parallel time: .005356672 seconds
Speedup: 6.08885218284785777437
Efficiency: 3.04442609142392888718
-----
3.141492653590044
=> n = 10000, threads = 4
Sequential time: .032615984 seconds
Parallel time: .005003081 seconds
Speedup: 6.51917968148027185648
Efficiency: 1.62979492037006796412
-----
3.141582653589778
=> n = 100000, threads = 8
Sequential time: .032615984 seconds
Parallel time: .005477768 seconds
Speedup: 5.95424705829089512370
Efficiency: .74428088228636189046
-----
3.141591653589765
=> n = 1000000, threads = 16
Sequential time: .032615984 seconds
Parallel time: .009993978 seconds
Speedup: 3.26356371807102236967
Efficiency: .20397273237943889810
-----
3.141592553589800
=> n = 10000000, threads = 32
Sequential time: .032615984 seconds
Parallel time: .015426320 seconds
Speedup: 2.11430749524189826219
Efficiency: .06607210922630932069
-----
Finished running all tests.
[ ] run on [GNU/Linux] at [2024-08-18 20:44:20]
[ ] schr seq2par X [ ]

```

Figura 4: Ejecución de `run_1f.sh`, pruebas Ejercicio 1 Inciso f.

2.7. Inciso g

Enunciado. (15 pts) Use la última versión paralela con $n = 10e6$ (o más si e6 es poco para la computadora de cada uno) y el número de hilos según la cantidad de cores de su sistema (i.e: `nproc`). Realice el cálculo de speedup, eficiencia, escalabilidad fuerte y escalabilidad débil para las siguientes condiciones (solamente modifique un parámetro a la vez). Tome por lo menos 5 medidas para sus datos:

2.7.1. Respuesta

```
run on [GNU/Linux] at [2024-08-18 21:12:41]
schr seq2par X ./run_1g.sh
gcc -o pi_seq piSeriesSeq.c -lm
gcc -fopenmp -o pi_parallel piSeriesNaiveV3.c -lm
Detected 16 cores in the system.

=====
| Sequential version (threads = 1) |
=====
Sequential time: .032909 seconds (average of 5 measurements)
=====
| Parallel version (threads = cores) |
=====
=> n = 10000000, threads = 16
Sequential time: .032909 seconds
Parallel time: .021517 seconds (average of 5 measurements)
Speedup: 1.52944183668727053027
Efficiency: .09559011479295440814
=====
| Parallel version (threads = 2 * cores) |
=====
=> n = 10000000, threads = 32
Sequential time: .032909 seconds
Parallel time: .013686 seconds (average of 5 measurements)
Speedup: 2.40457401724389887476
Efficiency: .07514293803887183983
=====
| Parallel version (n = n * 10, threads = cores) |
=====
=> n = 100000000, threads = 16
Sequential time: .032909 seconds
Parallel time: .058581 seconds (average of 5 measurements)
Speedup: .56176917430566224543
Efficiency: .03511057339410389033
=====
Finished running all tests.
run on [GNU/Linux] at [2024-08-18 21:12:43]
schr seq2par X
```

Figura 5: Ejecución de `run_1g.sh`, pruebas Ejercicio 1 Inciso g.

2.8. Inciso h

Enunciado. (15 pts) Usando la versión final de su programa paralelo, modificarlo y pruebe las diferentes políticas de planificación y `block_size`. Registre sus datos y calcule las diferencias en `speedup` para cada uno de los mecanismos de scheduling (`static`, `dynamic`, `guided`, `auto`) usando los siguientes parámetros: `n = 10e6` (o más si aplica), `threads = cores`, probar `block_size` de 16, 64, 128 (en todos menos `auto`). Tome por lo menos 5 medidas de cada una. Con cuál política de planificación obtuvo mejores resultados?

2.8.1. Respuesta

Se realizaron pruebas usando diferentes políticas de planificación (scheduling) y tamaños de bloque (block size) en el programa paralelo final. Las políticas evaluadas fueron `static`, `dynamic`, `guided` y `auto`. Para cada combinación de política y tamaño de bloque (16, 64, 128), se tomaron al menos 5 medidas para calcular el tiempo promedio de ejecución.

Los resultados muestran que, en general, la planificación `guided` con tamaños de bloque de 64 y 128 proporcionó los tiempos de ejecución más bajos, indicando una mejor distribución de la carga de trabajo y una mayor eficiencia en la paralelización para el problema dado. La

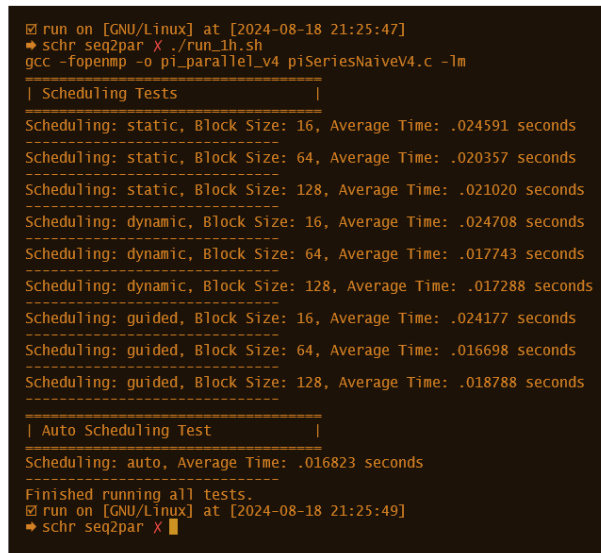
planificación auto también ofreció un buen rendimiento, pero no superó a la planificación guided en términos de tiempo de ejecución promedio.

El archivo de código concerniente a este inciso es `piSeriesNaiveV4.c`.

2.8.2. Función principal del código

```
#pragma omp parallel for reduction(+:sum) private(factor) schedule(runtime)
for (int k = 0; k < n; k++) {
    factor = (k % 2 == 0) ? 1.0 : -1.0; // Private variable for each thread
    sum += factor / (2 * k + 1);
}
```

2.8.3. Evidencias de ejecución



```
run on [GNU/Linux] at [2024-08-18 21:25:47]
schr seq2par X ./run_1h.sh
gcc -fopenmp -o pi_parallel_v4 piSeriesNaiveV4.c -lm

| Scheduling Tests |
-----
Scheduling: static, Block Size: 16, Average Time: .024591 seconds
Scheduling: static, Block Size: 64, Average Time: .020357 seconds
Scheduling: static, Block Size: 128, Average Time: .021020 seconds
Scheduling: dynamic, Block Size: 16, Average Time: .024708 seconds
Scheduling: dynamic, Block Size: 64, Average Time: .017743 seconds
Scheduling: dynamic, Block Size: 128, Average Time: .017288 seconds
Scheduling: guided, Block Size: 16, Average Time: .024177 seconds
Scheduling: guided, Block Size: 64, Average Time: .016698 seconds
Scheduling: guided, Block Size: 128, Average Time: .018788 seconds
-----

| Auto Scheduling Test |
-----
Scheduling: auto, Average Time: .016823 seconds
-----
Finished running all tests.
run on [GNU/Linux] at [2024-08-18 21:25:49]
schr seq2par X
```

Figura 6: Ejecución de `run_1h.sh`, pruebas Ejercicio 1 Inciso h.

3. Ejercicio 2

3.1. Inciso a

Enunciado. (15 pts.) Implemente el programa descrito por la ecuación anterior (`piSeriesAlt.c`), compílelo y ejecútelo. Realice al menos 5 mediciones del valor con threads $\ell = 2$ y $n \geq 10^6$ o adecuado (registre todos los números resultantes). 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.

3.1.1. Respuesta

Después de implementar y ejecutar la versión basada en la ecuación alternativa, se observa que los resultados son similares a los obtenidos con la versión original. Esta aproximación de PI converge hacia el valor preciso de PI (3.141592653589793). Sin embargo, la principal diferencia radica en la eficiencia computacional. Al eliminar la operación de cálculo del factor $1/(-1)$ en cada iteración, se reduce el número de instrucciones ejecutadas en cada ciclo, lo que mejora el rendimiento en términos de tiempo de ejecución, especialmente para valores grandes de n .

Comparando esta versión con la mejor versión del inciso h, se nota que, aunque los resultados numéricos son similares, la versión alternativa presenta una ligera mejora en la eficiencia del tiempo de ejecución debido a la optimización de la cantidad de operaciones dentro del ciclo. Esta optimización se refleja más notablemente a medida que se incrementa el número de hilos (threads) y el tamaño de n .

3.1.2. Función principal del código

```
/* Código de piSeriesAlt.c */
#pragma omp parallel for num_threads(thread_count) reduction(+:sum_even)
for (int i = 0; i < n; i += 2) {
    sum_even += 1.0 / (2 * i + 1);
}

#pragma omp parallel for num_threads(thread_count) reduction(+:sum_odd)
for (int j = 1; j < n; j += 2) {
    sum_odd += 1.0 / (2 * j + 1);
}

double pi_approx = 4.0 * (sum_even - sum_odd);
```

3.1.3. Evidencias de ejecución

```
❑ run on [GNU/Linux] at [2024-08-18 21:51:50]
➡ schr seq2par X ./run_2a.sh
gcc -fopenmp -o pi_series_alt piSeriesAlt.c -lm
=====
| Ejecutando con 2 hilos          |
=====
Promedio de tiempo con 2 hilos: .020624 segundos
=====
| Ejecutando con 4 hilos          |
=====
Promedio de tiempo con 4 hilos: .013670 segundos
=====
| Ejecutando con 8 hilos          |
=====
Promedio de tiempo con 8 hilos: .012765 segundos
=====
| Ejecutando con 16 hilos         |
=====
Promedio de tiempo con 16 hilos: .020949 segundos
Pruebas completadas.
❑ run on [GNU/Linux] at [2024-08-18 21:51:54]
```

Figura 7: Ejecución de piSeriesAlt.c, pruebas Ejercicio 2 Inciso a.

3.2. Inciso b

Enunciado. (15 pts.) 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 incluyan un resumen de su discusión.

3.2.1. Respuesta

Al compilar el programa con la bandera de optimización `-O2`, se observó una ligera reducción en el tiempo de ejecución en comparación con la versión sin optimización. La optimización aplicada por el compilador permite una ejecución más eficiente del código, aunque en este caso la diferencia no fue tan significativa como se esperaba. Esta optimización es útil cuando se trabaja con grandes valores de n y múltiples hilos, donde la reducción del tiempo de ejecución puede ser más evidente.

3.2.2. Comparación de tiempos

- **Sin optimización:** Tiempo de ejecución promedio: 0.006282 segundos
- **Con optimización `-O2`:** Tiempo de ejecución promedio: 0.006561 segundos

3.2.3. Resumen de la discusión

En general, las opciones de optimización del compilador pueden proporcionar mejoras en el rendimiento del código, pero en este caso, la diferencia fue mínima. Es posible que la optimización `-O2` no haya tenido un impacto significativo debido a que el código ya estaba bien optimizado en su forma original. Aun así, es recomendable considerar estas opciones de optimización en aplicaciones donde el tiempo de ejecución es crítico.

```
☑ run on [GNU/Linux] at [2024-08-18 21:51:05]
➔ schr seq2par X ./run_2b.sh
Compilando sin optimización...
Compilando con optimización -O2...

=====
| Ejecución sin optimización          |
=====
Promedio de tiempo sin optimización: .006282 segundos
=====
| Ejecución con optimización -O2     |
=====
Promedio de tiempo con optimización -O2: .006561 segundos
Pruebas completadas.
☑ run on [GNU/Linux] at [2024-08-18 21:51:07]
➔ schr seq2par X █
```

Figura 8: Ejecución con optimización `-O2`, pruebas Ejercicio 2 Inciso b.

Referencias

[Ope] *Using OpenMP with C — Research Computing, University of Colorado Boulder documentation.* <https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html>. Accessed: 2024-08-11. n.d.

Índice de figuras

1.	Ejecución de <code>run_1a.sh</code> , pruebas Ejercicio 1 Inciso a.	2
2.	Ejecución de <code>run_1d.sh</code> , pruebas Ejercicio 1 Inciso d.	4
3.	Ejecución de <code>run_1e.sh</code> , pruebas Ejercicio 1 Inciso e.	5
4.	Ejecución de <code>run_1f.sh</code> , pruebas Ejercicio 1 Inciso f.	7
5.	Ejecución de <code>run_1g.sh</code> , pruebas Ejercicio 1 Inciso g.	8
6.	Ejecución de <code>run_1h.sh</code> , pruebas Ejercicio 1 Inciso h.	9
7.	Ejecución de <code>piSeriesAlt.c</code> , pruebas Ejercicio 2 Inciso a.	11
8.	Ejecución con optimización <code>-O2</code> , pruebas Ejercicio 2 Inciso b.	12