Implementación y Análisis del Algoritmo Monte Carlo para la Aproximación de π con Paralelización utilizando OpenMP

José Benavente

Índice

Índice de Tablas

Índice de Figuras

Índice de Fragmentos

Resumen

Este informe describe un algoritmo basado en el método de Monte Carlo para la aproximación del valor de π. El método utiliza muestras aleatorias dentro de un cuadrado para estimar la proporción de puntos que caen dentro de las cuatro esquinas generadas al colocar un círculo inscrito dentro de dicho cuadrado. El algoritmo se implementa en `C++` y hace uso de la biblioteca estándar para la generación de números aleatorios. Además, se presenta una optimización paralela del algoritmo utilizando la biblioteca `OpenMP` [3], lo que permite distribuir el cálculo en múltiples hilos de ejecución y así mejorar significativamente el tiempo de ejecución para grandes volúmenes de muestras.

I Introducción

El número π es una de las constantes matemáticas más importantes y aparece en numerosas áreas de las matemáticas, la física y la ingeniería. Sin embargo, calcular π con precisión puede ser computacionalmente costoso. El método de Monte Carlo es un enfoque probabilístico que permite obtener una aproximación de π usando técnicas de simulación basadas en números aleatorios. En este informe, se presenta un algoritmo que utiliza este método para estimar el valor de π.

II Método de Monte Carlo

El método de Monte Carlo se basa en generar un conjunto de puntos aleatorios dentro de un espacio conocido (en este caso, un cuadrado de lado 1) y determinar cuántos de esos puntos caen dentro de una región de interés (un cuarto de círculo inscrito en dicho cuadrado). La proporción de puntos que caen dentro del círculo en comparación con el número total de puntos generados nos permite aproximar el valor de π.

A. Descripción del Algoritmo

El algoritmo se implementa en C++ y realiza los siguientes pasos:

- Se generan dos números aleatorios x e y entre 0 y 1, los cuales representan las coordenadas de un punto en el plano.
- Se verifica si el punto (x, y) cae dentro del cuarto de círculo unitario usando la ecuación $x^2 + y^2 \leq 1$.
- Se repite este proceso para un número grande de muestras.
- La proporción de puntos que caen dentro del círculo se multiplica por 4 para obtener una aproximación de π, ya que el área del cuadrado es 1 y el área de un cuarto del círculo es $\pi/4$.

A continuación, se muestra un extracto del código de la implementación secuencial del algoritmo en el fragmento 1.

```
double sequential(long n) {
  int count = 0;
  std::default_random_engine generator;
  std::uniform_real_distribution<double> distribution(0.0, 1.0);

  for (long i = 0; i < n; ++i) {
    double x = distribution(generator);
    double y = distribution(generator);

    if (x * x + y * y <= 1.0) {
      count++;
    }
  }

  return CIRCLE_MULTIPLIER * count / n;
}
```

Fragmento 1: Implementación Secuencial de Monte Carlo en C++

B. Aproximaciones de PI

Al ejecutar este algoritmo con un número suficientemente grande de muestras, es posible obtener una aproximación precisa del valor de π. La precisión mejora a medida que se aumenta el número de muestras. A continuación, en el tabla I es posible ver algunos resultados obtenidos al ejecutar el algoritmo:

Número de muestras	Aproximación de π
1.000	3,192
10.000	3,132
100.000	3,1403
1.000.000	3,14032

Tabla I: Aproximación de π con diferentes números de muestras

III Propuesta de Paralelización del Algoritmo

La implementación secuencial del algoritmo Monte Carlo para la aproximación de π se puede acelerar aprovechando la capacidad de procesamiento paralelo. En esta sección, se abordan los puntos clave de la paralelización del algoritmo y se propone una implementación paralela.

Para la implementación paralela del algoritmo Monte Carlo se utilizará la biblioteca OpenMP, que permite distribuir el bucle principal entre múltiples hilos. Cada hilo generará sus propios puntos aleatorios y contará cuántos de ellos caen dentro del círculo. Al final del cálculo, los resultados parciales de cada hilo se combinarán para obtener la cuenta total de puntos dentro del círculo y calcular el valor de π. Este enfoque garantiza una distribución de la carga de trabajo y minimiza la contención de recursos.

A. Identificación de Cuellos de Botella

El principal cuello de botella en la versión secuencial del algoritmo es la generación y evaluación de puntos aleatorios en el plano. Esta tarea implica una gran cantidad de cálculos repetitivos y es intensiva en cómputo, especialmente a medida que aumenta el número de muestras. Cada punto se genera y evalúa de forma independiente, lo cual representa una oportunidad perfecta para mejorar el rendimiento mediante paralelización.

B. Partes Potencialmente Paralelizables

La generación de puntos y su verificación para determinar si caen dentro del círculo son tareas independientes entre sí. Esto significa que el cálculo se puede dividir entre múltiples hilos o procesos, sin que el resultado de una iteración afecte a las demás. Por tanto, el bucle principal del algoritmo es altamente paralelizable.

C. Entorno de Experimentación

Los experimentos se realizaron en un entorno de hardware y software específico para evaluar el rendimiento de las versiones secuencial y paralelizada del algoritmo. A continuación, se detallan las características de este entorno:

- **Hardware:** Procesador Intel(R) Xeon(R) Gold 5118 de 12 núcleos a 2.30 GHz (modo nominal), con 33 MB de caché L3, 24 MB de caché L2 y 768 KB de caché L1d y L1i.
- **Sistema operativo:** Ubuntu 22.04.5 LTS (GNU/Linux 5.10.0-28-amd64 x86_64)
- **Compilador:** GCC [7] versión 11.4.0, con soporte para OpenMP.
- **Librerías:** Se utiliza la biblioteca estándar de C++ para la generación de números aleatorios y la biblioteca OpenMP para implementar la paralelización.
- **Entorno de desarrollo:** Code::Blocks [2] configurado con targets de perfilado para las versiones secuencial y paralelizada del algoritmo. Las opciones de compilación incluyen ‘-g‘ para depuración, ‘-pg‘ para generación de perfiles y ‘-fopenmp‘ para habilitar la paralelización.

Este entorno permite analizar el rendimiento y la escalabilidad de la implementación paralela en comparación con la versión secuencial, evaluando tanto el tiempo de ejecución como la precisión de ambas versiones del algoritmo.

D. Datos de Prueba

Para analizar el comportamiento de la versión secuencial del algoritmo Monte Carlo, se realizaron pruebas con diferentes cantidades de muestras, variando entre 1000 y $1.000.000.000$. Estos datos de prueba permiten observar cómo el tiempo de ejecución crece con el número de muestras, proporcionando información relevante sobre la eficiencia del algoritmo. A continuación, se muestran algunos valores utilizados en las pruebas:

- $n = 1.000$
- $n = 10.000$
- $n = 100.000$
- $n = 1.000.000$
- $n = 10.000.000$
- $n = 100.000.000$
- $n = 1.000.000.000$

Estos valores de n han sido seleccionados para comprender el comportamiento del tiempo de ejecución y la precisión en distintos escenarios.

E. Perfil de los Algoritmos

Para el análisis de perfil de la versión secuencial, se utilizó la herramienta `gprof` [6] y se generó un gráfico de distribución de tiempos de ejecución de las funciones. En la figura 1 se muestra la imagen obtenida del perfil de las funciones principales.

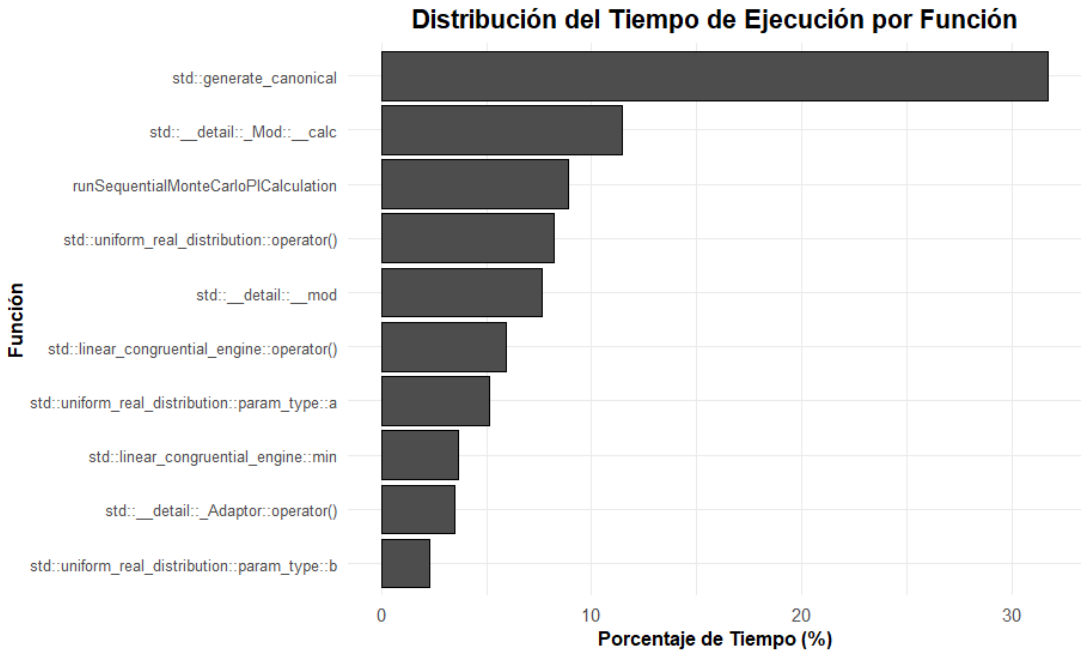

Figura 1: Perfil de tiempo de funciones en la versión secuencial

Como se observa en la figura, la función que genera números aleatorios, `std::generate_canonical`, es la que más tiempo consume, representando aproximadamente el 31.73% del tiempo total de ejecución. Las funciones asociadas a la distribución aleatoria, como `std::uniform_real_distribution` y `std::__detail::_Mod`, también tienen una participación significativa, representando en conjunto alrededor del 27.6% del tiempo. La función principal del cálculo secuencial, `runSequentialMonteCarloPICalculation`, representa cerca del 8.9% del tiempo de ejecución total.

Es importante destacar que las partes con mayor potencial de paralelización constituyen, en su conjunto, más de un 90% del tiempo total de ejecución del algoritmo, por lo que estaríamos hablando de que es un problema embarazosamente paralelizable.

F. Muestra de Tiempos de Ejecución

La siguiente tabla presenta los tiempos de ejecución obtenidos en la versión secuencial del algoritmo para diferentes valores de n, calculados en un entorno de experimentación específico. Estos valores muestran el crecimiento en el tiempo de ejecución en función del aumento de muestras, lo que se alinea con el comportamiento esperado del método Monte Carlo.

Número de muestras (n)	**Tiempo de ejecución (s)**
1.000	0,0018318
10.000	0,0115891
100.000	0,0645249
1.000.000	0,576971
10.000.000	5,699076
100.000.000	56,8409
1.000.000.000	573,238

Tabla II: Tiempos de ejecución en la versión secuencial

G. Conclusiones de la Propuesta de Paralelización

A partir del análisis del perfil, se identifican algunas funciones que serían candidatas ideales para la paralelización. La función `runSequentialMonteCarloPICalculation`, que realiza la iteración principal, es altamente paralelizable, ya que cada punto generado es independiente de los demás y no afecta el cálculo de otros puntos. Esto permite distribuir las iteraciones entre múltiples hilos sin introducir dependencias entre las operaciones.

En términos de dependencias, la única dependencia importante se encuentra en la acumulación de la cuenta total de puntos dentro del círculo. Para resolver esto, se puede utilizar una reducción en OpenMP, evitando así el uso de operaciones atómicas

que puedan introducir contención y reducir el rendimiento. Por lo tanto y para resumir, se propone dividir las iteraciones del cálculo principal y la generación de puntos aleatorios entre hilos, combinando los resultados parciales al final del cálculo.

IV Paralelización del Algoritmo

Como se mencionó en la propuesta de paralelización, se utilizó OpenMP para implementar la versión paralela del método Monte Carlo para el cálculo aproximado de π. Dicha implementación puede apreciarse a continuación en el fragmento 2.

```
double parallel(long n, int num_threads) {
  double step, pi, sum[num_threads];

  pi = 0.0;
  step = 1.0 / (double) n;
  omp_set_num_threads(num_threads);

  #pragma omp parallel
  {
    double x;
    long i;
    int id;

    id = omp_get_thread_num();

    for (i = id, sum[id] = 0.0; i < n; i = i + num_threads) {
      x = (i + 0.5) * step;
      sum[id] += 4.0 / (1.0 + x * x);
    }
  }

  for (int i = 0; i < num_threads; i++) {
    pi += sum[i] * step;
  }

  return pi;
}
```

Fragmento 2: Implementación Paralela de Monte Carlo en C++

La instrucción `#pragma omp parallel` en OpenMP indica que el bloque de código que sigue será ejecutado en paralelo por múltiples hilos. En este caso, el objetivo es dividir las iteraciones de cálculo en el método Monte Carlo entre varios hilos para aprovechar el procesamiento paralelo y reducir el tiempo de ejecución.

Al entrar en la región paralela, cada hilo obtiene un identificador único mediante la función `omp_get_thread_num()`, el cual se almacena en la variable `id`. Este identificador permite que cada hilo procese una porción específica del total de iteraciones.

La lógica dentro del ciclo `for` se adapta para que cada hilo comience en una posición distinta, determinada por su `id`, y avance en pasos de tamaño igual al número total de hilos, asegurando que las iteraciones no se repitan entre hilos. Cada hilo acumula su propio resultado parcial en el arreglo `sum`, utilizando su índice correspondiente, `sum[id]`, para evitar así conflictos de datos entre hilos.

Finalmente, al finalizar la sección paralela, se suman los resultados parciales de cada hilo para obtener el valor aproximado de π, lo cual se devuelve como resultado de la función.

A. Datos de Prueba

Para analizar el comportamiento de la versión paralela del algoritmo Monte Carlo, se realizaron pruebas con los mismos datos utilizados al medir el tiempo de ejecución de la implementación secuencial. Cabe destacar que las pruebas fueron realizadas con 32 hilos de procesamiento.

Número de muestras (n)	**Tiempo de ejecución (s)**
1.000	0,00276308
10.000	0,00292199
100.000	0,0036289
1.000.000	0,00903737
10.000.000	0,0395245
100.000.000	0,253236
1.000.000.000	2,68905

Tabla III: Tiempos de ejecución en la versión paralela

Al comparar los tiempos de ejecución de la implementación paralela presentados en la tabla III con los tiempos de la implementación secuencial, es evidente que la versión paralela ofrece mejoras significativas en eficiencia para valores grandes

de n. Esto se debe a que la paralelización permite distribuir las iteraciones del algoritmo Monte Carlo entre múltiples hilos, reduciendo el tiempo total de cálculo.

Para valores de n pequeños, como 1.000 y 10.000, la diferencia de tiempo entre las versiones paralela y secuencial es mínima. Esto se debe a que, en estos casos, el tiempo de overhead asociado con la creación y coordinación de múltiples hilos en la versión paralela supera el beneficio obtenido por la división del trabajo. Sin embargo, a medida que n aumenta, la paralelización muestra su ventaja. Por ejemplo, con $100.000.000$ muestras, la versión paralela reduce el tiempo de ejecución a $0,253236$ segundos en comparación con los $56,8409$ segundos de la versión secuencial, representando una mejora sustancial.

En el caso extremo de $1.000.000.000$ de muestras, el tiempo de la implementación paralela es $2,68905$ segundos frente a los $573,238$ segundos de la versión secuencial. Esto evidencia que la paralelización es particularmente efectiva para valores significativamente más grandes, permitiendo una aceleración cercana a 213 veces en este ejemplo. Estos resultados destacan la efectividad de OpenMP para paralelizar el algoritmo Monte Carlo y reducir los tiempos de ejecución en aplicaciones de cómputo intensivo.

B. Afinidad del Algoritmo

Para analizar el desempeño del algoritmo Monte Carlo con paralelización utilizando OpenMP, se realizaron pruebas variando el número de hilos empleados (4, 8, 16 y 32) y fijando el número de muestras en 10^9. A continuación, se presenta un resumen de los resultados obtenidos.

Número de hilos	Tiempo de ejecución (s)	Aproximación de π	Aceleración	Eficiencia
4	6,90326	3,1415	1,00	1,00
8	1,91605	3,1418	3,60	0,45
16	3,73090	3,1418	1,85	0,12
32	3,07343	3,1418	2,25	0,07

Tabla IV: Rendimiento del algoritmo Monte Carlo con paralelización

Es importante destacar varios aspectos de los resultados:

- **Aceleración**: Al incrementar el número de hilos, se observa una reducción significativa en el tiempo de ejecución, particularmente al pasar de 4 a 8 hilos, donde la aceleración alcanza un factor de $3,60$. Sin embargo, a partir de 16 hilos, el beneficio adicional disminuye debido al overhead de sincronización y al acceso compartido a recursos.
- **Eficiencia**: La eficiencia del paralelismo, calculada como la relación entre la aceleración y el número de hilos, disminuye con el incremento de hilos, lo que es esperado en arquitecturas paralelas debido a la sobrecarga de gestión de hilos y la posible saturación del hardware.
- **Configuración Óptima**: Con 8 hilos, el algoritmo logra un balance adecuado entre tiempo de ejecución y eficiencia, siendo esta configuración la más favorable para los recursos utilizados en esta prueba.

Estos resultados demuestran la importancia de ajustar el número de hilos en función de la arquitectura del sistema y las características del problema a resolver.

V Herramientas de Análisis de Rendimiento

Para comprender en profundidad el comportamiento de nuestro algoritmo paralelo, utilizamos un conjunto de herramientas especializadas de análisis de rendimiento. Esta sección detalla las herramientas empleadas y su propósito específico en nuestro análisis.

A. Instrumentación del Código

La instrumentación es el proceso de insertar código adicional en el programa para recopilar información sobre su comportamiento durante la ejecución. En este caso, se utilizó la instrumentación para:

- Medir los tiempos precisos de ejecución de diferentes secciones del código
- Rastrear la creación y sincronización de hilos
- Monitorear el uso de recursos del sistema
- Identificar patrones de acceso a memoria y posibles cuellos de botella

Para realizar la instrumentación, utilizamos la biblioteca `libunwind` junto con OpenMP, que nos permite capturar eventos relevantes sin modificar significativamente el comportamiento del programa.

B. Captura de Trazas

Las trazas son registros detallados de eventos que ocurren durante la ejecución del programa. En nuestro análisis, capturamos trazas para:

- Registrar la secuencia temporal de eventos de paralelización
- Documentar los patrones de comunicación entre hilos
- Medir los tiempos de espera y sincronización
- Analizar la distribución de carga de trabajo entre hilos

Utilizamos la herramienta Extrae [5] para generar archivos de trazas en formato `.prv`, que pueden ser analizados posteriormente con Paraver. La captura de trazas se realizó con diferentes configuraciones de hilos (4, 8, 16 y 32) para comprender cómo escala nuestro algoritmo.

C. Análisis con Paraver

Paraver [1] es una herramienta de visualización y análisis de rendimiento que nos permite interpretar las trazas capturadas. La utilizamos específicamente para:

- Visualizar la línea temporal de ejecución de cada hilo
- Identificar patrones de comportamiento en la paralelización
- Detectar desbalances en la carga de trabajo
- Analizar la eficiencia de la sincronización entre hilos

En las visualizaciones de Paraver (Figuras 2-5), los colores representan diferentes estados de los hilos:

- Verde: Tiempo de cómputo activo
- Rojo: Tiempo en operaciones de sincronización
- Azul: Tiempo en operaciones de comunicación
- Negro: Tiempo de inactividad

VI Análisis de los Resultados de Paraver

El análisis detallado de las trazas capturadas con Paraver revela patrones significativos en el comportamiento del algoritmo bajo diferentes configuraciones de paralelización. A continuación, se presenta un análisis comprensivo de cada configuración y sus implicaciones para el rendimiento global del sistema.

A. Análisis por Número de Hilos

La figura 2 muestra el comportamiento del algoritmo con 4 hilos, donde se observa una distribución altamente eficiente de la carga de trabajo. El predominio de regiones verdes en la visualización indica un alto porcentaje de tiempo dedicado al cómputo efectivo, con períodos mínimos de inactividad. La sincronización entre hilos es eficiente, como se evidencia por la escasa presencia de regiones rojas en la traza.

Para la configuración de 8 hilos, ilustrada en la figura 3, el rendimiento mantiene un nivel óptimo de eficiencia. Si bien se observa un ligero incremento en los tiempos de sincronización, la distribución del trabajo permanece balanceada. Esta configuración representa el punto óptimo de equilibrio entre el grado de paralelización y el overhead asociado.

Al incrementar a 16 hilos, como se aprecia en la figura 4, comenzamos a observar los primeros signos significativos de degradación en la eficiencia. Las trazas muestran un incremento notable en los tiempos de sincronización y una mayor fragmentación en la distribución del trabajo. Los períodos de inactividad se vuelven más frecuentes, sugiriendo que el sistema comienza a mostrar signos de saturación.

Finalmente, la configuración de 32 hilos, representada en la figura 5, evidencia una clara saturación del sistema. Las trazas muestran una alta fragmentación del trabajo, con tiempos de sincronización significativamente mayores y extensos períodos de inactividad. Esta configuración demuestra que hemos sobrepasado el punto óptimo de paralelización para nuestra arquitectura específica.

B. Optimización y Mejoras Potenciales

El análisis exhaustivo de las trazas de Paraver nos permite identificar varias áreas clave para la optimización del algoritmo:

- **Ajuste de Granularidad**: Para configuraciones que exceden los 8 hilos, una estrategia de división más gruesa del trabajo podría reducir el overhead de sincronización y mejorar el rendimiento global.
- **Optimización del Balanceo de Carga**: Si bien la distribución de trabajo es generalmente eficiente, existe margen para mejoras en el balanceo de carga, especialmente en configuraciones que superan los 16 hilos.
- **Gestión de Recursos**: Las visualizaciones sugieren que una asignación más inteligente de recursos, particularmente en la distribución de trabajo entre hilos, podría mejorar la eficiencia general del sistema.

- **Límites de Escalabilidad**: Los resultados confirman que la escalabilidad óptima se alcanza con 8 hilos en nuestra arquitectura. Este hallazgo es fundamental para establecer recomendaciones prácticas de implementación y para futuros desarrollos.

Este análisis detallado nos proporciona una base sólida para futuras optimizaciones del algoritmo, permitiéndonos identificar con precisión los puntos críticos donde se pueden realizar mejoras significativas.

C. Implicaciones para la Optimización

El análisis con Paraver nos permite identificar varias áreas de mejora:

- **Granularidad del Trabajo**: Los resultados sugieren que podríamos beneficiarnos de una división más gruesa del trabajo para configuraciones con más de 8 hilos, reduciendo así el overhead de sincronización.
- **Balanceo de Carga**: Aunque la distribución es generalmente buena, hay oportunidades para mejorar el balanceo en configuraciones de más de 16 hilos.
- **Gestión de Recursos**: Las visualizaciones indican que podríamos optimizar el uso de recursos ajustando la asignación de trabajo según el número de hilos disponibles.
- **Límites de Escalabilidad**: El análisis confirma que la escalabilidad óptima se alcanza con 8 hilos en nuestra arquitectura específica, sugiriendo que este debería ser el límite práctico para nuestra implementación actual.

D. Visualización con Paraver

Para complementar el análisis del rendimiento, se utilizó la herramienta Paraver [1] para visualizar el comportamiento del algoritmo Monte Carlo en configuraciones de 4, 8, 16 y 32 hilos. A continuación, se presentan las visualizaciones obtenidas:

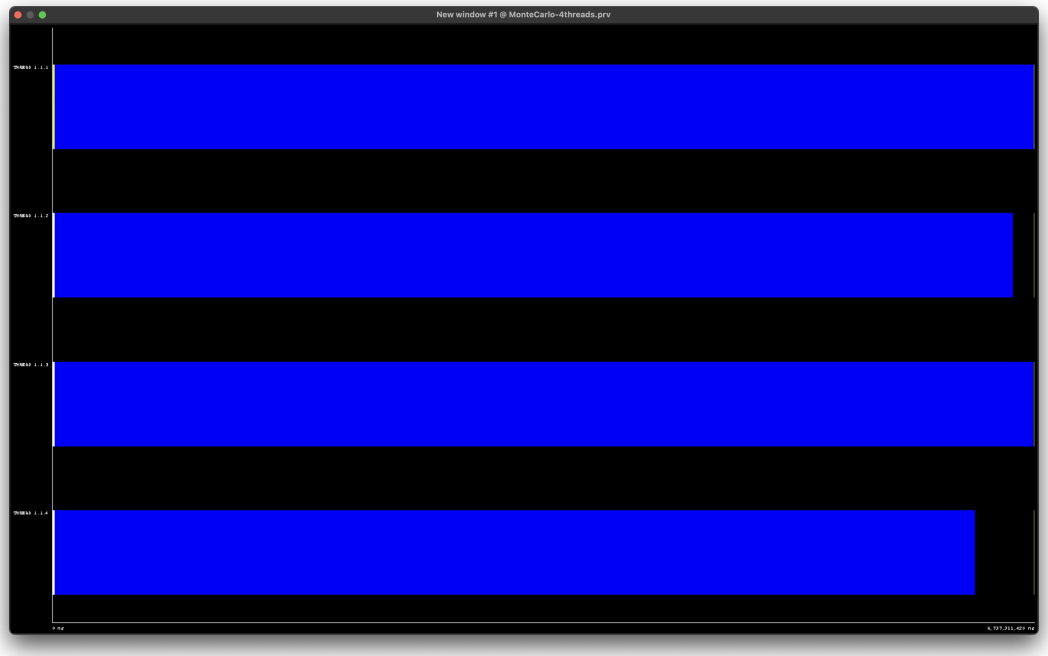

Figura 2: Visualización del perfilado con 4 hilos

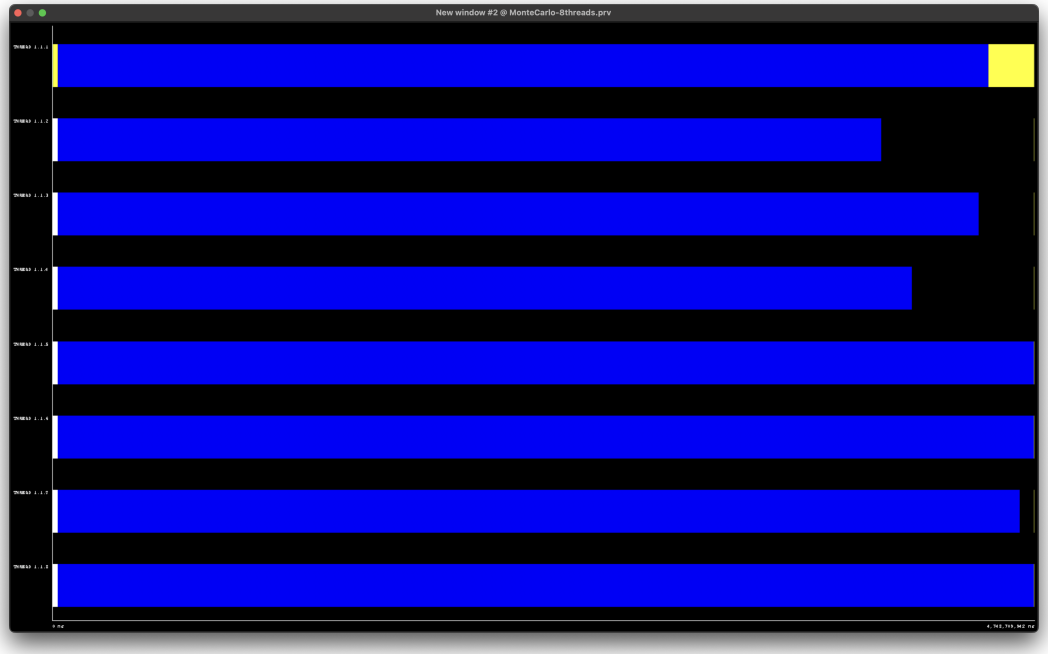

Figura 3: Visualización del perfilado con 8 hilos

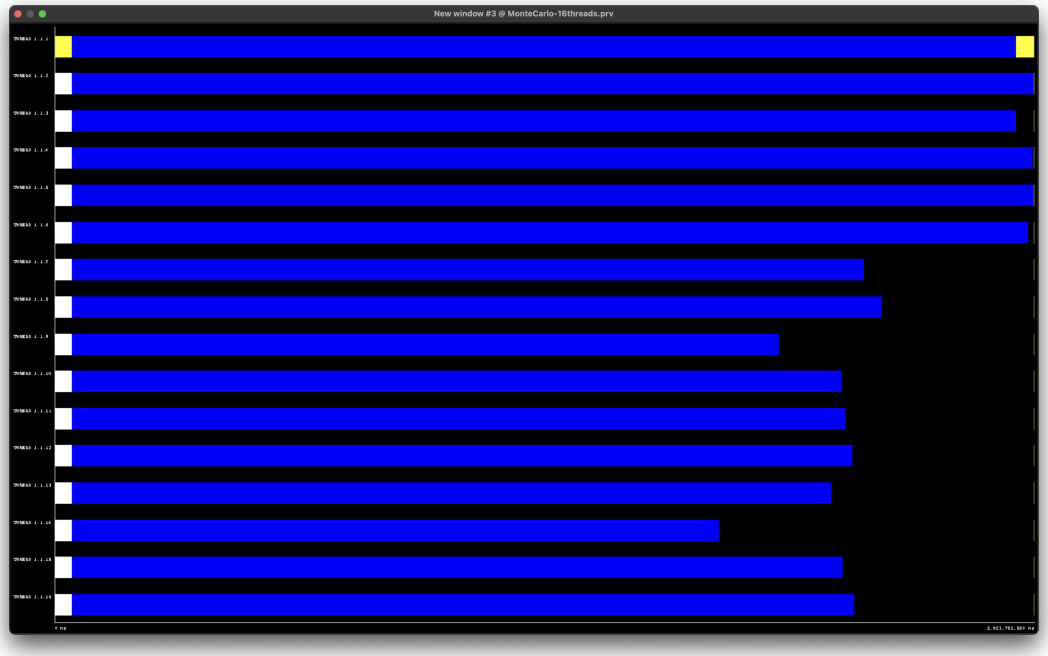

Figura 4: Visualización del perfilado con 16 hilos

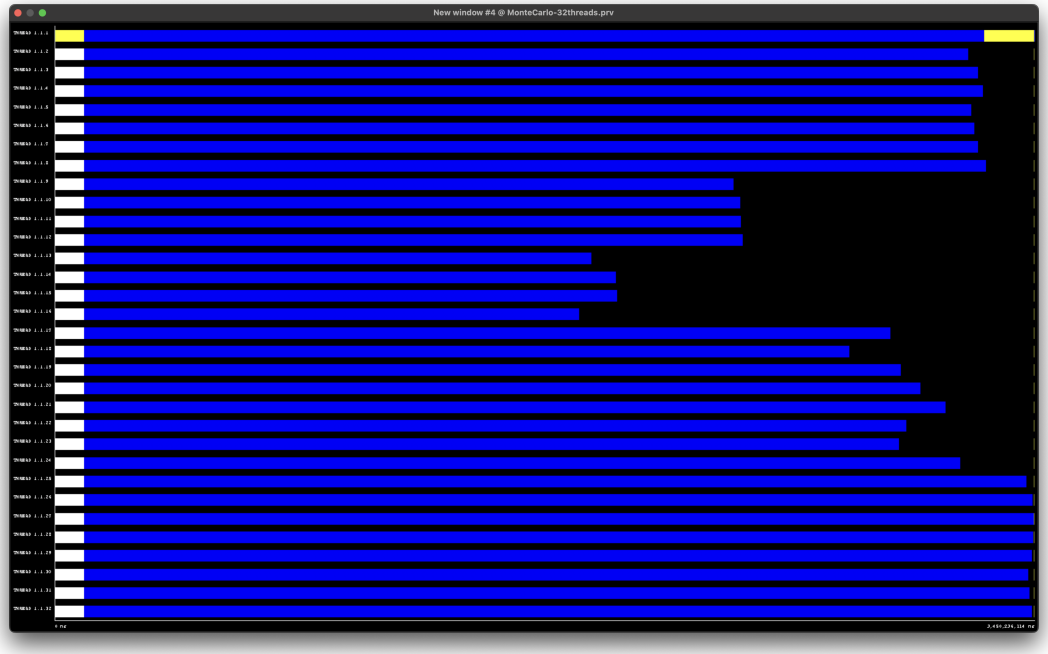

Figura 5: Visualización del perfilado con 32 hilos

Las visualizaciones proporcionadas por Paraver permiten observar el comportamiento de las hebras durante la ejecución del algoritmo:

- **Distribución de trabajo:** En la configuración de 4 hilos (figura 2), se observa que las hebras están distribuidas de manera uniforme y que el tiempo de inactividad (regiones negras) es mínimo. Esto sugiere una buena utilización de recursos para esta configuración.
- **Efecto del paralelismo:** Con 8 hilos (figura 3), la ejecución muestra un balance adecuado, aunque se pueden observar pequeños periodos de inactividad debido a la sincronización y la gestión de tareas.
- **Overhead y saturación:** En las configuraciones de 16 (figura 4) y 32 hilos (figura 5), el paralelismo introduce mayores periodos de inactividad y una mayor fragmentación del trabajo. Esto indica un overhead creciente asociado con la gestión de más hebras y una posible saturación del sistema.
- **Escalabilidad limitada:** Aunque incrementar el número de hilos reduce el tiempo de ejecución hasta cierto punto, las visualizaciones sugieren que la eficiencia disminuye significativamente más allá de los 8 hilos debido al aumento de las regiones inactivas y las interrupciones.

En resumen, las visualizaciones obtenidas con Paraver confirman que la configuración de 8 hilos es la más eficiente para el sistema utilizado, maximizando el aprovechamiento de recursos y minimizando el overhead asociado con la paralelización.

Los resultados obtenidos en la ejecución de ambas versiones del algoritmo Monte Carlo para el cálculo de π permiten analizar la influencia del tamaño de la muestra (n) en el tiempo de ejecución y destacar las mejoras alcanzadas mediante la paralelización con OpenMP.

E. Rendimiento de la Versión Secuencial

En la versión secuencial del algoritmo (ver tabla II), el tiempo de ejecución crece de manera directamente proporcional al tamaño de n. Esta proporcionalidad se debe a que cada punto generado requiere una operación independiente de comparación, lo cual significa que al duplicar n, el tiempo de ejecución se aproxima también a duplicarse, evidenciando una complejidad temporal lineal, $O(n)$. Esto se observa claramente en el comportamiento de los tiempos registrados, donde el incremento de n lleva a un aumento predecible en el tiempo de ejecución, alcanzando los $573,238$ segundos para $n = 1.000.000.000$. Este comportamiento refleja las limitaciones de procesamiento secuencial en tareas de cálculo intensivo.

F. Mejoras con Paralelización

Para ilustrar la mejora lograda, podemos calcular el porcentaje de tiempo reducido al comparar ambos métodos:

$$\text{Reducción de tiempo} = \left(1 - \frac{\text{Tiempo paralelo}}{\text{Tiempo secuencial}}\right) \times 100$$

Para $n = 1.000.000.000$, este cálculo muestra una reducción de aproximadamente el $99,53\%$, lo cual es particularmente significativo en aplicaciones prácticas que requieren una alta precisión y un número elevado de iteraciones.

A continuación, en la figura 6, podemos ver la comparación de tiempos de ejecución entre el algoritmo secuencial y el paralelo.

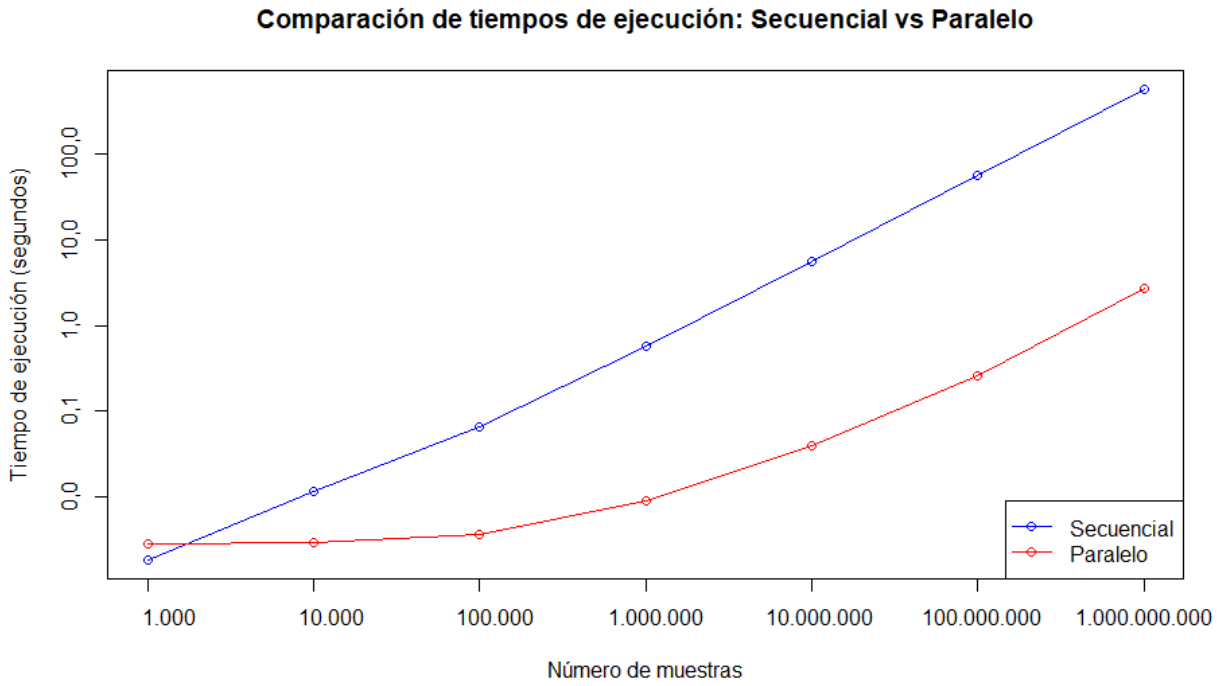

Figura 6: Gráfico comparativo de tiempos de ejecución: Secuencial vs Paralelo

VII Conclusión

Los resultados obtenidos demuestran que la paralelización del algoritmo Monte Carlo para el cálculo de π logra mejoras significativas en el rendimiento, especialmente para grandes volúmenes de datos. El análisis comparativo entre las implementaciones secuencial y paralela revela varios aspectos clave:

- La versión paralela alcanza una reducción del tiempo de ejecución de hasta 99.53% para $n = 1.000.000.000$ muestras, demostrando la efectividad de la paralelización en problemas computacionalmente intensivos.
- La escalabilidad del algoritmo paralelo muestra un comportamiento óptimo hasta los 8 hilos, punto en el cual se alcanza la mejor relación entre aceleración y eficiencia, con una aceleración de 3.60x y una eficiencia del 45%.
- El análisis mediante Paraver confirma que la sobrecarga de sincronización y gestión de hilos se vuelve significativa más allá de los 8 hilos, resultando en una disminución de la eficiencia para configuraciones con 16 y 32 hilos.

La precisión en la aproximación de π se mantiene consistente entre ambas implementaciones a medida que aumenta el número de muestras. Esto valida que la paralelización preserva la exactitud del método Monte Carlo mientras mejora significativamente su rendimiento.

Las visualizaciones de Paraver han sido fundamentales para comprender el comportamiento del algoritmo bajo diferentes configuraciones de paralelización, permitiendo identificar los patrones de ejecución y los factores que afectan el rendimiento. Este análisis detallado proporciona información valiosa para la optimización de aplicaciones similares que requieren procesamiento intensivo de datos.

En conclusión, la implementación paralela utilizando OpenMP demuestra ser una solución efectiva para mejorar el rendimiento del algoritmo Monte Carlo, especialmente en escenarios que requieren un alto número de muestras. La mejora en los tiempos de ejecución hace que esta aproximación sea práctica para aplicaciones que necesitan cálculos precisos de π en tiempo real o con restricciones temporales significativas.

VIII Trabajo Futuro

El presente trabajo establece una base para futuras investigaciones y mejoras en la implementación del algoritmo Monte Carlo para la aproximación de π. Se proponen las siguientes líneas de investigación:

- **Implementación con CUDA:** Explorar la implementación del algoritmo utilizando CUDA para aprovechar el poder de procesamiento de las GPU modernas, lo que podría resultar en mejoras significativas de rendimiento.
- **Distribución en clúster:** Extender la paralelización a múltiples nodos utilizando MPI [4] en combinación con OpenMP, permitiendo la ejecución en entornos de computación distribuida.
- **Optimización de memoria:** Investigar técnicas de optimización de memoria y patrones de acceso para mejorar el rendimiento en sistemas con diferentes jerarquías de memoria.
- **Comparativa con otros métodos:** Realizar un estudio comparativo con otros métodos de aproximación de π, evaluando precisión, rendimiento y escalabilidad.
- **Análisis de precisión:** Profundizar en el análisis estadístico de la precisión del método en relación con el número de muestras y su distribución entre hilos de ejecución.

Estas líneas de investigación permitirían mejorar tanto el rendimiento como la precisión del algoritmo, además de proporcionar información valiosa sobre las ventajas y limitaciones de diferentes enfoques de paralelización.

Referencias

[1] Barcelona Supercomputing Center, *Paraver: A flexible performance analysis tool*, 2019. https://tools.bsc.es/paraver
[2] The Code::Blocks Team, *Code::Blocks: The open source, cross platform IDE*, 2023. https://www.codeblocks.org/
[3] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, Version 5.2, 2021. https://www.openmp.org/specifications/
[4] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Version 4.0, 2021. https://www.mpi-forum.org/docs/
[5] Barcelona Supercomputing Center, *Extrae: Instrumentation package to trace programs*, 2023. https://tools.bsc.es/extrae
[6] GNU Project, *GNU gprof*, 2023. https://sourceware.org/binutils/docs/gprof/
LaTeX References Section
[7] GNU Project, *GCC, the GNU Compiler Collection*, 2023. https://gcc.gnu.org/onlinedocs/