Diego Lemus - 21469

Laboratorio 1

Computación Paralela y Distribuida

Repositorio

Acceso: https://github.com/TheDeloz-v2/CPD-LAB1

Ejercicio 1

Inciso a.

```c
    clock_t start, end;
    double cpu_time_used;

    start = clock();

    double factor = 1.0;
    double sum = 0.0;
    int n = 1000;

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

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS  COMMENTS  DEBUG CONSOLE

root@d78ceaaf3fb5:/usr/src/app# gcc -o piSeriesSeq piSeriesSeq.c
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesSeq
PI aproximation: 3.140592653839794
Tiempo de ejecución: 0.000014 segundos
root@d78ceaaf3fb5:/usr/src/app#
```

```c
    double start, end;
    double factor = 1.0;
    double sum = 0.0;
    int n, thread_count;

    if (argc != 3) {
        printf("Usage: %s <n> <thread_count>\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]);
    thread_count = atoi(argv[2]);

    start = omp_get_wtime();

    #pragma omp parallel for num_threads(thread_count) reduction(+:sum)
    for (int k = 0; k < n; k++) {
        sum += factor/(2*k+1);
        factor = factor * -1.0;
    }
    double pi_approx = 4.0*sum;

    end = omp_get_wtime();
```

```
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 1000 2
PI approximation 3.1405926538
- Threads: 2
- N value: 1000
Execution time: 0.000124 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 10000 2
PI approximation 3.1414926536
- Threads: 2
- N value: 10000
Execution time: 0.000103 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 100000 4
PI approximation 3.1415826536
- Threads: 4
- N value: 100000
Execution time: 0.000796 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 1000000 4
PI approximation 3.1415916536
- Threads: 4
- N value: 1000000
Execution time: 0.001069 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 1000000 6
PI approximation 3.1415916536
- Threads: 6
- N value: 1000000
Execution time: 0.000771 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 10000000 6
PI approximation 3.1415925536
- Threads: 6
- N value: 10000000
Execution time: 0.006380 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive 10000000 8
PI approximation 3.1415925536
- Threads: 8
- N value: 10000000
Execution time: 0.009777 s
root@d78ceaaf3fb5:/usr/src/app#
```

Para n = 1000000, la aproximación es 3.1415925536, que es muy cercana al valor real de pi, con solo una pequeña desviación en las últimas cifras. El número de hilos no afecta directamente la precisión de la aproximación, sino el tiempo de ejecución.

El resultado muestra que, para aproximar pi con alta precisión, es necesario un valor grande de n. El número de hilos ayuda a reducir el tiempo de cálculo pero no afecta la exactitud de la aproximación.

Inciso b.

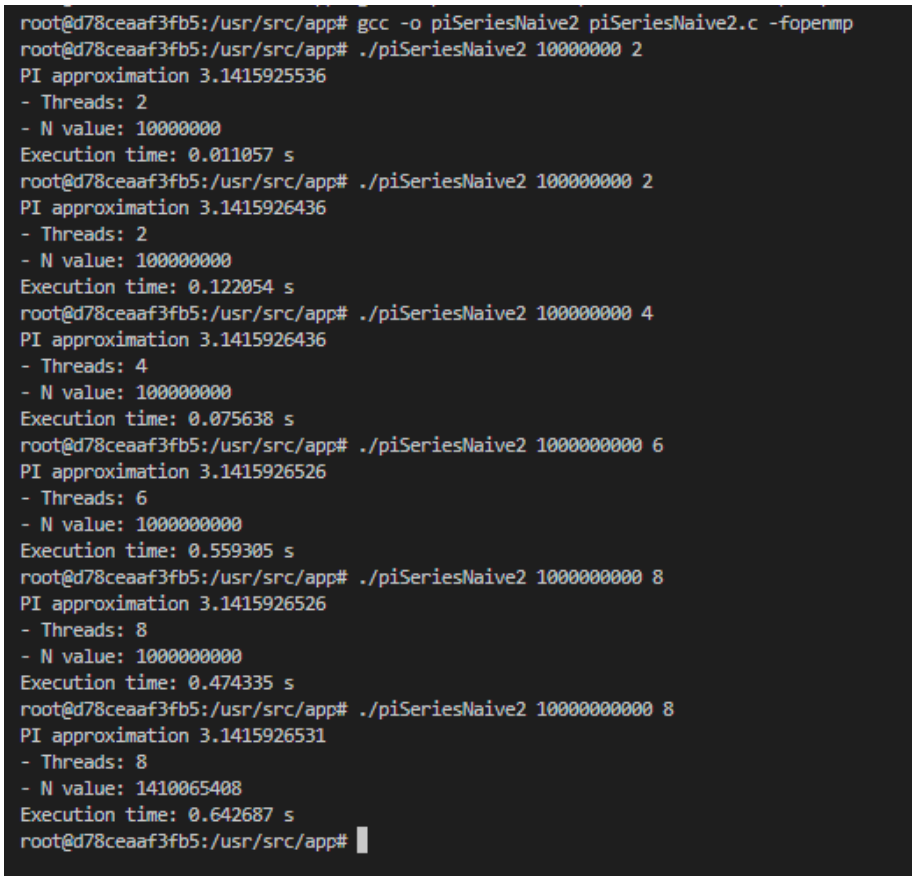
En este caso, la variable factor introduce una dependencia de flujo. Esta dependencia ocurre porque el valor de factor en cada iteración del bucle depende del valor que tenía en la iteración anterior. Esta dependencia causa un problema porque varios hilos pueden intentar acceder y modificar la variable factor simultáneamente, y dado que la variable factor es compartida entre los hilos, esto puede llevar a race conditions. Este problema puede dar lugar a un cálculo incorrecto de la serie para la aproximación de pi.

Inciso c.

La razón por la cual se utiliza factor = -factor es para asegurar que los términos de la serie numérica alternen entre positivo y negativo, lo que es necesario para que la serie de Leibniz aproxime correctamente el valor de pi. Sin este cambio de signos, la suma simplemente crecería indefinidamente o sería incorrecta.

Inciso d.

```
6     double start, end;
7     double sum = 0.0;
8     int n, thread_count;
9
10    if (argc != 3) {
11        printf("Usage: %s <n> <thread_count>\n", argv[0]);
12        return 1;
13    }
14
15    n = atoi(argv[1]);
16    thread_count = atoi(argv[2]);
17
18    start = omp_get_wtime();
19
20    #pragma omp parallel for num_threads(thread_count) reduction(+:sum)
21    for (int k = 0; k < n; k++) {
22        double factor = (k % 2 == 0) ? 1.0 : -1.0;
23        sum += factor / (2 * k + 1);
24    }
25
26    end = omp_get_wtime();
27
28    double pi_approx = 4.0 * sum;
```



El valor aproximado de pi obtenido en cada ejecución es 3.141592653, que es extremadamente cercano al valor preciso de pi. La precisión se mantiene consistente a lo largo de las diferentes ejecuciones, casi independientemente del número de hilos utilizados o del valor de n. Al usar n = 1000000000, la aproximación de pi es muy precisa, con una desviación mínima en las últimas cifras decimales. Incluso con 2, 4, 6, y 8 hilos, la aproximación de pi se mantiene consistentemente precisa, mostrando que el paralelismo no introduce errores significativos en la aproximación.

Inciso e.

```
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive2 10000000 1
PI approximation 3.1415925536
- Threads: 1
- N value: 10000000
Execution time: 0.024958 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive2 100000000 1
PI approximation 3.1415926436
- Threads: 1
- N value: 100000000
Execution time: 0.225376 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive2 1000000000 1
PI approximation 3.1415926526
- Threads: 1
- N value: 1000000000
Execution time: 2.261760 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive2 10000000000 1
PI approximation 3.1415926531
- Threads: 1
- N value: 1410065408
Execution time: 3.172715 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive2 100000000000 1
PI approximation 3.1415926528
- Threads: 1
- N value: 1215752192
Execution time: 2.758422 s
```

Al observar los resultados obtenidos cuando se ejecuta el código con un solo hilo, el tiempo de ejecución es significativamente mayor. Esto se debe a que no se está aprovechando el paralelismo para dividir el trabajo entre múltiples núcleos de la CPU. A pesar del aumento en el tiempo de ejecución, la aproximación de pi se mantiene extremadamente precisa, con un valor de 3.1415926531 o cercano, lo que coincide con las ejecuciones anteriores en términos de precisión. El tiempo de ejecución aumenta de manera proporcional al tamaño del problema (n), ya que el número de operaciones que deben realizarse crece linealmente con n.

Inciso f.

```
    double start, end;
    double sum = 0.0;
    int n, thread_count;

    if (argc != 3) {
        printf("Usage: %s <n> <thread_count>\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]);
    thread_count = atoi(argv[2]);

    start = omp_get_wtime();

    double factor;

    // private() clausule
    #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);
    }

    end = omp_get_wtime();

    double pi_approx = 4.0 * sum;
```

```
root@d78ceaaf3fb5:/usr/src/app# gcc -o piSeriesNaive3 piSeriesNaive3.c -fopenmp
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 10000000 2
PI approximation 3.1415925536
- Threads: 2
- N value: 10000000
Execution time: 0.011527 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 100000000 2
PI approximation 3.1415926436
- Threads: 2
- N value: 100000000
Execution time: 0.117115 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 100000000 4
PI approximation 3.1415926436
- Threads: 4
- N value: 100000000
Execution time: 0.078080 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 100000000 6
PI approximation 3.1415926436
- Threads: 6
- N value: 100000000
Execution time: 0.066011 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 1000000000 6
PI approximation 3.1415926526
- Threads: 6
- N value: 1000000000
Execution time: 0.570459 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 1000000000 8
PI approximation 3.1415926526
- Threads: 8
- N value: 1000000000
Execution time: 0.452852 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesNaive3 10000000000 8
PI approximation 3.1415926531
- Threads: 8
- N value: 1410065408
Execution time: 0.638236 s
root@d78ceaaf3fb5:/usr/src/app#
```

Los valores de π aproximados en las ejecuciones son 3.1415926536, 3.1415926531, y cercanos a estos, lo cual es extremadamente preciso y se acerca mucho al valor real de pi. Esta precisión se mantiene constante a través de diferentes números de hilos (threads = 2, 4, 6, 8) y diferentes tamaños de n. No hay desviaciones notables en la precisión de la aproximación de pi, lo que indica que la paralelización con la cláusula private está funcionando correctamente.

Inciso g.

Medidas generales

Tiempo Secuencial (threads = 1 n = 10e8)	Tiempo paralelo (threads = 12 n = 10e8)	Tiempo paralelo (threads = 24 n = 10e8)	Tiempo paralelo (threads = 12 n = 10e9)
2.205135	0.458560	0.467975	0.561351
2.237946	0.425888	0.454874	0.547877
2.259760	0.422997	0.478706	0.574972
2.273707	0.439279	0.455099	0.568442
2.251157	0.421805	0.464534	0.593359

(threads = 12, n = 1e8)

Tiempo Secuencial (threads = 1, n = 1e8)	Tiempo paralelo	Speedup	Eficiencia
2.205135	0.45856	4.808825454	0.40073545
2.237946	0.425888	5.254775904	0.43789799
2.25976	0.422997	5.342260111	0.44518834
2.273707	0.439279	5.175997487	0.43133312
2.251157	0.421805	5.336961392	0.44474678

(threads = 24, n = 1e8)

Tiempo Secuencial (threads = 1, n = 1e8)	Tiempo paralelo	Speedup	Eficiencia
2.205135	0.467975	4.712078637	0.19633661
2.237946	0.454874	4.919925078	0.20499688
2.25976	0.478706	4.720559174	0.19668997
2.273707	0.455099	4.996071185	0.20816963
2.251157	0.464534	4.846054325	0.20191893

(threads = 12, n = 1e9)

Tiempo Secuencial (threads = 1, n = 1e8)	Tiempo paralelo	Speedup	Eficiencia
2.205135	0.561351	3.92826413	0.32735534
2.237946	0.547877	4.0847599	0.34039666
2.25976	0.574972	3.93020878	0.3275174
2.273707	0.568442	3.99989269	0.33332439
2.251157	0.593359	3.79392071	0.31616006

Escalabilidad fuerte (24/12 threads, n = 1e8)	Escalabilidad débil (1e9/1e8, 12 threads)
0.979881404	0.816886404
0.936276859	0.77734236
0.883625858	0.735682781
0.965238333	0.77277717
0.908017497	0.710876552

Inciso h.

```
    double start, end;
    double sum = 0.0;
    int n, thread_count;

    if (argc != 4) {
        printf("Usage: %s <n> <thread_count> <block_size>\n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]);
    thread_count = atoi(argv[2]);
    int block_size = atoi(argv[3]);

    start = omp_get_wtime();

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

    end = omp_get_wtime();

    double pi_approx = 4.0 * sum;
```

Static Scheduling			
Block size	Tiempo secuencial	Tiempo paralelo (threads = 12)	Speedup
16	0.023168	0.01101038 0.01100403 0.01101189 0.01102064 0.01100307 promedio = 0.011010	2.104268
64	0.023355	0.01138529 0.01140342 0.01139530 0.01138293 0.01139305 promedio = 0.011392	2.050122
128	0.023208	0.01185902 0.01185899 0.01186607 0.01184452 0.01184640 promedio = 0.011855	1.957654

Dynamic Scheduling			
Block size	Tiempo secuencial	Tiempo paralelo (threads = 12)	Speedup
16	0.028963	0.01706754 0.01706303 0.01707630 0.01706408 0.01705904 promedio = 0.017066	1.697117
64	0.023623	0.01124198 0.01122507 0.01122800 0.01121308 0.01122188 promedio = 0.011226	2.104311

128	0.023439	0.01074522 0.01073260 0.01074787 0.01073811 0.01074120 promedio = 0.010741	2.182199

Guided Scheduling			
Block size	Tiempo secuencial	Tiempo paralelo (threads = 12)	Speedup
16	0.024545	0.01076498 0.01078952 0.01077086 0.01076042 0.01077922 promedio = 0.010773	2.278381
64	0.032511	0.01069400 0.01070829 0.01068661 0.01069292 0.01070817 promedio = 0.010698	3.038979
128	0.023053	0.01270336 0.01269768 0.01269481 0.01269296 0.01268119 promedio = 0.012694	1.816055

Auto Scheduling		
Tiempo secuencial	Tiempo paralelo (threads = 12)	Speedup
0.027690	0.01189389 0.01189648 0.01191166 0.01190452	2.327282

	0.01188346 promedio = 0.011898	

Promedios de speedup para cada política de planificación y el promedio general:

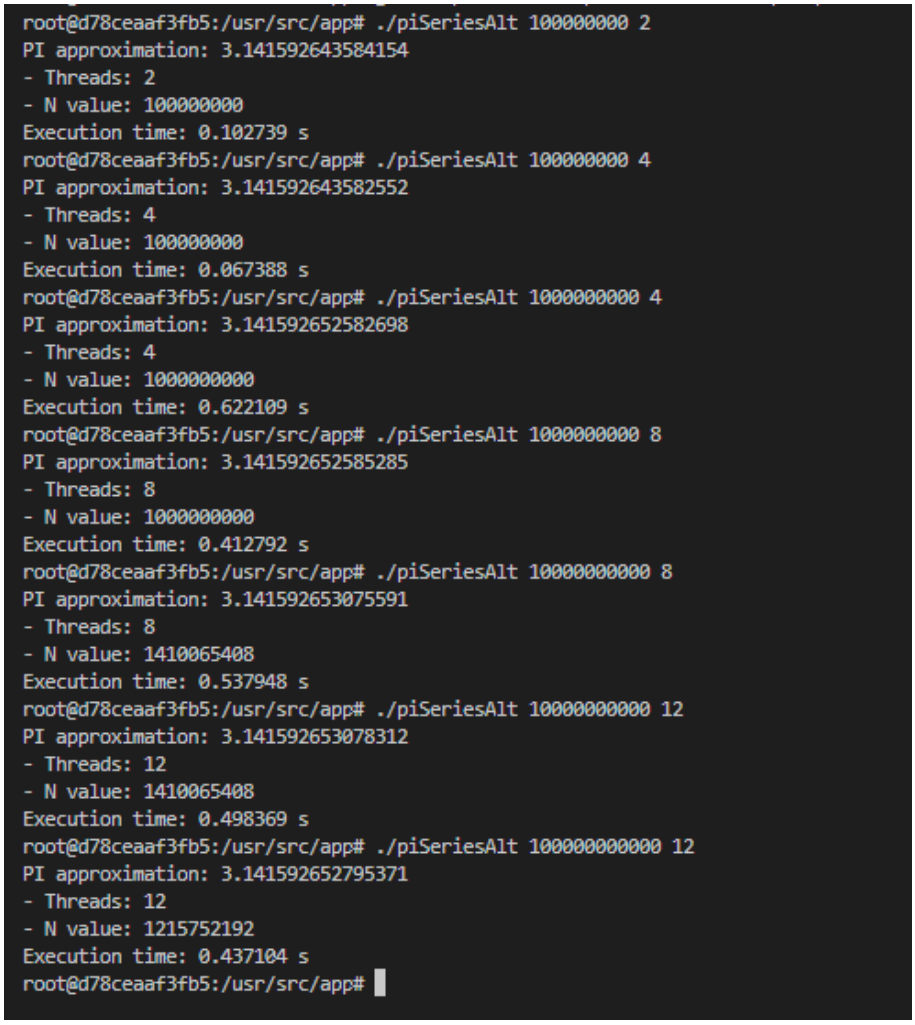
- Static Scheduling: 2.037348
- Dynamic Scheduling: 1.994542
- Guided Scheduling: 2.377805
- Auto Scheduling: 2.327282

Estos datos indican que, en promedio, la política de Guided Scheduling ofrece el mejor rendimiento, seguida de cerca por Auto Scheduling.

Ejercicio 2

Inciso a.

```
6     double start, end;
7     double sum_even = 0.0, sum_odd = 0.0;
8     int n, thread_count;
9
10    if (argc != 3) {
11        printf("Usage: %s <n> <thread_count>\n", argv[0]);
12        return 1;
13    }
14
15    n = atoi(argv[1]);
16    thread_count = atoi(argv[2]);
17
18    start = omp_get_wtime();
19
20    #pragma omp parallel for num_threads(thread_count) reduction(+:sum_even, sum_odd)
21    for (int i = 0; i < n; i++) {
22        if (i % 2 == 0) {
23            sum_even += 1.0 / (2 * i + 1);
24        } else {
25            sum_odd += 1.0 / (2 * i + 1);
26        }
27    }
28
29    end = omp_get_wtime();
30
31    double pi_approx = 4.0 * (sum_even - sum_odd);
```



La aproximación de pi obtenida en las ejecuciones con este código es 3.141592653, lo que es extremadamente cercano al valor real: 3.14159265358979323846. En el inciso h, el código se basa en la versión original que utilizaba un solo acumulador y alternaba el signo en cada iteración del bucle. La aproximación de pi obtenida en las ejecuciones es muy similar: 3.141592653, lo cual también es muy cercano al valor real. Ambos enfoques, tanto en el inciso g como en el inciso h, proporcionan una aproximación que es extremadamente precisa y cercana al valor real, con desviaciones mínimas que caen dentro de las limitaciones de la representación numérica en la computadora.

La ventaja del Inciso g es que al separar las sumas de índices pares e impares, se elimina la necesidad de calcular el factor alternante 1/-1 en cada iteración, lo cual puede ofrecer una ligera ventaja en eficiencia computacional al reducir el número de operaciones.

Inciso b.

```
root@d78ceaaf3fb5:/usr/src/app# gcc -o piSeriesAltOptimized piSeriesAlt.c -fopenmp -O2
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 100000000 2
PI approximation: 3.141592643584154
- Threads: 2
- N value: 100000000
Execution time: 0.067479 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 100000000 4
PI approximation: 3.141592643582552
- Threads: 4
- N value: 100000000
Execution time: 0.037030 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 1000000000 4
PI approximation: 3.141592652582698
- Threads: 4
- N value: 1000000000
Execution time: 0.331439 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 1000000000 8
PI approximation: 3.141592652585285
- Threads: 8
- N value: 1000000000
Execution time: 0.232139 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 10000000000 8
PI approximation: 3.141592653075588
- Threads: 8
- N value: 1410065408
Execution time: 0.338143 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 10000000000 12
PI approximation: 3.141592653078312
- Threads: 12
- N value: 1410065408
Execution time: 0.320137 s
root@d78ceaaf3fb5:/usr/src/app# ./piSeriesAltOptimized 100000000000 12
PI approximation: 3.141592652795374
- Threads: 12
- N value: 1215752192
Execution time: 0.266496 s
root@d78ceaaf3fb5:/usr/src/app#
```

Con optimización -O2 los tiempos de ejecución son consistentemente más bajos. Por ejemplo, con 2 hilos y n=10000000, el tiempo de ejecución es de aproximadamente 0.067 segundos. Sin optimización los tiempos de ejecución son mayores. Con los mismos 2 hilos y n=10000000, el tiempo de ejecución es de aproximadamente 0.102 segundos.

Ambos programas muestran una escalabilidad similar cuando se incrementa el número de hilos. Sin embargo, la versión optimizada tiende a ser más eficiente debido a la reducción de las instrucciones innecesarias y optimizaciones en la ejecución de bucles.

Los tiempos de ejecución de la versión optimizada son más consistentes. Esto se debe a las optimizaciones aplicadas que mejoran la predictibilidad del rendimiento.

La versión optimizada con la bandera -O2 claramente mejora el rendimiento en términos de tiempo de ejecución. Este es un resultado esperado, ya que la optimización de código por el compilador elimina instrucciones innecesarias y optimiza el flujo del programa para que se ejecute de manera más eficiente en términos de tiempo.