Universidad de Valladolid

Facultad de Ciencias

Trabajo Fin de Grado

Grado en Estadística

Estadística Computacional y Programación en Paralelo

Autora: **Susana Gutiérrez Martín**

Tutor: **Eusebio Arenal Gutiérrez**

¡Ay de los que luchan toda la vida!

Esos son los imprescindibles. *(Bertolt Brecht)*

Abandonar puede tener justificación,

abandonarse no la tiene jamás. *(Ralph W. Emerson)*

Con constancia y tenacidad se

obtiene lo que se desea,

la palabra imposible no tiene significado. *(Napoleón)*

Con orden y tiempo se encuentra el

secreto de hacerlo todo, y de hacerlo bien. *(Pitágoras)*

Agradecimientos

Quiero dar las gracias a todas las personas que han creído siempre en mí, a todos los que han estado en las buenas y en las malas, a todos los que me apoyan y motivan para haber llegado hasta aquí y seguir creciendo día a día.

Como bien decía mi abuela, que en paz descanse, Siempre en positivo (excepto en el Covid).

Índice general

Índice de figuras

Índice de tablas y gráficos

Resultados

Resumen

La estadística computacional y la programación en paralelo son áreas importantes en el campo de la ciencia de datos y la informática. Dichas materias están vinculadas, y su interrelación permite el análisis eficiente de grandes conjuntos de datos. La estadística computacional está estrechamente relacionada con la ciencia de datos y la programación en paralelo es un concepto importante en el campo de la informática.

La estadística computacional se refiere a la aplicación de métodos y técnicas estadísticas con el apoyo de algoritmos informáticos. Tradicionalmente, los métodos estadísticos se aplicaban manualmente, pero con el advenimiento de la computación, se han desarrollado algoritmos y herramientas que permiten automatizar estos procesos con objeto de su implementación informática.

En una gran variedad de campos en los que se usa la estadística computacional (como la investigación científica, el análisis de datos empresariales, la bioinformática, la genómica, el aprendizaje automático y la Inteligencia Artificial) hay que analizar grandes conjuntos de datos. La programación en paralelo permite aplicar muchos de los algoritmos utilizados en la estadística computacional de una manera más rápida y eficiente.

La programación en paralelo implica dividir una tarea en hilos más pequeños que se ejecutan simultáneamente en múltiples procesadores o núcleos de CPU. Las tarjetas gráficas son eficientes empleando esta técnica, ya que pueden ejecutar múltiples hilos en muchos núcleos al mismo tiempo. El objetivo es acelerar los cálculos dividiendo la tarea en hilos y ejecutándolos a la vez, a diferencia del enfoque de "divide y vencerás" de los algoritmos tradicionales que, aunque los divide, no los ejecuta simultáneamente, si no que los ejecuta uno a uno.

Abstract

Computational statistics and parallel programming are important areas in the field of data science and computer science. These materias are interconnected, combining statistics and computer science to analyze large datasets efficiently. Computational statistics is closely related to data science, and parallel programming is an important concept in the field of computer science.

Computational statistics refers to the application of statistical methods and techniques with the support of computer algorithms. Traditionally, statistical methods were applied manually, but with the advent of computing, algorithms and tools have been developed to automate these processes for computer implementation.

In a wide range of fields where computational statistics is used, such as scientific research, business data analysis, bioinformatics, genomics, machine learning, and artificial intelligence, large datasets need to be analyzed. Parallel programming enables the application of many algorithms used in computational statistics in a fast and efficient manner.

Parallel programming involves dividing a task into smaller threads that are executed simultaneously on multiple processors or CPU cores. Graphics processing units (GPUs) are efficient in this technique as they can run multiple threads on many cores at the same time. The goal is to accelerate calculations by dividing the task into threads and executing them simultaneously, unlike the "divide and conquer" approach of traditional algorithms.

1. INTRODUCCIÓN

1.1. Definiciones

Se necesita saber qué es la Estadística Computacional y qué es la Programación en Paralelo.

- Estadística Computacional: Se trata de un campo de estudio dentro del ámbito científico y tecnológico que se enfoca en examinar cómo la computación afecta a la metodología estadística. (i.e. Algoritmos, modelación gráfica, métodos inferenciales intensivos en cálculo recursivo, análisis exploratorio de datos, etc.), es decir, es el vínculo entre la estadística, el cálculo numérico y la informática.

  Se refiere a los métodos estadísticos que se habilitan mediante el uso de métodos computacionales.

- Programación en Paralelo (3,5,10): Consiste en aprovechar múltiples recursos computacionales de manera conjunta para abordar un problema en particular. A diferencia de la programación secuencial, en la programación paralela es posible llevar a cabo varias operaciones de manera simultánea.
  "Dividir un gran problema en varios pequeños y resolverlos al mismo tiempo".

1.2. Breve historia de la computación paralela

En 1837, el profesor británico Charles Babbage presentó al mundo su increíble invención, la máquina analítica. Este hito marcó el comienzo de una serie de teorías revolucionarias sobre la computación paralela. Cinco años después, en 1842, Babbage se encontró con el matemático italiano Luigi Menabrea (8) durante un viaje por Italia. Este encuentro inspirador llevó a la publicación del primer artículo en francés sobre la máquina analítica. Sin embargo, no fue hasta 1843 que la obra de Menabrea fue traducida al inglés por Ada Lovelace, quien agregó su propio toque especial al diseñar el primer algoritmo destinado a ser ejecutado en una máquina. Ada Lovelace es ahora reconocida como la pionera en la programación de computadoras debido a esta contribución crucial. Saltando hacia adelante en el tiempo, se llega a 1954, cuando la renombrada compañía IBM lanzó al mercado el revolucionario IBM 704. Este lanzamiento marcó un hito significativo en la historia de la informática, impulsando avances importantes en el campo. Desde aquellos primeros días hasta ahora, se ha recorrido un largo camino en el mundo de la tecnología, gracias a los brillantes pioneros y sus innovaciones.

En 1958, S. Gill Ferranti abrió la puerta a la discusión sobre el tema, destacando la importancia del "branching" y el "waiting" en la programación. Ese mismo año, dos investigadores de IBM, Cocke y Slotnick, mantuvieron una conversación histórica sobre el uso del paralelismo en los cálculos numéricos. Fue entonces cuando Slotnick presentó el emocionante proyecto SOLOMON, destinado a construir un supercomputador revolucionario. Aunque el proyecto no se materializó, sentó las bases para futuros desarrollos en esta área emocionante.

Avanzando hacia 1962, la empresa Burroughs Corporation desarrolló un sistema informático con 4 procesadores y 16 módulos de memoria. En ese mismo año, el computador ATLAS se destacó como la primera máquina en incorporar conceptos de memoria virtual y paginación. Luego, en 1964, la Fuerza Aérea de los Estados Unidos financió el revolucionario ILLIAC IV, considerado el primer supercomputador paralelo masivo de la historia, con impresionantes 256 procesadores. Slotnick fue el encargado de liderar este ambicioso proyecto que dejó una huella imborrable en la computación paralela.

En 1965, Dijkstra realizó importantes contribuciones al describir y acuñar la terminología relacionada con los tramos críticos. Ese mismo año, James Cooley y John Tukey desarrollaron el algoritmo de la transformada rápida de Fourier, que se convirtió en uno de los algoritmos más utilizados en operaciones de punto flotante. En 1967, Amdahl y Slotnick mantuvieron una discusión sobre la viabilidad del procesamiento paralelo, lo que llevó al establecimiento de la conocida ley de Amdahl, que establece las limitaciones del rendimiento en función de los componentes del sistema. *“La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.”*

En 1968, Edsger Dijkstra presentó los semáforos como una solución potencial para las secciones críticas, brindando una perspectiva detallada sobre su aplicación en este contexto. Durante el período de 1968 a 1976, se llevaron a cabo diversos proyectos en Estados Unidos, Rusia, Japón y Europa para impulsar la computación paralela, con una fuerte inversión tanto del sector tecnológico como de las instituciones académicas.

En 1976, se realizó la primera implementación en ILLIAC IV, que obtuvo el dudoso título de "el más infame de los supercomputadores" debido a que solo se completó en un 25% después de 11 años, superando cuatro veces el costo estimado. A pesar de las dificultades, el ILLIAC IV se convirtió en el ordenador más rápido de su época y sentó las bases para proyectos futuros.

En 1981, el proyecto ILLIAC IV fue desmantelado por la NASA. Aunque se considera que fue un fracaso desde el punto de vista económico, logró convertirse en el

ordenador más veloz de su tiempo. Además, varios de los conceptos fundamentales utilizados en la construcción del ILLIAC IV se implementaron con éxito en proyectos futuros.

En la década de 1980, surgieron supercomputadoras de una nueva generación gracias al proyecto "Concurrent Computation" liderado por Caltech. Este proyecto demostró que era posible obtener un rendimiento excepcional utilizando microprocesadores convencionales fácilmente disponibles en el mercado, abriendo nuevas posibilidades para la computación paralela. Hacia finales de los años 80, los clústeres se convirtieron en una alternativa competitiva a los MPP, utilizando computadoras estándar interconectadas mediante redes también estándar. Hoy en día, los clústeres son la arquitectura predominante en los centros de datos.

En la década de 1990, tanto los MPP como los clústeres adoptaron el estándar MPI (Interfaz de Paso de Mensajes) para facilitar la programación paralela. Además, los multiprocesadores con memoria compartida también experimentaron avances significativos con la aparición de pthreads y OpenMP. Estos avances representaron hitos importantes en el campo de la computación paralela.

En la actualidad, la computación paralela se ha vuelto común, ya que la mayoría de los dispositivos informáticos vienen equipados con procesadores de múltiples núcleos. El software ha desempeñado un papel fundamental en el desarrollo de la programación paralela, aunque presenta mayores desafíos en comparación con la programación secuencial debido a la necesidad de establecer una comunicación y sincronización efectivas entre las tareas paralelizadas. A medida que se avanza en la era digital, la computación paralela continúa evolucionando y desempeñando un papel crucial en la resolución de problemas complejos y el impulso de la innovación tecnológica.

1.3. Objetivos del trabajo

En el presente trabajo de fin de grado, se abordará el tema de la programación paralela y su aplicación en el ámbito del análisis estadístico. La computación paralela se ha convertido en una herramienta crucial para mejorar el rendimiento de los algoritmos y reducir los tiempos de ejecución en diversas áreas de la ciencia y la tecnología. En particular, en el campo de la estadística, donde los análisis suelen requerir cálculos intensivos, la programación paralela ofrece grandes ventajas en términos de eficiencia y escalabilidad.

El primer objetivo de este trabajo es adquirir una comprensión profunda de los fundamentos teóricos de la programación paralela. Esto implica estudiar los diferentes modelos de programación paralela, como el modelo de memoria compartida y el modelo de memoria distribuida, así como las técnicas y herramientas utilizadas en este enfoque.

El siguiente objetivo consiste en investigar cómo se puede aplicar la programación paralela en el campo de la estadística. Se analizarán las técnicas estadísticas comunes que involucran cálculos intensivos y se identificarán las oportunidades para mejorar el rendimiento mediante la programación paralela. Algunas áreas de interés podrían ser el ajuste de modelos lineales, el análisis de regresión o la simulación de Monte Carlo.

Basándonos en el conocimiento adquirido en los dos objetivos anteriores, se procederá al diseño e implementación de algoritmos paralelos específicamente adaptados a los problemas estadísticos identificados. Esto implicará la adaptación de algoritmos existentes o la creación de nuevos algoritmos que aprovechen al máximo las capacidades de paralelización.

Una vez implementados los algoritmos paralelos, se realizará una evaluación exhaustiva del rendimiento. Se medirán los tiempos de ejecución y se compararán con los algoritmos secuenciales o no paralelos equivalentes. También se analizará la escalabilidad de los algoritmos paralelos en función del tamaño del problema y de la cantidad de recursos computacionales disponibles.

Finalmente, se presentarán los resultados obtenidos y se discutirán las conclusiones del estudio. Se destacarán las mejoras en el rendimiento logradas mediante la programación paralela y se analizarán las limitaciones y desafíos encontrados durante la implementación y evaluación de los algoritmos. Además, se identificarán posibles áreas de mejora y se propondrán recomendaciones para futuras investigaciones en el campo de la programación paralela aplicada a la estadística.

1.4. Asignaturas relacionadas

Matemática discreta, Inferencia estadística II, Computación estadística, Análisis de datos, Fundamentos de Programación, Técnicas de aprendizaje automático.

2. Programación en paralelo

2.1. Descripción del problema

La utilización del paralelismo ha sido aplicada durante un largo período de tiempo, especialmente en el ámbito de la computación de alto rendimiento. Sin embargo, recientemente ha habido un creciente interés en este tipo de programación, motivado por las restricciones físicas que impiden el aumento de la frecuencia de procesamiento.

La programación paralela involucra múltiples aspectos que difieren de la programación secuencial. Al diseñar un programa paralelo, es necesario tener en cuenta diversos elementos, como el tipo de arquitectura en la que se ejecutará el programa, los requisitos de tiempo y espacio de la aplicación, el modelo de programación paralela más adecuado para implementarla, así como la forma de coordinar y comunicar entre diferentes procesadores para resolver un problema compartido.

Existen varios tipos de paralelismo:

- Paralelismo a nivel de bit
- Paralelismo a nivel de instrucción
- Paralelismo de tareas
- Paralelismo de datos

2.2. Conceptos Clave

A. Tareas: Se refieren a segmentos computacionales que se consideran discretos desde un punto de vista lógico. Una tarea se compone de un conjunto de directivas que serán procesadas por una unidad de procesamiento.

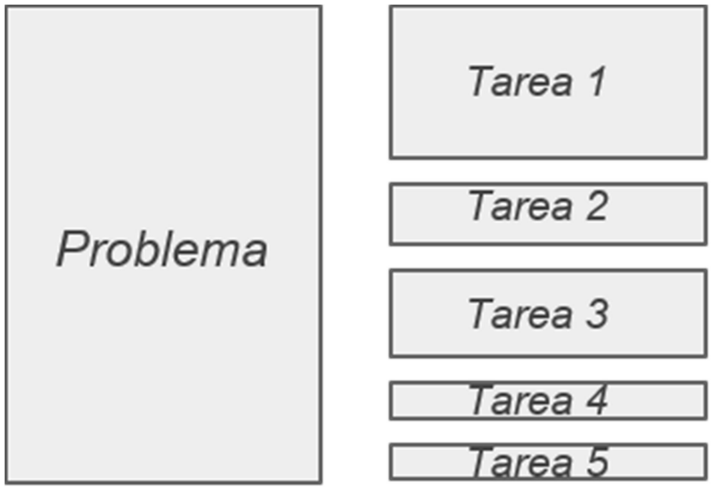

Figura 1. Tareas [6]

B. Hilos: Un proceso principal que requiere un alto consumo de recursos puede descomponerse en múltiples subprocesos más ligeros, los cuales se ejecutan de forma simultánea. Cada uno de estos subprocesos se denomina hilo. Estos hilos se intercambian información entre sí a través de una memoria compartida global.

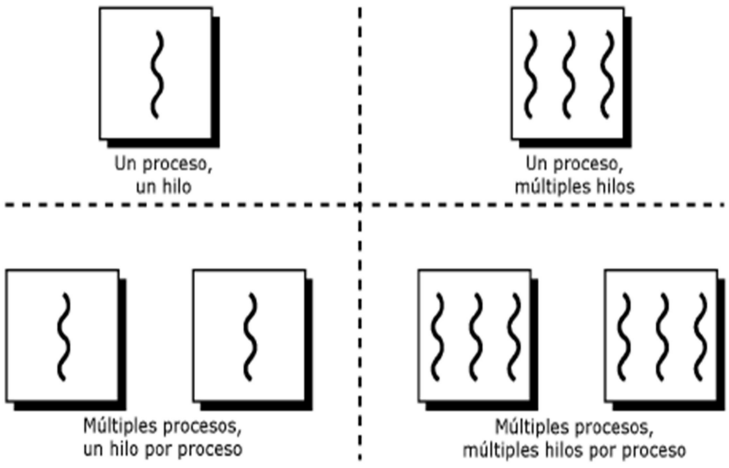

Figura 2. Hilos [7]

2.3. Ventajas y desventajas

2.3.1. Ventajas:

- Resuelve problemas que son demasiado complejos para ser realizados en una sola unidad de procesamiento central (CPU).
- Resuelve problemas que, de otra manera, requerirían un tiempo excesivo para su resolución.

- Permite abordar problemas de mayor envergadura y complejidad.
- Proporciona una ejecución más rápida del código (aceleración).
- Permite ejecutar un mayor número de problemas en general.
- Obtención de resultados en un menor tiempo.
- Permite la ejecución simultánea de múltiples instrucciones.
- Permite la división de una tarea en partes independientes.
- Ofrece un mejor equilibrio entre rendimiento y costo en comparación con la computación secuencial.
- Presenta una gran capacidad de expansión y escalabilidad.

2.3.2. Desventajas:

- Incremento en el consumo de energía.
- Mayor complejidad al escribir programas.
- Dificultad para lograr una sincronización y comunicación efectiva entre las tareas.
- Retardos causados por la comunicación entre las diferentes tareas.
- El número de componentes utilizados aumenta la probabilidad de posibles fallos.
- Elevados costos asociados a la producción y mantenimiento.
- Condiciones de carrera:
    - Cuando múltiples procesos comparten recursos y el resultado de su ejecución depende del orden en el que llegan, se generan situaciones de condición de carrera.
    - Si los procesos que están en condición de carrera no son correctamente sincronizados, puede producirse una corrupción de dato Si los procesos en condición de carrera no son sincronizados de manera adecuada, existe el riesgo de que se produzca corrupción de datos.

2.4. Resolución de problemas en paralelo

La resolución de problemas en paralelo es una técnica utilizada en computación que permite abordar y resolver problemas complejos dividiéndolos en subproblemas más pequeños que se ejecutan simultáneamente en múltiples unidades de procesamiento. En lugar de resolver un problema en secuencia, la computación paralela aprovecha el poder de cálculo de varios procesadores para acelerar el tiempo de ejecución y mejorar la eficiencia computacional.

En el contexto de la estadística, la computación paralela puede ser extremadamente útil para realizar análisis de datos en conjuntos de información masivos o para ejecutar algoritmos complejos. Al dividir el problema en subproblemas más pequeños y resolverlos en paralelo, se pueden obtener

resultados más rápidos y manejar grandes volúmenes de datos de manera más eficiente.

Existen diferentes enfoques para implementar la computación paralela, como el procesamiento distribuido, donde múltiples sistemas independientes trabajan en conjunto para resolver un problema común, y el procesamiento en paralelo en una sola máquina, donde varios núcleos de procesador trabajan simultáneamente en diferentes tareas.

Algunas técnicas comunes utilizadas en la resolución de problemas en paralelo son:

- División de datos: Consiste en dividir los datos en subconjuntos y asignarlos a diferentes unidades de procesamiento para su procesamiento simultáneo. Cada unidad de procesamiento trabaja de forma independiente en su conjunto de datos asignado y, al final, los resultados se combinan para obtener el resultado final.
- División de tareas: Esta técnica implica dividir las tareas en hilos que se ejecutan simultáneamente en diferentes unidades de procesamiento. Cada subproceso se encarga de una parte del problema y, al final, se combinan los resultados para obtener la solución completa.
- Paralelismo a nivel de instrucción: Esta técnica se basa en la ejecución simultánea de múltiples instrucciones en diferentes núcleos de procesador. Cada núcleo ejecuta su propia instrucción en paralelo con los demás, lo que acelera el tiempo de ejecución general.
- Computación en la nube: La computación en la nube se basa en gran medida en la computación paralela para ofrecer servicios escalables y de alto rendimiento, utilizando arquitecturas distribuidas y paralelas para satisfacer las demandas de computación de manera eficiente y confiable.
- Procesamiento de imágenes y gráficos: Las aplicaciones de procesamiento de imágenes, animación por computadora y gráficos en 3D requieren procesamiento paralelo para realizar cálculos intensivos en tiempo real, dividiendo la carga de trabajo para acelerar la generación y representación de imágenes.
- Simulaciones y modelado: La computación paralela se utiliza en simulaciones científicas y de ingeniería para resolver modelos matemáticos complejos, dividiendo los cálculos en múltiples procesadores para obtener resultados más rápidos y precisos.
- Redes neuronales y aprendizaje profundo: Los algoritmos de aprendizaje automático y las redes neuronales profundas requieren cálculos matemáticos intensivos, y la computación paralela acelera el entrenamiento y la inferencia al distribuir estos cálculos en múltiples unidades de procesamiento, como las GPUs.

Es importante destacar que la resolución de problemas en paralelo no siempre es la mejor opción. Dependiendo de la naturaleza del problema y del hardware disponible, es posible que el costo computacional y la complejidad de implementación superen los beneficios obtenidos. Es fundamental realizar un análisis detallado de cada caso para determinar si la computación paralela es la estrategia más adecuada.

En resumen, la resolución de problemas en paralelo es una técnica poderosa que permite abordar problemas complejos de manera eficiente mediante la ejecución simultánea de subproblemas en múltiples unidades de procesamiento. En el campo de la estadística, esta técnica puede ser especialmente útil para analizar grandes volúmenes de datos y ejecutar algoritmos complejos. Sin embargo, es importante evaluar cuidadosamente cada caso para determinar si la computación paralela es la estrategia óptima en términos de rendimiento y costo.

3. Programación en paralelo en estadística computacional

En el ámbito de la Estadística Computacional, hay diversos algoritmos en paralelo y así sacar partido del potencial de los sistemas con múltiples procesadores o núcleos. Algunos ejemplos son:

- Cadenas de Markov Monte Carlo: Los algoritmos MCMC, como el muestreo de Gibbs y el muestreo de Metropolis-Hastings, se utilizan para realizar inferencia estadística y explorar distribuciones de probabilidad. Generan secuencias de muestras que convergen hacia la distribución objetivo. En el caso de distribuciones con muchas dimensiones, es posible implementar versiones paralelas de MCMC que generen múltiples cadenas de Markov de forma simultánea para acelerar la convergencia.
- Muestreo de Gibbs: El muestreo de Gibbs es una técnica ampliamente utilizada en inferencia estadística. Mediante este método, es posible obtener muestras de una distribución conjunta de variables aleatorias al condicionar cada variable a los valores actuales de las demás. En el caso del muestreo de Gibbs paralelo, se pueden muestrear múltiples variables de manera simultánea, haciendo uso de diferentes núcleos de procesamiento.
- Bootstrap: El método de bootstrap es una técnica de remuestreo utilizada para estimar la distribución de un estadístico de interés. Consiste en generar múltiples muestras de datos a partir de la muestra original mediante muestreo con reemplazo. En el bootstrap paralelo, es posible generar múltiples muestras bootstrap de forma paralela utilizando distintos hilos o procesadores, lo cual agiliza el cálculo de las estimaciones bootstrap.
- Algoritmos de clustering: Los algoritmos de clustering, como el k-means o el clustering jerárquico, se encargan de agrupar conjuntos de datos similares. Estos algoritmos se prestan para la implementación en paralelo al asignar diferentes puntos de datos o iteraciones a distintos procesadores o núcleos.
- Métodos de Monte Carlo: Los métodos de Monte Carlo son técnicas estadísticas que se basan en el muestreo aleatorio para aproximar soluciones numéricas. Algunos ejemplos de algoritmos de Monte Carlo, como el muestreo de importancia o el muestreo por cadena de Markov Monte Carlo (MCMC), pueden implementarse en paralelo para mejorar la eficiencia computacional. En estos casos, distintas cadenas de Markov o muestras de importancia se pueden calcular de forma simultánea en diferentes procesadores o núcleos.
- Procesamiento de grandes volúmenes de datos: Con el crecimiento exponencial de los datos, como el análisis de big data, la inteligencia artificial y el aprendizaje automático (1,4), la computación paralela permite un

procesamiento rápido y eficiente al dividir la carga de trabajo en múltiples procesadores o nodos de computación.

- Supercomputación: Los superordenadores y los clústeres de alto rendimiento utilizan técnicas de computación paralela para resolver problemas científicos y de ingeniería complejos, dividiendo el trabajo en múltiples nodos para obtener resultados más rápidos.

Estos son solo algunos ejemplos de algoritmos utilizados en Estadística Computacional que se pueden programar en paralelo. La programación en paralelo resulta especialmente útil para agilizar el procesamiento de grandes conjuntos de datos o para realizar tareas computacionalmente intensivas en estadística.

En este caso, se hará especial hincapié en el Método de Monte Carlo.

Método MonteCarlo

El método de Monte Carlo (2,9) es una técnica estadística que utiliza números aleatorios para estimar valores desconocidos o realizar cálculos numéricos. Se llama así por el casino de Monte Carlo en Mónaco. Se aplica en campos como física, matemáticas, ingeniería, finanzas y ciencias de la computación, y su objetivo es estimar propiedades de un sistema mediante muestras aleatorias. Cuando es impracticable enumerar todas las combinaciones posibles, este método genera muestras aleatorias para hacer estimaciones aproximadas, mejorando la precisión con más muestras.

La variante paralela del método de Monte Carlo usa múltiples procesadores para acelerar los cálculos, dividiendo las tareas en subconjuntos y asignándolos a diferentes procesadores. Cada procesador genera menos muestras y realiza cálculos en su subconjunto, combinando luego los resultados parciales para obtener la estimación final. Esto reduce significativamente el tiempo de cálculo y requiere un entorno de programación paralela, siendo eficiente si se logra dividir y distribuir el trabajo adecuadamente entre los procesadores.

4. Ejemplos de simulación en paralelo en R

A continuación se explicarán unos ejemplos, siendo los primeros más simples, y los últimos algo más complejos.
Antes de abordar los ejemplos, se va a exponer las funciones principales que se van a utilizar:

- library(foreach): Esta línea carga el paquete "foreach" en el entorno de trabajo de R. El paquete "foreach" proporciona una construcción de bucle flexible y eficiente que se puede utilizar para iterar sobre una secuencia de valores en paralelo o en serie.
- library(doParallel): Esta línea carga el paquete "doParallel" en el entorno de trabajo de R. El paquete "doParallel" proporciona un backend para el paquete "foreach" que permite ejecutar bucles en paralelo utilizando múltiples núcleos de la CPU.
- library(microbenchmark): Esta línea carga el paquete "microbenchmark" en el entorno de trabajo de R. El paquete "microbenchmark" se utiliza para medir el tiempo de ejecución de expresiones en R.
- simulacion_dados_tiradas <- function(num_dados, num_tiradas) {: Esta línea define una función llamada "simulacion_dados_tiradas" que toma dos argumentos: "num_dados" (el número de dados a tirar en cada tirada) y "num_tiradas" (el número total de tiradas a simular).
- num_cores <- detectCores(): Esta línea utiliza la función "detectCores()" del paquete "parallel" para determinar el número de núcleos de CPU disponibles en el sistema y lo asigna a la variable "num_cores".
  Es importante destacar que el número de núcleos detectados por detectCores() no siempre coincide con el número físico de núcleos de CPU en el sistema, ya que algunos sistemas pueden tener tecnologías de hiperprocesamiento o múltiples subprocesos por núcleo. Sin embargo, la función generalmente devuelve un valor cercano al número físico de núcleos disponibles para su uso.
- registerDoParallel(cores = num_cores): Esta línea registra el backend "doParallel" y especifica el número de núcleos a utilizar para la ejecución en paralelo. Se utiliza la variable "num_cores" para indicar el número de núcleos determinado en la línea anterior.
- tiempo <- microbenchmark(: Esta línea inicia la medición de tiempo utilizando la función "microbenchmark". El código dentro de los paréntesis es la expresión que se va a medir.
- resultados <- foreach(i = 1:num_tiradas, .combine = rbind) %dopar% { … }, times = 1): Esta línea inicia un bucle "foreach" que se ejecutará en paralelo. El bucle iterará desde 1 hasta "num_tiradas". La opción ".combine = rbind" indica

que los resultados de cada iteración se combinarán en una matriz utilizando la función "rbind". El operador "%dopar%" indica que las iteraciones del bucle se ejecutarán en paralelo. La línea times = 1 especifica que la medición de tiempo se realizará solo una vez.

- sum(sample(1:6, num_dados, replace = TRUE)): Esta línea genera una muestra aleatoria de números del 1 al 6 (simulando el resultado de tirar un dado) con tamaño "num_dados" (el número de dados a tirar en cada tirada). La función "sample" selecciona aleatoriamente los números del 1 al 6, y la función "sum" calcula la suma de los números seleccionados.
- distribucion <- table(resultados) / num_tiradas: Esta línea calcula la distribución de frecuencias de los resultados obtenidos en las tiradas. La función "table" cuenta las ocurrencias de cada resultado en el vector "resultados". Dividiendo el resultado por "num_tiradas" se obtiene la proporción de veces que ocurrió cada resultado.
- stopImplicitCluster(): Esta línea detiene el backend "doParallel" y restaura el comportamiento secuencial para las futuras operaciones.
- return(list(distribucion = distribucion, tiempo = tiempo)): Esta línea devuelve una lista que contiene la distribución de frecuencias calculada y el objeto de tiempo devuelto por la medición de tiempo.
    - resultado <- simulacion_dados_tiradas(7, 100000): Esta línea llama a la función "simulacion_dados_tiradas" con los argumentos "7" y "100000". Realiza una simulación de 1000 tiradas de 3 dados, calcula la distribución de frecuencias y mide el tiempo de ejecución.
- print(resultado$distribucion): Esta línea imprime la distribución de frecuencias obtenida del resultado de la función.
- print(resultado$tiempo): Esta línea imprime el objeto de tiempo que contiene los resultados de la medición de tiempo.
- Sys.time(): La función Sys.time() es una función utilizada en el lenguaje de programación R para obtener la fecha y hora actual del sistema. Cuando se llama a Sys.time(), devuelve un objeto de clase "POSIXct" que representa la fecha y hora actual en el sistema.

  La clase "POSIXct" es un tipo de dato en el lenguaje de programación R que se utiliza para representar fechas y horas. El término "POSIX" se refiere a los estándares del sistema operativo UNIX, y "ct" significa "class timestamp" (marca de tiempo de clase). Proporciona una forma conveniente de realizar operaciones y cálculos con fechas y horas. La clase "POSIXct" se basa en el estándar POSIX, que es una especificación para sistemas operativos y software que define la forma en que se representa y manipula el tiempo.

  Al llamar a Sys.time(), obtendrás la fecha y hora actual del sistema en el formato "AAAA-MM-DD hh:mm:ss", donde "AAAA" representa el año, "MM" el

mes, "DD" el día, "hh" la hora en formato de 24 horas, "mm" los minutos y "ss" los segundos.

- parSapply: Es una función en R que permite realizar cálculos en paralelo utilizando múltiples núcleos de procesamiento. Esta función forma parte del paquete parallel en R y se utiliza principalmente cuando se necesita aplicar una función a una lista o vector en paralelo. Divide los elementos de la lista o vector en partes más pequeñas y distribuye esas partes a los núcleos de procesamiento disponibles para realizar los cálculos de manera simultánea. Esto puede acelerar significativamente el procesamiento cuando se tienen grandes conjuntos de datos o cálculos intensivos.
  La sintaxis básica de parLapply es la siguiente:
  parSapply(cl, X, FUN, ...) , donde:
  - cl: Es el clúster de computación creado previamente utilizando la función makeCluster. Representa los núcleos de procesamiento disponibles para realizar los cálculos en paralelo.
  - X: Es la lista o vector sobre el cual se aplicará la función FUN en paralelo.
  - FUN: Es la función que se aplicará a cada elemento de X en paralelo. Puede ser una función definida por el usuario o una función predefinida en R.
  - ...: Son argumentos adicionales que se pueden pasar a la función FUN.

  A diferencia de la función sapply, que realiza los cálculos en serie de forma secuencial, parSapply distribuye los cálculos en paralelo, lo que puede mejorar considerablemente el rendimiento en sistemas con múltiples núcleos.

  Es importante destacar que para utilizar parSapply es necesario tener instalado el paquete parallel y también contar con un entorno que admita la computación paralela, como una máquina con múltiples núcleos o un clúster de computación.

- montecarlo_pi: La función montecarlo_pi es una función que utiliza el método de Monte Carlo para estimar el valor de π (pi) mediante la generación aleatoria de puntos dentro de un círculo unitario inscrito en un cuadrado.
  Los pasos que sigue la función son los siguientes:
  - Paso 1: Toma un argumento llamado "datos", que se espera que sea un marco de datos (data frame) con dos columnas llamadas *X* e *Y*. Estas columnas representan las coordenadas x e y de los puntos generados aleatoriamente.
  - Paso 2: Determina el número de filas en el marco de datos *datos* utilizando la función nrow() y lo asigna a la variable *n*. La función nrow()

es una función que se utiliza para determinar la cantidad de filas en un objeto de datos en R.

- Paso 3: Se inicializa una variable llamada *puntos_dentro_circulo* con el valor 0. Esta variable se utilizará para contar la cantidad de puntos generados que están dentro del círculo unitario.
- Paso 4: A continuación, se inicia un bucle "for" que recorre los números del 1 a *n* (número de filas en el marco de datos). Dentro del bucle, se extraen las coordenadas x e y del punto actual del marco de datos y se asignan a las variables *x* e *y*, respectivamente.
- Paso 5: Se verifica si el punto *(x, y)* está dentro del círculo unitario utilizando la ecuación del círculo: x^2 + y^2 <= 1 mediante un bucle if. Si esta condición se cumple, significa que el punto está dentro del círculo, por lo que se incrementa el contador *puntos_dentro_circulo* en 1.
- Paso 6: Una vez se hayan recorrido todos los puntos (*x,y*), se calcula el valor aproximado de π dividiendo la cantidad de puntos dentro del círculo *puntos_dentro_circul*o por el número total de puntos generados *n* y multiplicando el resultado por 4. Esto se basa en la relación entre el área del círculo y el área del cuadrado circunscrito (área del círculo / área del cuadrado = π/4).
- Paso 7: Finalmente, se devuelve el valor aproximado de π utilizando la instrucción return(pi_aproximado).

- Split: Es una función en R que se utiliza para dividir un objeto en subconjuntos más pequeños basados en uno o más factores. Esta función es útil para dividir un conjunto de datos en grupos según ciertas categorías o criterios.
  La sintaxis básica de split() es la siguiente:
  split(x, f), donde:
  - x es el objeto que se desea dividir, como un vector, un marco de datos (data frame) o una lista.
  - f es el factor o los factores que se utilizarán para determinar cómo se dividirá x. Puede ser un vector o una lista de factores.

  La función split() devuelve una lista de subconjuntos, donde cada elemento de la lista representa un grupo definido por los factores especificados. Los nombres de los elementos de la lista corresponden a los niveles de los factores utilizados para la división.

- makeCluster() y stopCluster(): La función makeCluster() es una función en R que se utiliza para crear un clúster o conjunto de nodos de procesamiento para la programación paralela. Esta función es parte del paquete 'parallel' y se utiliza para distribuir tareas y cálculos en diferentes núcleos o máquinas.
  La sintaxis básica de makeCluster() es la siguiente:
  makeCluster(n, ...), donde:

- n: El número de nodos o clústeres que se desea crear.
- ...: Son argumentos adicionales que se pueden especificar, como la especificación del tipo de clúster o las opciones de configuración.

La función makeCluster() devuelve un objeto de clúster que representa el conjunto de nodos disponibles para la ejecución paralela. Es importante detener el clúster después de su uso utilizando la función stopCluster() para liberar los recursos y permitir que los nodos vuelvan a estar disponibles para otros usos.

A continuación se procederá a explicar los ejemplos:

Ejemplo 1.

En este ejemplo de computación paralela, se va a calcular la suma de los cuadrados de una lista de números utilizando múltiples núcleos de procesamiento:

```
# Carga del paquete 'parallel'
library(parallel) (11)

# Crear una lista de números
numeros <- 1:10^6

# Especificar el número de núcleos de procesamiento a utilizar
num_nucleos <- detectCores()

# Dividir la lista de números en partes iguales para cada núcleo
particiones <- split(numeros, 1:num_nucleos)

# Iniciar el clúster paralelo
cl <- makeCluster(num_nucleos)

# Enviar cada partición de números a un núcleo diferente para realizar los cálculos
resultados_parciales <- parLapply(cl, particiones, function(x) sum(x^2))

# Combinar los resultados parciales en un solo vector
resultado_final <- unlist(resultados_parciales)

# Calcular la suma total de los cuadrados
suma_cuadrados <- sum(resultado_final)

# Detener el clúster paralelo
stopCluster(cl)
```

```
# Imprimir el resultado
print(suma_cuadrados)
```

En este ejemplo, se utiliza la función detectCores() para determinar el número de núcleos disponibles en tu sistema. Luego, se divide la lista de números en partes iguales utilizando la función split(). A continuación, se inicia un clúster paralelo con makeCluster() y se utiliza parLapply() para aplicar la función anónima a cada partición de números de forma paralela.

Después de obtener los resultados parciales, se combinan en un solo vector utilizando unlist() y se calcula la suma total de los cuadrados con sum(). Finalmente, se detiene el clúster paralelo con stopCluster() y se muestra el resultado.

Antes de empezar con los otros dos ejemplos, se van a presentar otros medios para calcular el tiempo que tardan diferentes métodos de computación en paralelo usando este primer ejemplo:

Código para cargar todo lo necesario:

```
# Carga del paquete 'parallel'
library(parallel) (11)

# Especificar el número de núcleos de procesamiento a utilizar
num_nucleos <- detectCores()- 1

# Crear una lista de números
numeros <- 1:10^6
```

Se usarán los núcleos disponibles menos uno, salvo en el caso de dos núcleos, en el que se utilizarán todos los núcleos.

Método 1. Aplicar la función directamente, sin programación en paralelo.

El propósito de este código es medir el tiempo que tarda en ejecutar una operación (en este caso, elevar al cuadrado una serie de números) y calcular el tiempo promedio de dicha operación en 1000 iteraciones.

```
# Crear una matriz para almacenar los tiempos de ejecución

tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico

# Iterar 1000 veces
```

```
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  sum(numeros^2)           # Operación a medir
  fin_fun <- Sys.time()     # Capturar tiempo de fin
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 2. Aplicar la función utilizando la función "sapply".

El objetivo de este código es medir el tiempo que tarda en ejecutar una operación (en este caso, elevar al cuadrado una serie de números), que utiliza la función "sapply", y calcular el tiempo promedio de dicha operación en 1000 iteraciones.

```
# Crear una matriz para almacenar los tiempos de ejecución
tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico
# Iterar 1000 veces
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  sum(sapply(numeros, function(x) x^2))          # Operación a medir
  fin_fun <- Sys.time()     # Capturar tiempo de fin
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 3. Aplicar la función utilizando la función "parSapply".

El objetivo de este código es realizar un cálculo paralelo para medir el tiempo de ejecución de elevar al cuadrado una serie de números y sumar los resultados, usando el paquete parallel de R y calcular el tiempo promedio de dicha operación en 1000 iteraciones.

```
# Crear una matriz para almacenar los tiempos de ejecución
tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico
# Iterar 1000 veces
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  cl <- makeCluster(num_nucleos)  # Crear un clúster de procesamiento paralelo
  sum(parSapply(cl, numeros, function(x) x^2))          # Operación a medir
  stopCluster(cl)    # Detener el clúster
  fin_fun <- Sys.time()    # Capturar tiempo de fin
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 4. Aplicar la función utilizando la función "parSapply" y la función "split".

Este código mide el tiempo de ejecución de la suma de los cuadrados de una serie de números, utilizando procesamiento paralelo con la función parSapply y registrando los tiempos en 1000 iteraciones

```
# Crear una matriz para almacenar los tiempos de ejecución
tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico
# Iterar 1000 veces
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  cl <- makeCluster(num_nucleos)  # Crear un clúster de procesamiento paralelo
  sum(parSapply(cl, split(numeros, 1:num_nucleos), function(x) sum(x^2))) # Operación a medir
  stopCluster(cl)    # Detener el clúster
  fin_fun <- Sys.time()    # Capturar tiempo de fin
```

```
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 5. Aplicar la función utilizando la función "parSapply" con particiones en paralelo.

El código está diseñado para medir el tiempo de ejecución de una operación paralela (sumar cuadrados de una serie de números) en 1000 iteraciones.

```
# Crear una matriz para almacenar los tiempos de ejecución
tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico
# Iterar 1000 veces
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  cl <- makeCluster(num_nucleos)  # Crear un clúster de procesamiento paralelo
  particiones <- split(numeros, 1:num_nucleos)# Dividir el vector 'numeros' en partes iguales según el número de núcleos
  sum(parSapply(cl, particiones, function(x) sum(x^2)))# Operación a medir
  stopCluster(cl)    # Detener el clúster
  fin_fun <- Sys.time()    # Capturar tiempo de fin
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 6. Aplicar la función utilizando la función "parSapply" con particiones en paralelo usando "unlist".

El código está diseñado para medir el tiempo de ejecución de una operación paralela (sumar cuadrados de una serie de números) en 1000 iteraciones.

```
# Crear una matriz para almacenar los tiempos de ejecución
tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico
# Definición de la función para calcular la suma de los cuadrados
suma_cuadrados <- function(indices) {
  suma <- 0
  indices<-sort(indices)
  i <- indices[1]
  j <- indices[2]-indices[1]
  while (i <= max(indices)) {
    suma <- suma + i^2
    i <- i + j
  }
  return(suma)
}
# Iterar 1000 veces
for (i in 1:1000) {
  inicio_fun <- Sys.time()  # Capturar tiempo de inicio
  cl <- makeCluster(num_nucleos)  # Crear un clúster de procesamiento paralelo
  particiones <- split(numeros, 1:(num_particiones))# Dividir el vector 'numeros' en partes iguales según el número de particiones
  sum(unlist(parLapply(cl, particiones, suma_cuadrados)))# Operación a medir
  stopCluster(cl)     # Detener el clúster
  fin_fun <- Sys.time()     # Capturar tiempo de fin
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}
# Calcular la media de los tiempos registrados
mean(tiempos)
```

Método 7. Aplicar la función utilizando la función "parSapply" con particiones en paralelo usando "unlist" cambiando la distribución de las particiones.

El código está diseñado para medir el tiempo de ejecución de una operación paralela (sumar cuadrados de una serie de números) en 1000 iteraciones, pero ahora las particiones, en lugar de dividirse por defecto (primera partición: 1, num_particiones, 2*num_particiones …), se dividendo tal forma que los números vayan de 1 a num_particiones (primera partición: 1, 2, 3…, num_particiones).

```
# Crear una matriz para almacenar los tiempos de ejecución

tiempos <- numeric(1000)  # Usamos numeric para inicializar un vector numérico

# Definición de la función para calcular la suma de los cuadrados

suma_cuadrados <- function(indices) {

 suma <- 0

 i <- min(indices)

 while (i <= max(indices)) {

  suma <- suma + i^2

  i <- i + 1

 }

 return(suma)

}

# Iterar 1000 veces

for (i in 1:1000) {

 inicio_fun <- Sys.time()  # Capturar tiempo de inicio

 cl <- makeCluster(num_nucleos)  # Crear un clúster de procesamiento paralelo

particiones <- split(numeros, cut(numeros, breaks = seq(0, length(numeros) + max(numeros)/num_particiones, by = max(numeros)/num_particiones), labels = FALSE))# Dividir el vector 'numeros' en partes iguales según el número de particiones

 sum(unlist(parLapply(cl, particiones, suma_cuadrados)))# Operación a medir

 stopCluster(cl)    # Detener el clúster

 fin_fun <- Sys.time()    # Capturar tiempo de fin
```

```
  tiempos[i] <- fin_fun - inicio_fun  # Guardar tiempo transcurrido
}

# Calcular la media de los tiempos registrados

mean(tiempos)
```

Gráficos realizados.

Se han guardado las medias de los 7 métodos, probando con distintos números de núcleos (utilizando otras computadoras) y también la media de dichas medias.

A continuación se calcula qué método es el que menos tarda según el número de núcleos de la computadora:

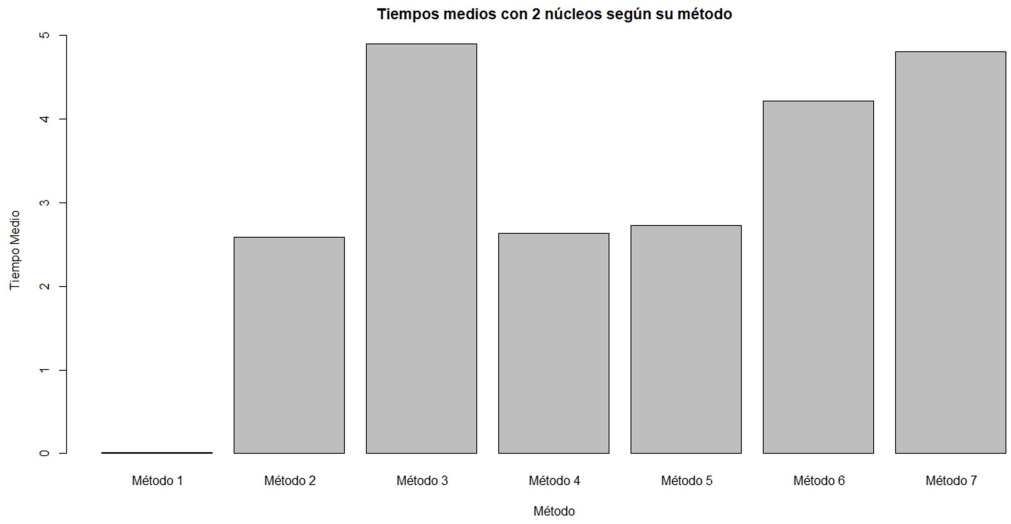

Gráfico 1. Tiempos medios según sus métodos con 2 núcleos.

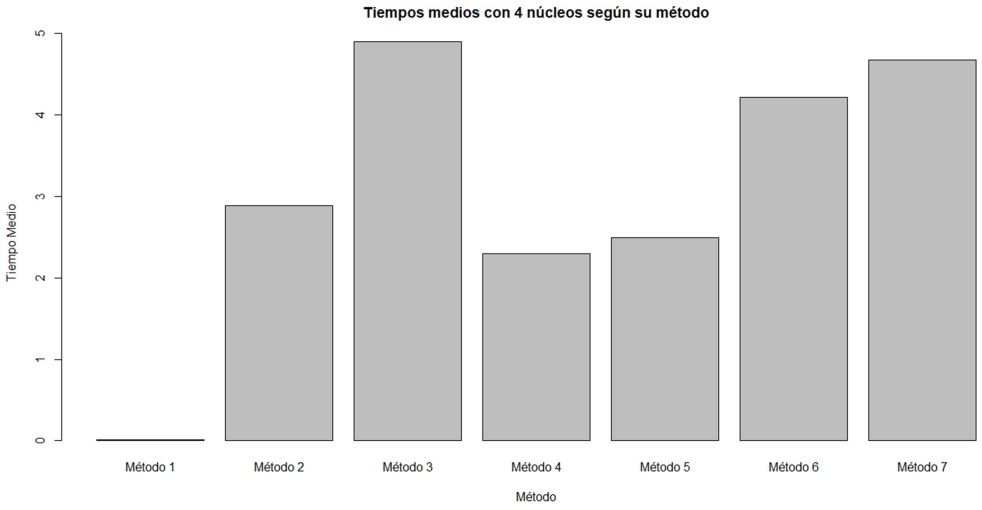

Gráfico 2. Tiempos medios según sus métodos con 4 núcleos.

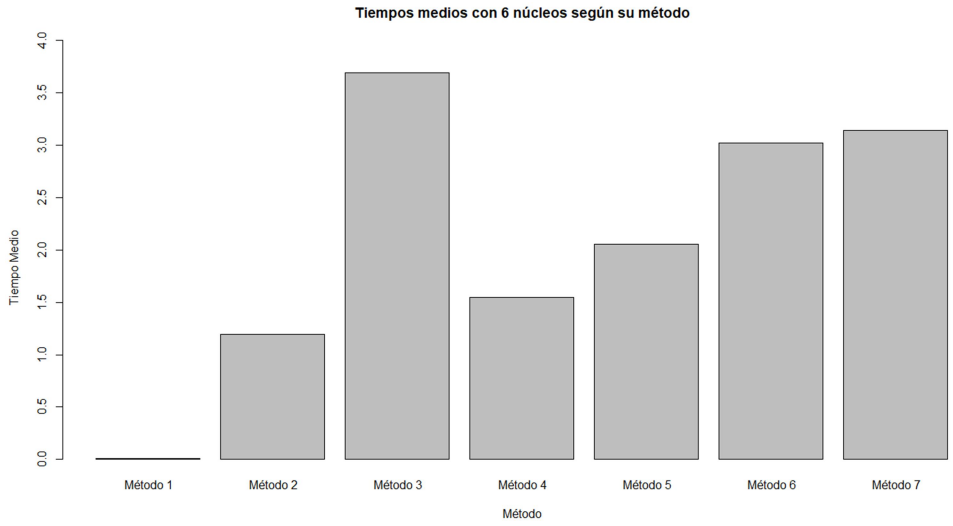

Gráfico 3. Tiempos medios según sus métodos con 6 núcleos.

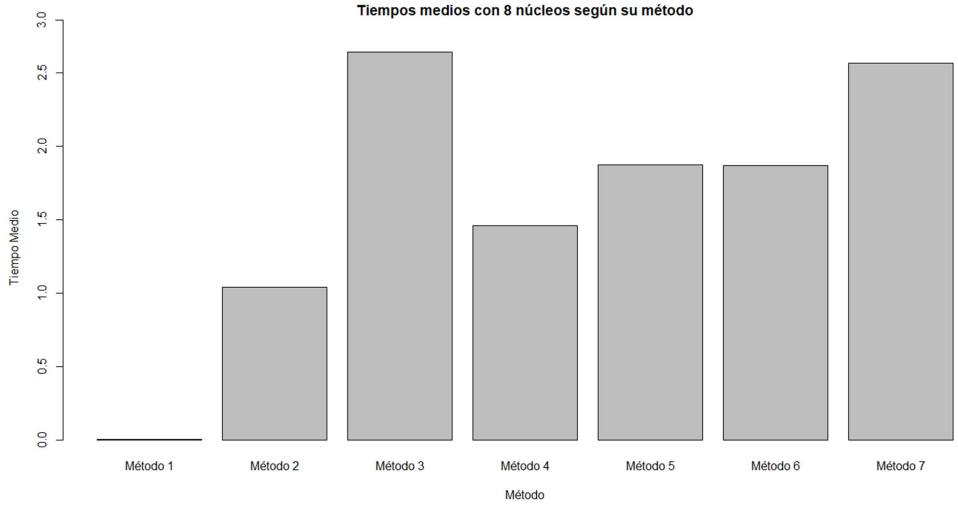

Gráfico 4. Tiempos medios según sus métodos con 8 núcleos.

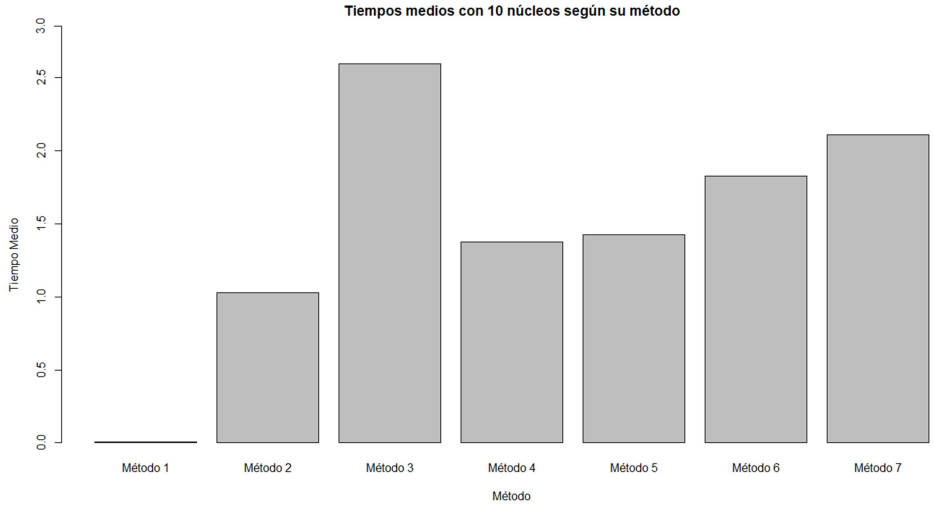

Gráfico 5. Tiempos medios según sus métodos con 10 núcleos.

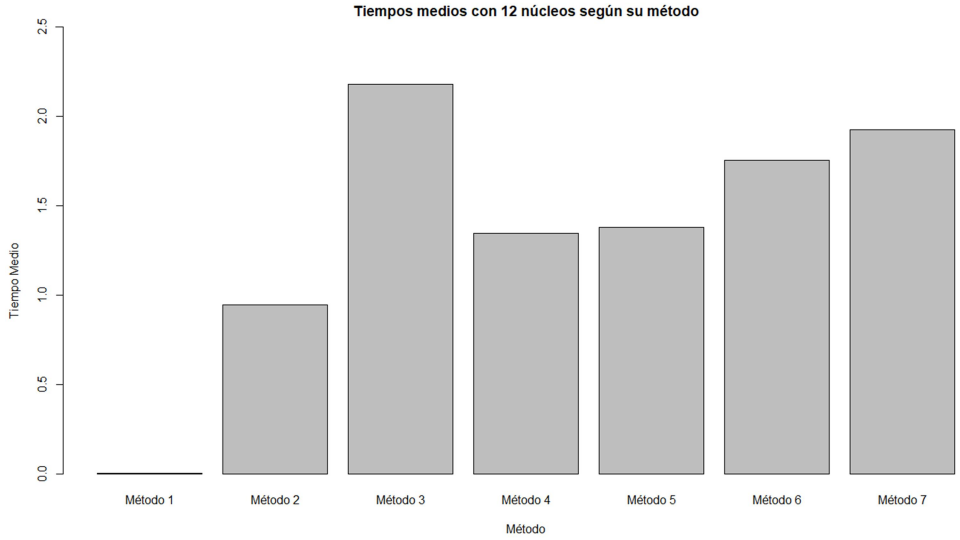

Gráfico 6. Tiempos medios según sus métodos con 12 núcleos.

Como se puede ver, el método 1 es el que menos tarda. Este método consiste en hacer la operación directamente, y el motivo de por qué es el más rápido es debido a que la operación es sencilla de ejecutar, aunque sirve para hacer pruebas. El método 2 tampoco utiliza computación paralela, por lo que también será obviado.

En los que respecta a los métodos del 3 al 7, se puede observar que el método 3 es con diferencia el ejemplo que más tarda en ejecutarse. El que menos tarda entre estos podría estar entre el método 4 y el método 5.

A continuación se va a mostrar el tiempo que tarda cada método en paralelo según el número de núcleos que tiene la computadora:

Método 3:

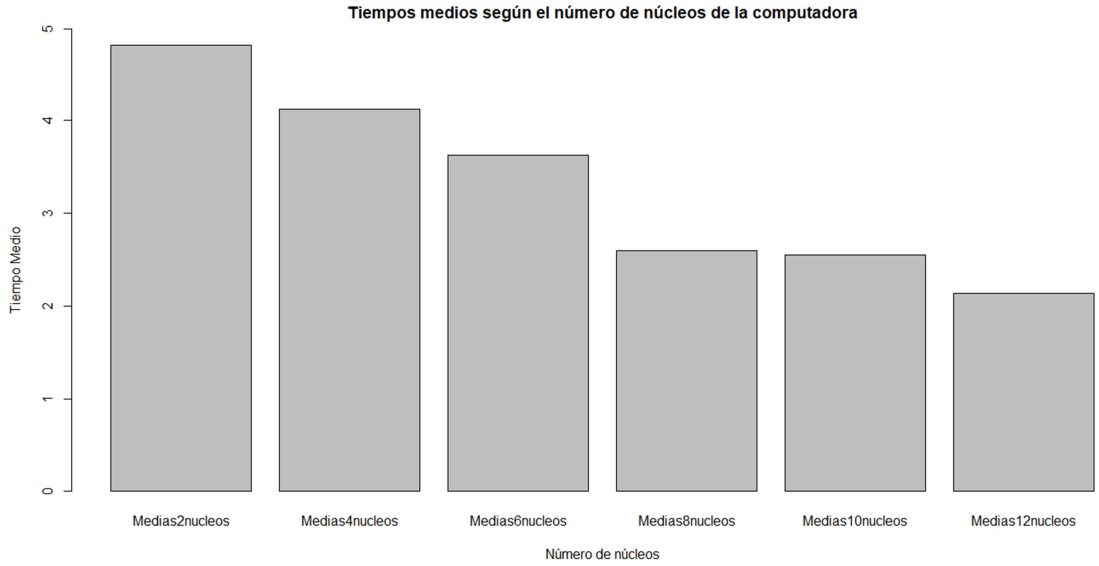

Gráfico 7. Tiempos medios con el método 3 según su número de núcleos.

Método 4:

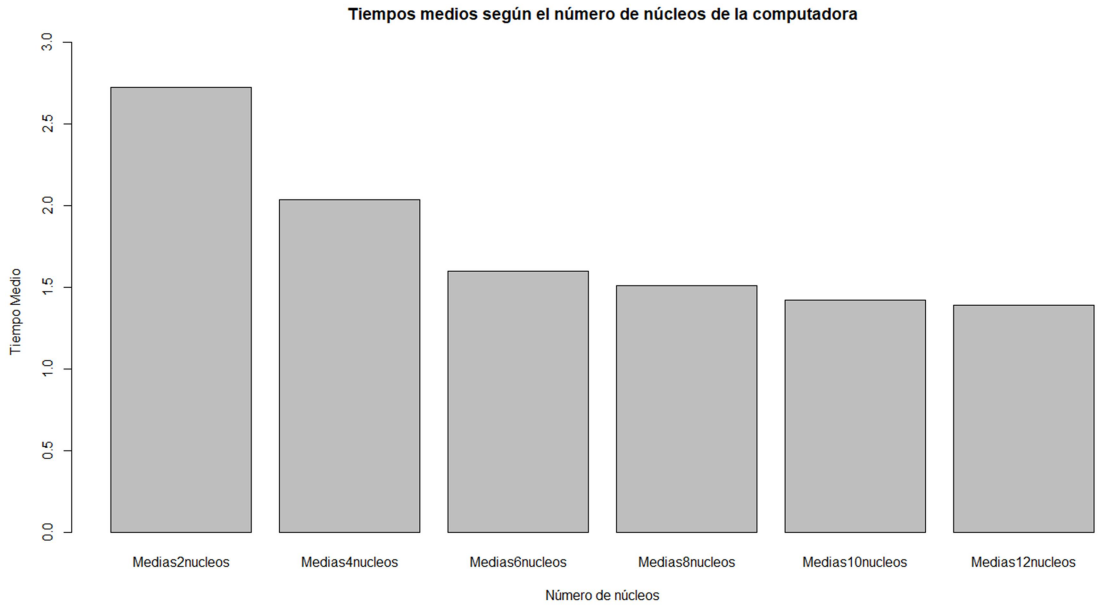

Gráfico 8. Tiempos medios con el método 4 según su número de núcleos.

Método 5:

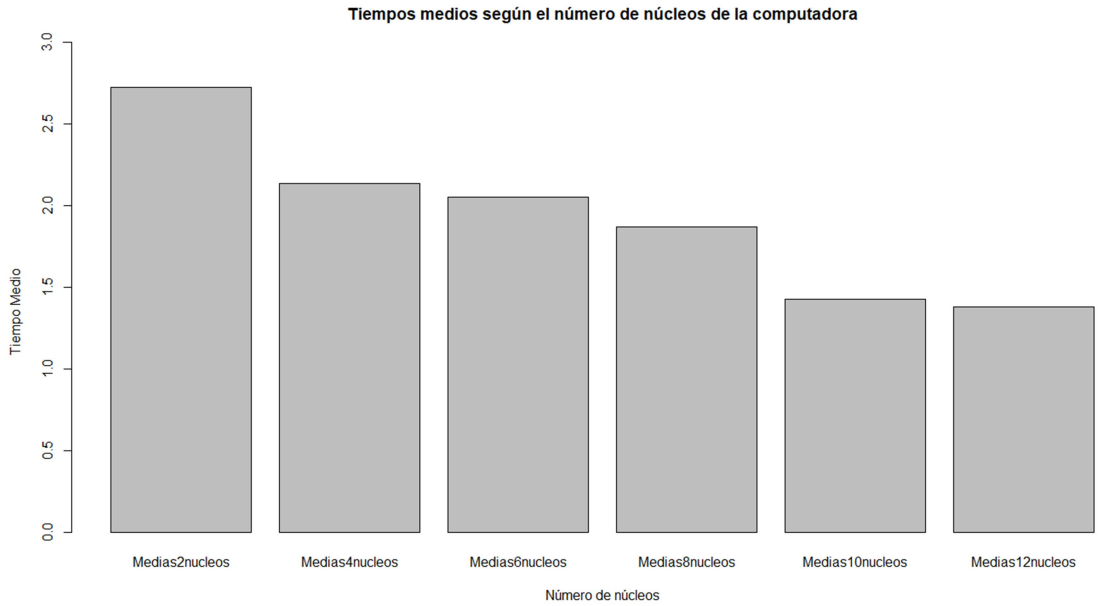

Gráfico 9. Tiempos medios con el método 5 según su número de núcleos.

Método 6:

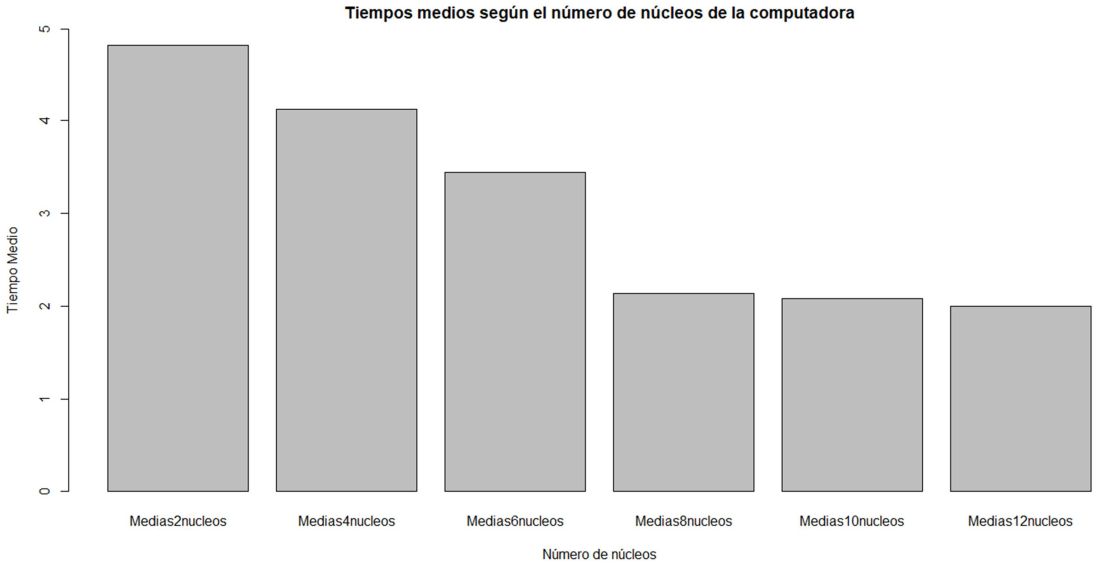

Gráfico 10. Tiempos medios con el método 6 según su número de núcleos.

Método 7:

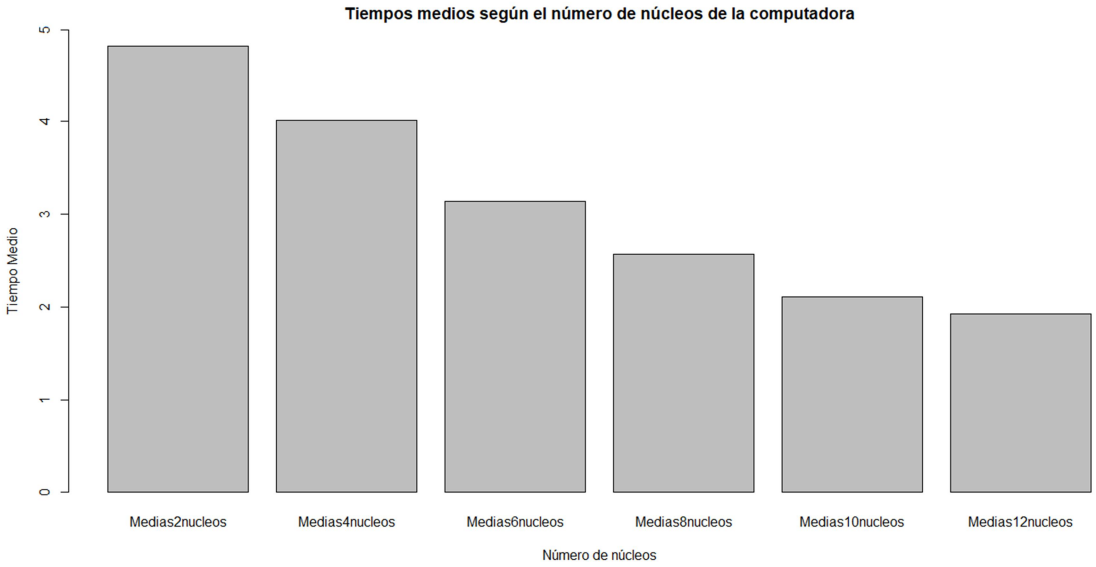

Gráfico 11. Tiempos medios con el método 7 según su número de núcleos.

Como se puede observar, a medida que aumenta el número de núcleos, disminuye el tiempo de ejecución. Esto ocurre con todos los métodos.

Esto quiere decir que, en una computadora con muchos núcleos, el tiempo de ejecución se verá considerablemente reducido.

Código utilizado para los gráficos (Pondremos solo para un ejemplo, los demás son análogos):

Abrir los archivos que necesitamos para poder hacer los gráficos.

```
tiempos<-read.csv("tiemposMedios.csv")
```

	Método	Medias2nucleos	Medias4nucleos	Medias6nucleos	Medias8nucleos	Medias10nucleos	Medias12nucleos
1	Método 1	0.01032647	0.00941223	0.006745094	0.004780139	0.004292561	0.0037082
2	Método 2	2.58355000	2.47588300	1.197287000	1.042048000	1.030739000	0.9475300
3	Método 3	4.89666100	4.19865700	3.688546000	2.640849000	2.594697000	2.1767300
4	Método 4	2.63419400	1.96904300	1.545176000	1.460686000	1.373997000	1.3472110
5	Método 5	2.72309100	2.13749900	2.053238000	1.872251000	1.426978000	1.3802070
6	Método 6	4.21633000	3.61525700	3.017500000	1.869732000	1.824967000	1.7515820
7	Método 7	4.80654400	4.01096000	3.137184000	2.567185000	2.107652000	1.9257090
8	Media	3.12438521	2.63095875	2.092239442	1.636790163	1.480474652	1.3618110

Tabla 1. Datos de los tiempos medios en segundos y la media según el número de núcleos.

```
tiempos2<-read.csv("tiemposMediosTranspuesta.csv")
```

	Número.de.núcleos	Método.1	Método.2	Método.3	Método.4	Método.5	Método.6	Método.7
1	Medias2nucleos	0.010326470	2.583550	4.896661	2.634194	2.723091	4.216330	4.806544
2	Medias4nucleos	0.009412230	2.475883	4.198657	1.969043	2.137499	3.615257	4.010960
3	Medias6nucleos	0.006745094	1.197287	3.688546	1.545176	2.053238	3.017500	3.137184
4	Medias8nucleos	0.004780139	1.042048	2.640849	1.460686	1.872251	1.869732	2.567185
5	Medias10nucleos	0.004292561	1.030739	2.594697	1.373997	1.426978	1.824967	2.107652
6	Medias12nucleos	0.003708200	0.947530	2.176730	1.347211	1.380207	1.751582	1.925709
7	Media	0.006544116	1.546173	3.366023	1.721718	1.932211	2.715895	3.092539

Tabla 2. Datos de los tiempos medios en segundos y la media según el método.

Gráficos, eliminando las medias

```
barplot(tiempos$Medias2nucleos[1:7], names.arg = tiempos$Método[1:7], main = [
"Tiempos medios con 2 núcleos según su método", xlab ="Método", ylab ="Tiempo Medio")

barplot(tiempos2$Método.1[1: 6], names.arg = tiempos2$Número.de.núcleos[1:6], main
="Tiempos medios según el número de núcleos de la computadora", xlab = "Número de
núcleos", ylab =" Tiempo Medio ")
```

También influye en el tiempo medio el tamaño del vector numeros, se procede a mostrar la forma en la que afecta el tamaño:

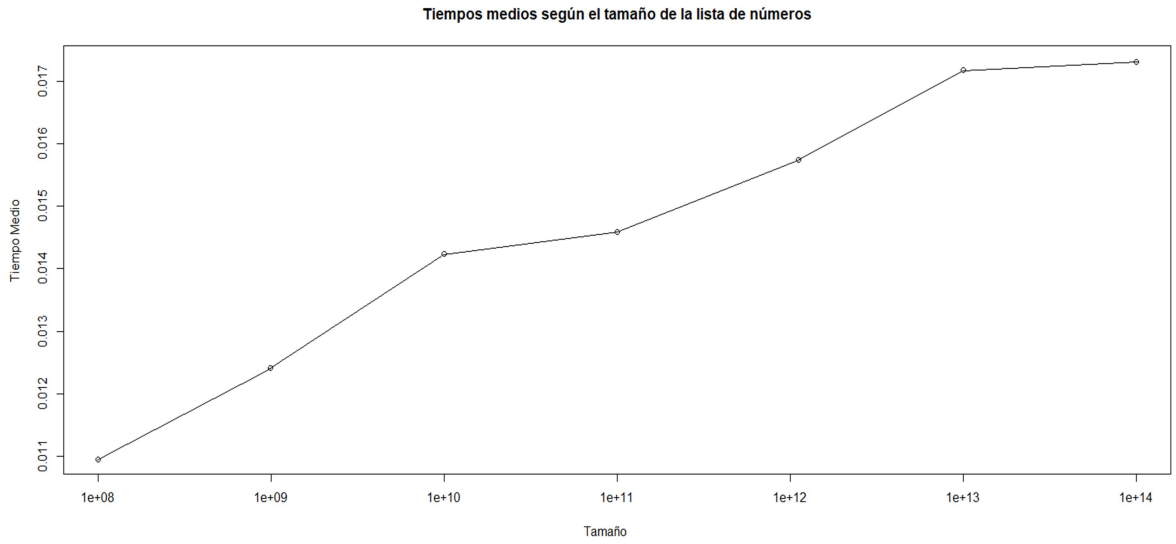

Gráfico 12. Tiempos medios según el tamaño de la lista de números.

Se observa que el tiempo medio crece con respecto al tamaño del vector. Esto se repite independientemente el número de núcleos. El tiempo disminuye si aumenta el número de núcleos y aumenta si aumenta el tamaño del vector.

Ahora que se ha mostrado cómo afecta el número de núcleos y el método que se utilice, se acompañará esta información con más ejemplos:

Ejemplo 2.

El objetivo de este ejemplo es calcular el valor de Pi utilizando Monte Carlo, distribuir el cálculo en paralelo entre múltiples núcleos de CPU para acelerar el proceso y comparar el tiempo transcurrido en hacer el cálculo sin programación paralela y con programación paralela.

Ahora se va a explicar paso por paso el código que se ha utilizado para resolver el problema.

- Primero, se ha cargado la biblioteca "parallel" con la función:
  ```
  # Cargar el paquete 'parallel'
  library(parallel)
  ```
  (11)

- Lo segundo ha sido generar los datos con una distribución uniforme bivariante:
  ```
  #Especifica el número de observaciones
      n <- 10000
      #n <- 100000
      #n <- 1000000
  ```

```
# Genera datos uniformemente distribuidos en el rango [0, 1]
    X <- runif(n)
    X <- runif(n)

# Crea un data frame con los datos
    datos <- data.frame(X = x, Y = y)
```

- Luego, se ha definido una función llamada montecarlo_pi que toma como argumentos X, Y y n. Esta función realiza el cálculo de Monte Carlo para aproximar el valor de Pi. La función está explicada anteriormente, antes de empezar los ejemplos.

```
# Definir la función para el cálculo de Monte Carlo
    montecarlo_pi <- function(datos) {
        n <- nrow(datos)
        puntos_dentro_circulo <- sum(datos$X^2 + datos$Y^2 <= 1)
        # Calcular el valor aproximado de Pi
        pi_aproximado  <-  4 * puntos_dentro_circulo / n

        return(pi_aproximado)
    }
```

- A continuación, se mide el tiempo que tarda en ejecutarse la función montecarlo_pi sin programación paralela:

```
#Se mide el tiempo que tarda en ejecutar la función sin programación paralela
```

  - Se guarda el tiempo de inicio utilizando inicio_fun <- Sys.time():

```
    # Guarda el tiempo de inicio
    inicio_fun <- Sys.time()
```

  - Se realiza el cálculo de Monte Carlo llamando a la función montecarlo_pi(datos), y el resultado se guarda en la variable pi_aproximado:

```
    # Realiza el cálculo de Montecarlo
    pi_aproximado <- montecarlo_pi(datos)
```

  - Se guarda el tiempo de finalización utilizando fin_fun <- Sys.time():

```
    # Guarda el tiempo de finalización
    fin_fun <- Sys.time()
```

  - Se calcula la diferencia de tiempo transcurrido usando tiempo_transcurrido_fun <- fin_fun - inicio_fun:

```
    # Calcula la diferencia de tiempo
    tiempo_transcurrido_fun <- fin_fun - inicio_fun
```

- Luego, se mide el tiempo que tarda en ejecutarse la función montecarlo_pi con programación paralela:

#Se mide el tiempo que tarda en ejecutar la función con programación paralela

- Se obtiene el número de núcleos de CPU disponibles utilizando num_nucleos <- detectCores():

```
# Obtener el número de núcleos de CPU disponibles
num_nucleos <- detectCores()
```

- Se divide los datos en partes iguales para cada núcleo utilizando particiones <- split(datos, 1:num_nucleos):

```
# Dividir los datos en partes iguales para cada núcleo
particiones <- split(datos ,1:num_nucleos)
```

- Se inicia el clúster de computación paralela utilizando cl <- makeCluster(num_nucleos):

```
# Iniciar el clúster de computación paralela
cl <- makeCluster(num_nucleos)
```

- Se envían los datos a cada núcleo del clúster utilizando clusterExport(cl, "montecarlo_pi") y clusterExport(cl, "particiones"):

```
# Enviar los datos a cada núcleo del clúster
clusterExport(cl, "montecarlo_pi")
clusterExport(cl, "particiones")
```

- Se guarda el tiempo de inicio utilizando inicio <- Sys.time():

```
# Guarda el tiempo de inicio
inicio <- Sys.time()
```

- Se realiza el cálculo en paralelo en cada núcleo utilizando resultado <- parLapply(cl, particiones, montecarlo_pi). La función parLapply realiza una versión paralela de la función sapply aplicando la función montecarlo_pi a cada parte de los datos:

```
# Realizar el cálculo en paralelo en cada núcleo
resultado <- parSapply(cl, particiones, montecarlo_pi)
```

- Se cierra el clúster utilizando stopCluster(cl):

```
# Cerrar el clúster
stopCluster(cl)
```

- Los resultados de cada núcleo se pasan a un vector numérico res para calcular el valor promedio de Pi:

```
#Pasar a vector numérico los valores del vector-lista resultado
res <- c(resultado[ [ "1" ]],resultado[["2"]])
```

- Se calcula el valor promedio de Pi utilizando pi_promedio <- mean(res):

```
# Calcular el valor promedio de Pi de los resultados de cada núcleo
pi_promedio <- mean(res)
```

- Se guarda el tiempo de finalización utilizando fin <- Sys.time():

```
# Guarda el tiempo de finalización
fin <- Sys.time()
```

- Se calcula la diferencia de tiempo transcurrido utilizando tiempo_transcurrido <- fin – inicio:

```
# Calcula la diferencia de tiempo
tiempo_transcurrido <- fin – inicio
```

- Luego, se definen dos funciones, una para cuando no usamos paralelismo (calculos_fun) y otra para cuando sí lo usamos (calculos_par), que generan varios datos con distribución uniforme y realiza el cálculo sin paralelismo y con paralelismo respectivamente:

```
# Función para generar varias variables uniformes y calcular la estimación de pi sin paralelismo
calculos_fun <- function(n) {
 pi_medio <- numeric(100)
 tiempos <- numeric(100)
 for (i in 1: 100) {
  x <- runif(n)
  y <- runif(n)
  datos <- data.frame(X = x, Y = y)
  inicio_fun <- Sys.time()
  pi_estimado <- montecarlo_pi(datos)
  pi_medio[i] <- pi_estimado
  fin_fun <- Sys.time()
  tiempo_transcurrido_fun <- fin_fun - inicio_fun
  tiempos[i] <- tiempo_transcurrido_fun
  i<-i+1
 }
 return(pi_medio)
#return(tiempos)

}

# Función para generar varias variables uniformes y calcular la estimación de pi con paralelismo
calculos_par<- function(n) {
 pi_medio <- numeric(100)
 tiempos <- numeric(100)
 for (i in 1: 100) {
  x <- runif(n)
  y <- runif(n)
  datos <- data.frame(X = x, Y = y)
  inicio_fun <- Sys.time()
  resultado <- parSapply(cl, particiones,montecarlo_pi)
  res <- c(resultado[ [ "1" ]],resultado[["2"]])
  pi_estimado <- mean(res)
  fin_par <- Sys.time()
```

```
  tiempo_transcurrido_par<- fin_par - inicio_par
  tiempos[i] <- tiempo_transcurrido_par
  i<-i+1
 }
 return(pi_medio)
#return(tiempos)
}
```

Se pondrá un return diferente para poder calcular también los intervalos para el tiempo, así no se repetirá código.

- A continuación, se guardarán los resultados en las variables pi_medio_fun y pi_medio_par.

```
pi_medio_fun <-calculos_fun(n) #Sin paralelismo
pi_medio_par <-calculos_par(n) # Con paralelismo
```

Se cambia el return de la función para que nos devuelva:

```
tiempo_medio_fun <-calculos_fun(n) #Sin paralelismo
tiempo_medio_par <-calculos_par(n) # Con paralelismo
```

- Después, se definirá la función intervalo, que toma como argumentos pi_medio y confianza: Esta función realiza el cálculo de intervalos de confianza dados la media y la confianza, dos variables de tipo float:

```
# Función que calcula el intervalo de confianza
intervalo<-function(pi_medio,confianza){
 media_pi <- mean(pi_medio)
 error_estandar_pi <- sd(pi_medio) / sqrt(length(pi_medio))
 z <- qnorm(1 - (1 - confianza) / 2)
 intervalo_confianza <- media_pi + c(-1,1) * z * error_estandar_pi
 return(intervalo_confianza)
}
```

- Se calculan los intervalos de confianza sin paralelismo y con paralelismo:

```
# Fijamos el nivel de confianza
confianza <- 0.95

# Calcular intervalos de confianza para pi
intervalo_confianza_pi_fun<-intervalo(pi_medio_fun,confianza) #Sin paralelismo
intervalo_confianza_pi_par<-intervalo(pi_medio_par,confianza) #Con paralelismo

# Calcular intervalos de confianza para el tiempo
```

```
intervalo_confianza_tiempo_fun<-intervalo(tiempo_medio_fun,confianza)
#Sin paralelismo
intervalo_confianza_tiempo_par<-intervalo(tiempo_medio_par,confianza)
#Con paralelismo
```

- Se crea la tabla resultados utilizando la función rbind() para unir filas de diferentes objetos de la tabla:
  - La primera fila de la tabla resultados corresponde al cálculo sin paralelismo. Se utiliza la función data.frame() para crear un objeto en la tabla con cuatro columnas: "Metodo", "Valor_Pi" y "Tiempo_Transcurrido". Los valores correspondientes se especifican utilizando las variables pi_aproximado, tiempo_transcurrido_fun e intervalo_confianza_fun.
  - La segunda fila de la tabla resultados corresponde al cálculo con paralelismo. Al igual que en la primera fila, se utiliza la función data.frame para crear un objeto en la tabla con las mismas tres columnas. Los valores correspondientes se especifican utilizando las variables pi_promedio, tiempo_transcurrido e intervalo_confianza_par.
  - Cada objeto de la tabla se separa con comas dentro de la función rbind(), lo que indica que se deben unir como filas en la tabla resultados.

```
# Crear tabla para almacenar los resultados

resultados <- data.frame (

    Metodo = c("Sin Paralelismo" , "Con Paralelismo"),

    Valor_Pi = c(pi_aproximado, pi_promedio),

    Tiempo_Transcurrido = c(tiempo_transcurrido_fun, tiempo_transcurrido)

    Intervalo_Confianza_pi=c(intervalo_confianza_pi_fun,
    intervalo_confianza_pi_par)

    Intervalo_Confianza_tiempo=c(intervalo_confianza_tiempo_fun,
    intervalo_confianza_tiempo_par)

)
```

- Al final, se imprime la tabla de resultados utilizando print(resultados):

```
#Imprimir tabla de resultados
print(resultados)
```

A continuación se procederá a comparar los resultados de los valores de π y los tiempos de ejecución entre dos enfoques: uno sin paralelismo y otro con paralelismo.

		Tamaño de n		
		n=10000	n=100000	n=1000000
Sin programación paralela	Valor aproximado de π	3.1396	3.14452	3.14038
	Tiempo transcurrido (secs)	0.5251319	4.781207	41.508754
	Intervalo de confianza del 95% para pi	[3.139474, 3.145742]	[3.140619, 3.142527]	[3.140388, 3.142176]
	Intervalo de confianza del 95% para el tiempo (secs)	[0.611947, 1.146433]	[3.513671, 5.573915]	[31.596031, 44.100032]
Con programación paralela	Valor aproximado de π	3.1396	3.14452	3.14038
	Tiempo transcurrido (secs)	0.4241078	2.511636	28.912745
	Intervalo de confianza del 95% para pi	[3.139756, 3.145916]	[3.137892, 3.143748]	[3.139973, 3.146163]
	Intervalo de confianza del 95% para el tiempo (secs)	[0.023689, 0.048899]	[2.077232, 3.827055]	[28.030922, 35.394013]

Tabla 3. Comparación de resultados de la estimación de π y del tiempo sin y con paralelismo.

En la Tabla 3, se ha obtenido el valor aproximado de π y el tiempo transcurrido en segundos sin paralelismo y con paralelismo, además de sus respectivos intervalos de confianza.

En este caso, el método con paralelismo logró reducir significativamente el tiempo de ejecución en comparación con el método sin paralelismo.

También se puede observar que, a medida que crece el tamaño de los datos, el tiempo transcurrido aumenta. Esto es normal, puesto que, a mayor tamaño, más tiempo utiliza.

Si se pone atención, también se sacan las diferencias entre los resultados anteriores:

		Tamaño de n		
		n=10000	n=10000	n=10000
Diferencias	Valor aproximado	6.938894e-16	4.440892e-18	0
	Tiempo transcurrido (secs)	0.1010242	2.269571	12.59601

Tabla 4. Diferencia de resultados de la estimación de π y del tiempo sin y con paralelismo.

Como se puede observar, las diferencias entre los valores aproximados de π es prácticamente 0 y la diferencia entre el tiempo es bastante grande. Además, se ve que, cuanto más grande es el tamaño de los datos, mayor es la diferencia del tiempo y menor la del valor aproximado de π.

Se puede corroborar el crecimiento de la diferencia del tiempo según su tamaño y el decrecimiento de la diferencia del valor aproximado de π según su tamaño, mediante los siguientes gráficos:

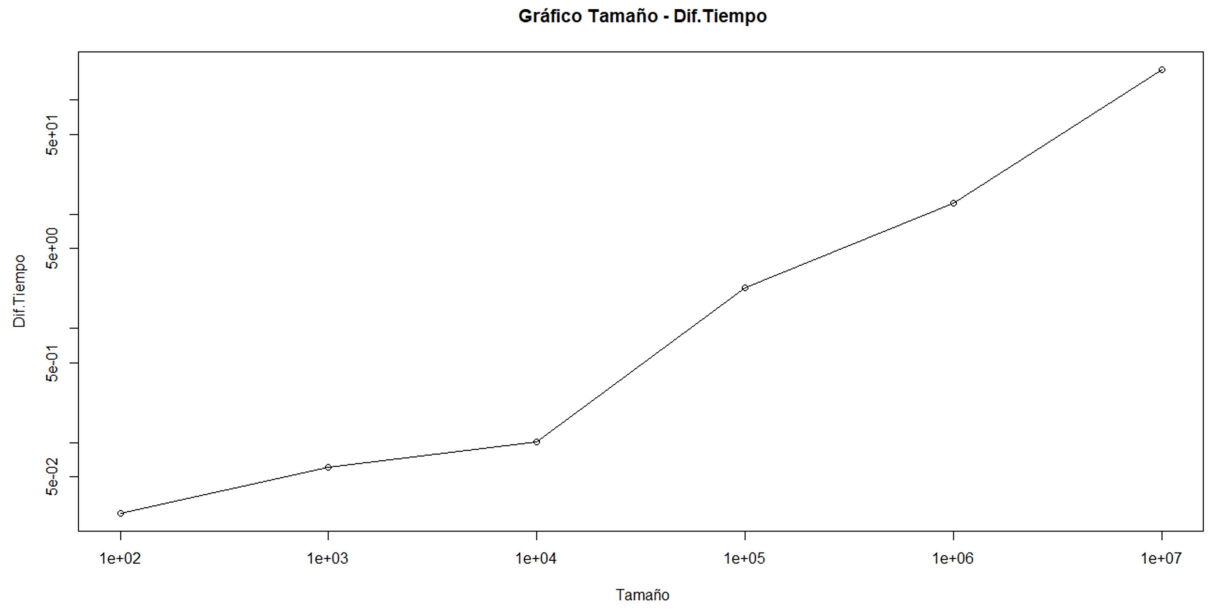

Gráfico 13. Crecimiento de la diferencia del tiempo según su tamaño.

Como se observa, este gráfico tiene la forma de una función exponencial. Lo que significa que, a medida que el valor del tamaño aumenta, también aumenta el valor de la diferencia entre los tiempos que se saca en el código.

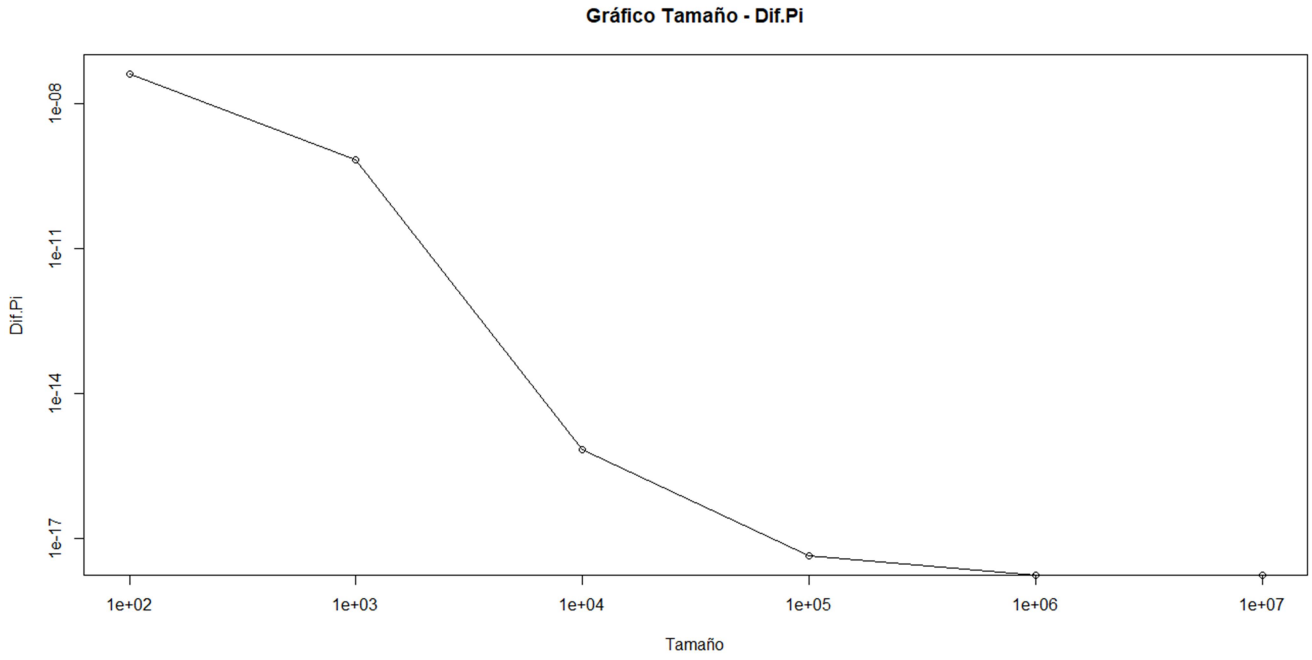

Gráfico 14. Decrecimiento de la diferencia del valor aproximado de π según su tamaño.

Sin embargo, en este gráfico se observa que tiene la forma de una función exponencial negativa. Lo que significa que, a medida que el valor del tamaño aumenta, disminuye el valor de la diferencia entre los valores aproximados de π que se saca en el código.

Ejemplo 3.

El objetivo de la función "simulacion_dados_tiradas" consiste en realizar una simulación de tiradas de dados y calcular la distribución de frecuencias de los resultados obtenidos.
La función toma dos argumentos: "num_dados" que representa el número de dados que se lanzarán en cada tirada, y "num_tiradas" que indica cuántas tiradas se realizarán en total.

Se ha ejecutado el siguiente código:

```
# Cargar el paquete 'foreach'
library(foreach)

# Cargar el paquete 'doParallel'
library(doParallel) (12)

# Configurar el backend para utilizar computación paralela
num_cores <- detectCores()
registerDoParallel(cores = num_cores)

# Crear la función para simular las n tiradas con m dados
simulacion_dados_tiradas <- function(num_dados, num_tiradas) {

    # Condicionar a que ambos argumentos sean mayores a 0
     if (num_dados <= 0 || num_tiradas <= 0) {
       stop("El número de dados y tiradas debe ser mayor que cero.")
     }

    # Iniciar tiempo de ejecución
    tiempo_inicio <- Sys.time()

    # Simular las tiradas de los dados en paralelo con un bucle "foreach"
    resultados <- foreach(i = 1:num_tiradas, .combine = rbind) %dopar% {
              sum(sample(1:6, num_dados, replace = TRUE))
    }

    # Detener tiempo de ejecución
    tiempo_fin <- Sys.time()

    # Calcular tiempo transcurrido
    tiempo <- tiempo_fin - tiempo_inicio
```

```
    # Calcular la distribución
    distribucion <- table(resultados) / num_tiradas

    # Detener el backend y volver al comportamiento secuencial
    stopImplicitCluster()

    # Liberar recursos
    gc()

    # Devolver la distribución de la simulación
    return(list(distribucion = distribucion, tiempo = tiempo))
}
# Imprimir el resultado
resultado <- simulacion_dados_tiradas(7, 100000)
print(resultado$distribucion)
print(resultado$tiempo)
```

A continuación se ejecuta la misma función pero sin usar programación paralela:

```
# Crear la función para simular las n tiradas con m dados
simulacion_dados_tiradas2 <- function(num_dados, num_tiradas) {

    # Condicionar a que ambos argumentos sean mayores a 0
     if (num_dados <= 0 || num_tiradas <= 0) {
       stop("El número de dados y tiradas debe ser mayor que cero.")
      }

    # Iniciar tiempo de ejecución
    tiempo_inicio <- Sys.time()

    #  Simular las tiradas de los dados
    resultados <- replicate(num_tiradas, sum(sample(1:6, num_dados, replace = TRUE)))

    # Detener tiempo de ejecución
    tiempo_fin <- Sys.time()

    # Calcular tiempo transcurrido
    tiempo <- tiempo_fin - tiempo_inicio
    # Calcular la distribución
    distribucion <- table(resultados) / num_tiradas

    # Devolver la distribución de la simulación
    return(list(distribucion = distribucion, tiempo = tiempo))
}
```

```
# Imprimir el resultado
resultado2 <- simulacion_dados_tiradas2(7, 100000)
print(resultado2$distribucion)
print(resultado2$tiempo)
```

A continuación se procede a comparar los resultados de los valores de la distribución y los tiempos de ejecución entre dos enfoques: uno sin paralelismo y otro con paralelismo.

Con paralelismo:

Tabla de distribución:

7	8	9	10	11	12	13	14	15
0.00002	0.00003	0.00004	0.00028	0.00065	0.00156	0.00357	0.00631	0.00957
16	17	18	19	20	21	22	23	24
0.01589	0.02334	0.0317	0.04374	0.0545	0.06609	0.07451	0.08296	0.08654
25	26	27	28	29	30	31	32	33
0.08592	0.08136	0.07471	0.06517	0.05525	0.04422	0.03193	0.02328	0.01521
34	35	36	37	38	39	40	41	42
0.00963	0.006	0.0031	0.00175	0.00082	0.00023	0.00009	0.00002	0.00001

Tabla 5. Tabla de distribución de la simulación de tiradas de dados con paralelismo

Tiempo: 2.022395 segundos

Histograma:

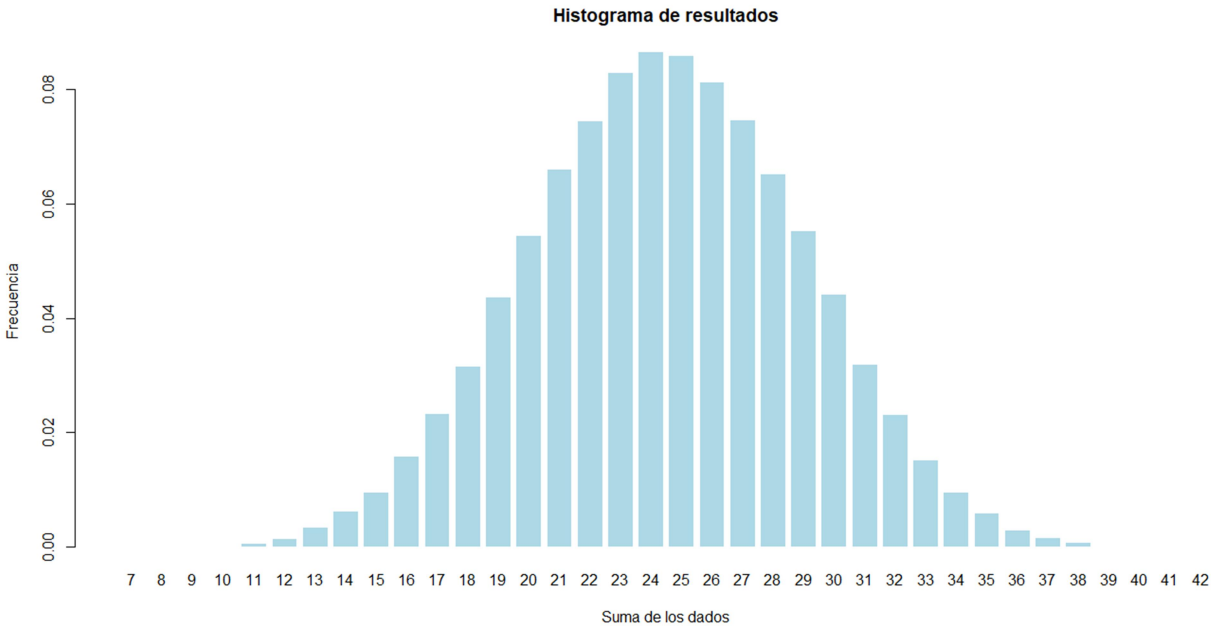

Gráfico 15. Gráfico de distribución de la simulación de tiradas de dados con paralelismo

Sin paralelismo:

Tabla de distribución:

7	8	9	10	11	12	13	14	15
0.00001	0.00003	0.00014	0.00034	0.00094	0.0017	0.00332	0.00592	0.00974
16	17	18	19	20	21	22	23	24
0.01531	0.02361	0.03253	0.04269	0.056	0.06591	0.07577	0.08225	0.08497
25	26	27	28	29	30	31	32	33
0.08634	0.08253	0.07414	0.06688	0.054	0.04368	0.03158	0.02217	0.01575
34	35	36	37	38	39	40	41	42
0.00956	0.006	0.00345	0.00154	0.00072	0.00033	0.00011	0.00003	0.00001

Tabla 6. Tabla de distribución de la simulación de tiradas de dados sin paralelismo

Tiempo: 3.51982 segundos

Histograma:

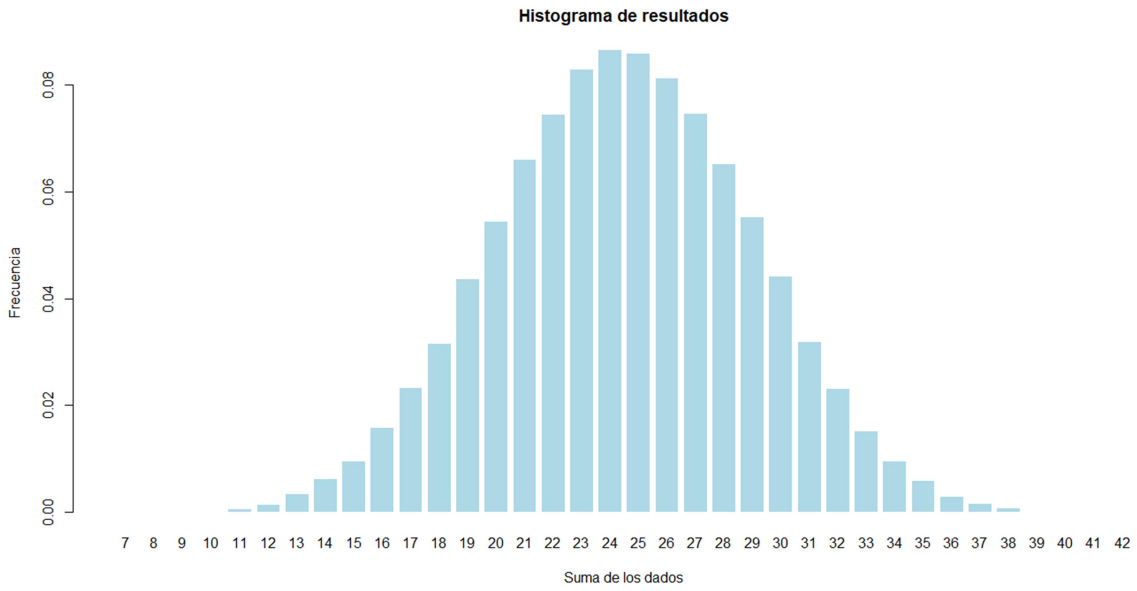

Gráfico 16. Gráfico de distribución de la simulación de tiradas de dados sin paralelismo

Se puede observar que la tabla de distribución tiene valores similares en ambos tipos de métodos.

Con respecto al tiempo, se puede ver que con paralelismo es algo más rápido, y si aumentamos el número de dados o el de las tiradas, se podrá comprobar que la diferencia de tiempo es mayor.

Código de los histogramas:

```
# Crea el histograma para la distribución en paralelo
barplot(resultado$distribucion,
        main = "Histograma de resultados",
        xlab = "Suma de los dados",
        ylab = "Frecuencia",
        col = "lightblue",
        border = "white"
)
# Crea el histograma para la distribución sin ser en paralelo
barplot(resultado2$distribucion,
        main = "Histograma de resultados",
        xlab = "Suma de los dados",
        ylab = "Frecuencia",
        col = "lightblue",
        border = "white"
)
```

5. Conclusiones

El presente trabajo de fin de grado aborda de manera exitosa la temática de la estadística computacional y la computación paralela, explorando su intersección y destacando su importancia en el análisis de datos a gran escala.

En primer lugar, se ha profundizado en los fundamentos de la estadística computacional, comprendiendo su relación con la teoría estadística clásica y su evolución en el contexto de la era digital. Se ha demostrado cómo los avances en el poder computacional han permitido el desarrollo de técnicas y algoritmos más complejos, capaces de manejar grandes volúmenes de datos y extraer información relevante de ellos.

Asimismo, se ha estudiado en detalle la computación paralela y su aplicación en la estadística computacional. Mediante el uso de múltiples procesadores y sistemas distribuidos, se ha demostrado cómo es posible acelerar significativamente el procesamiento de datos y realizar análisis estadísticos de manera más eficiente. Se ha analizado la arquitectura paralela, los modelos de programación y las herramientas disponibles para implementar algoritmos estadísticos en entornos paralelos.

Además, se ha realizado una revisión exhaustiva de los principales algoritmos y métodos estadísticos que se benefician de la computación paralela. Se ha mostrado cómo técnicas como el muestreo de Montecarlo, el aprendizaje automático y la optimización pueden aprovechar el paralelismo para mejorar la precisión y reducir el tiempo de cálculo. Se han presentado ejemplos concretos de aplicación del Método de MonteCarlo en el lenguaje de programación R y se ha visto que se optimiza con la programación en paralelo.

En resumen, este trabajo ha demostrado que la combinación de la estadística computacional y la computación paralela representa un enfoque prometedor para el análisis de datos a gran escala. El uso de técnicas paralelas permite obtener resultados más rápidos y precisos, lo que resulta fundamental en el contexto actual de grandes volúmenes de datos generados por diversas fuentes. A medida que la tecnología continúa avanzando, se espera que la estadística computacional y la computación paralela desempeñen un papel cada vez más relevante en la toma de decisiones basada en datos y el desarrollo de nuevas investigaciones.

Anexos

A. Manual de instalación

- Instalación de R:
    - Ve al sitio web oficial de R en https://www.r-project.org/
    - Haz clic en el enlace de descarga correspondiente a tu sistema operativo (Windows, macOS o Linux). En nuestro caso, Windows.
    - Sigue las instrucciones de instalación proporcionadas en el sitio web de R.
    - Una vez completada la instalación, deberías tener R correctamente configurado en tu sistema.
- Instalación de librerías en R:
    - Abre R o RStudio (un entorno de desarrollo integrado muy utilizado para R). En nuestro caso, abriremos RStudio.
    - En la consola de R, puedes instalar una librería utilizando la función install.packages("nombre_del_paquete"). En este caso, utilizaremos las librerías parallel y tibble. Las instalaremos con la función install.packages("parallel").
    - R descargará las librerías desde un repositorio de paquetes y las instalará en tu sistema.
    - Una vez que la instalación esté completa, puedes cargar la librería en tu sesión de R utilizando la función library(nombre_del_paquete). En nuestro caso será library(parallel).

Algunas librerías pueden tener dependencias adicionales fuera del alcance de R, como bibliotecas de sistema o software externo. En tales casos, es posible que debas seguir las instrucciones específicas proporcionadas en la documentación de la librería para instalar esas dependencias. Además, es importante tener en cuenta que R cuenta con un amplio repositorio de paquetes mantenidos por la comunidad.

Bibliografía

1. BISHOP, C. M., *Pattern Recognition and Machine Learning,* Springer, 2006.
2. ROBERT, C. P., & CASELLA, G., *Monte Carlo Statistical Methods* (2nd ed.). Springer, 2004.
3. GRAMA, A., KARYPIS, G., KUMAR, V., & GUPTA, A., *Introduction to Parallel Computing (2nd ed.),* Addison-Wesley, 2003.
4. HASTIE, T., TIBSHIRANI, R., & FRIEDMAN, J., *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.), Springer, 2009
5. QUINN, M. J., *Parallel Computing: Theory and Practice* (2nd ed.). McGraw-Hill, 2004.
6. Bernal, F., Albarracín, C., Gaona, J., Giraldo, L., Mosquera, C., Peña, S., Torres, Y., Ovalle, J., Nieto, J., Chacón, D., Salcedo, S., Suarez, A., Cortés, D., Pinzón, J., Higuera, P., Baquero, C. (s/f). *tareas*. Recuperado de https://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#twelve
7. Bernal, F., Albarracín, C., Gaona, J., Giraldo, L., Mosquera, C., Peña, S., Torres, Y., Ovalle, J., Nieto, J., Chacón, D., Salcedo, S., Suarez, A., Cortés, D., Pinzón, J., Higuera, P., Baquero, C. (s/f). *hilos*. Recuperado de https://ferestrepoca.github.io/paradigmas-de-programacion/paralela/paralela_teoria/index.html#twelve
8. IBM. (s.f.). Simulación de Monte Carlo. Recuperado el 1 de junio de 2024, de https://www.ibm.com/es-es/topics/monte-carlo-simulation
9. Autor. (2019). Historia de Las Computadoras Paralelas. Recuperado el 1 de junio de 2024, de https://es.scribd.com/document/414002819/Historia-de-Las-Computadoras-Paralelas
10. UNIR. (s.f.). Computación paralela. Recuperado el 1 de junio de 2024, de https://www.unir.net/ingenieria/revista/computacion-paralela/
11. R Core Team (2023). _R: A Language and Environment for Statistical Computing_. R Foundation for Statistical Computing, Vienna, Austria. <https://www.R-project.org/>.
12. Corporation M, Weston S (2022). _doParallel: Foreach Parallel Adaptor for the 'parallel' Package_. R package version 1.0.17, <https://CRAN.R-project.org/package=doParallel>.