UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Análisis del impacto de la paralelización en el rendimiento de programas computacionales en sistemas multinúcleo

Omar Sánchez Gudiño.
MTRA EN COMPUTO APLICADO | ARQUITECTURA DE COMUNICACIONES.

11 de febrero de 2025

Resumen

Este trabajo de investigación se centra en analizar el impacto de la paralelización en el tiempo de ejecución de programas utilizando múltiples núcleos de procesamiento. Se realizaron pruebas con diferentes códigos, incluyendo cálculos matemáticos y modelos de redes neuronales profundas, para observar cómo varía el rendimiento al aumentar el número de núcleos utilizados, desde 1 hasta 16. Los experimentos fueron ejecutados en una máquina con 16 núcleos, utilizando la librería 'multiprocessing' de Python para distribuir las tareas. Los resultados mostraron que, en la mayoría de los casos, el uso de más núcleos reduce el tiempo de ejecución, aunque no siempre de manera proporcional debido a la sobrecarga de la gestión de procesos. Se midieron métricas como el tiempo de ejecución y la precisión de los modelos de redes neuronales, y los resultados fueron presentados en gráficos para facilitar su análisis. Este estudio resalta la importancia de la paralelización para mejorar la eficiencia computacional y proporciona insights para la optimización de algoritmos en sistemas multinúcleo.

ÍNDICE

I. Objetivo.

En el presente trabajo de investigación tienen como objetivo realizar el análisis y optimización del tiempo de ejecución de programas mediante el uso de paralelismo, utilizando múltiples núcleos de procesamiento (cores). El objetivo principal fue evaluar cómo la paralelización afecta el rendimiento de diferentes aplicaciones, con el fin de obtener una mejor comprensión de la relación entre la cantidad de núcleos empleados y la reducción en los tiempos de ejecución. Este trabajo es relevante en el contexto de la mejora de la eficiencia computacional y la optimización de recursos en sistemas de procesamiento multinúcleo. El proceso experimental se llevó a cabo mediante la implementación y evaluación de diversos programas que incluyen desde cálculos matemáticos simples hasta modelos de aprendizaje automático utilizando redes neuronales profundas. Cada uno de estos programas fue diseñado para ejecutarse en un solo núcleo y luego se paralelizó utilizando la librería 'multiprocessing' de Python, la cual permite dividir el trabajo entre varios núcleos de la CPU. Los experimentos se realizaron en una máquina con 16 núcleos, y se probó el rendimiento con diferentes configuraciones de paralelismo, comenzando con 1 núcleo y aumentando gradualmente hasta 16 núcleos. Para los cálculos matemáticos, se utilizó un programa de estimación de pi mediante el método de Monte Carlo, que genera una gran cantidad de operaciones que resultan ser una excelente base para medir el impacto de la paralelización. Adicionalmente, se integraron modelos de redes neuronales profundas utilizando Keras, con el objetivo de evaluar cómo se ve afectado el tiempo de entrenamiento de un modelo complejo en función del número de núcleos utilizados. Se consideraron diversas métricas de rendimiento, incluyendo el tiempo de ejecución total y la precisión de los modelos entrenados.En cuanto a la evaluación de los resultados, se observó que la paralelización efectivamente reduce el tiempo de ejecución en muchos casos, especialmente cuando el número de núcleos se incrementó. Sin embargo, en algunas situaciones, el aumento en el número de núcleos no siempre resultó en una mejora proporcional, y en ciertos casos, el rendimiento disminuyó debido a la sobrecarga asociada con la gestión de múltiples procesos. Se identificaron además algunos factores como la gestión de memoria, la eficiencia del uso de los recursos de la CPU y la capacidad del sistema para manejar múltiples tareas simultáneamente como elementos críticos en el comportamiento observado.Los resultados de los experimentos fueron plasmados en gráficos y tablas, donde se presentan las métricas de tiempo de ejecución para cada configuración de núcleos, además de la precisión obtenida en el caso de los modelos de redes neuronales. Estos resultados fueron discutidos en relación con las características de los programas evaluados y la infraestructura de hardware utilizada.Este estudio proporciona una visión detallada sobre cómo el paralelismo puede ser utilizado de manera efectiva para mejorar el rendimiento de programas complejos. Los resultados obtenidos ofrecen una base para futuras investigaciones en la optimización de algoritmos en entornos multicore, así como para la aplicación de estas técnicas en áreas como la computación científica, la inteligencia artificial y la minería de datos. Las conclusiones sugieren que, aunque la paralelización ofrece mejoras sustanciales en la eficiencia, también es importante tener en cuenta los costos asociados con la distribución del trabajo y la sincronización entre procesos para evitar sobrecargas innecesarias.

II. Introducción.

La optimización de los tiempos de ejecución en sistemas de procesamiento ha sido uno de los objetivos clave en el desarrollo de aplicaciones computacionales de alto rendimiento. En la actualidad, la mayoría de los sistemas de cómputo cuentan con arquitecturas multinúcleo, lo que permite la ejecución concurrente de múltiples procesos en paralelo. El uso de estas arquitecturas ha demostrado ser una estrategia efectiva para mejorar la eficiencia de aplicaciones que requieren grandes volúmenes de procesamiento, como simulaciones científicas, procesamiento de grandes bases de datos y modelos de aprendizaje automático, entre otros. El paralelismo en la ejecución de programas se ha convertido en una técnica fundamental para optimizar el uso de los recursos disponibles en los sistemas de procesamiento multinúcleo. Sin embargo, su efectividad no siempre es lineal, ya que la mejora en el tiempo de ejecución depende de diversos factores, tales como

la complejidad de la tarea, la sobrecarga de comunicación entre los procesos, la sincronización de los mismos y la capacidad del hardware para gestionar múltiples procesos de manera eficiente. Estos factores, en conjunto, pueden influir en la reducción del rendimiento esperado al aumentar el número de núcleos disponibles. El objetivo principal de este trabajo es evaluar cómo la utilización de múltiples núcleos impacta el rendimiento de varios programas, tanto en términos de tiempo de ejecución como de precisión en el caso de modelos complejos de aprendizaje automático. Para ello, se diseñaron y ejecutaron una serie de pruebas utilizando diferentes tipos de programas que abarcan desde tareas de cálculo intensivo, como la estimación de pi mediante el método de Monte Carlo, hasta la formación de redes neuronales profundas utilizando la librería Keras. Estos experimentos fueron realizados sobre una máquina de 16 núcleos, y en cada uno de ellos se probó el rendimiento variando la cantidad de núcleos, desde 1 hasta 16.A lo largo del estudio, se utilizaron herramientas de paralelización proporcionadas por Python, como la librería "multiprocessing", que permite distribuir de manera eficiente las tareas entre varios núcleos del procesador. Con ello, se buscó analizar cómo la distribución de las cargas de trabajo influye en la mejora del rendimiento, así como identificar los límites de la paralelización en función de los diferentes tipos de programas y configuraciones de hardware.En el caso de los modelos de redes neuronales, se prestó especial atención a la precisión de los resultados obtenidos y cómo esta se ve afectada por el número de núcleos utilizados. La hipótesis central de este trabajo es que, aunque la paralelización reduce significativamente el tiempo de ejecución de las tareas en general, existen limitaciones inherentes a la distribución de recursos y la carga de trabajo que pueden generar un rendimiento subóptimo si no se manejan correctamente.El análisis y la interpretación de los resultados obtenidos no solo proporcionan una comprensión más profunda sobre los beneficios del paralelismo en arquitecturas multinúcleo, sino que también permiten identificar las mejores prácticas para la optimización de algoritmos y programas en entornos de procesamiento paralelo. Así, este estudio tiene implicaciones directas en áreas como la computación científica, la inteligencia artificial, la simulación numérica y la minería de datos, donde el uso eficiente de los recursos computacionales es crucial.A través de este trabajo, se busca aportar una base sólida para futuras investigaciones que exploren la mejora del rendimiento en sistemas de múltiples núcleos, proporcionando tanto un análisis técnico de los métodos empleados como recomendaciones sobre cómo maximizar los beneficios de la paralelización en distintos tipos de aplicaciones.

III. Marco teórico.

I. Paralelización y Concurrencia.

La paralelización es el proceso de dividir una tarea en sub-tareas más pequeñas que pueden ejecutarse de forma simultánea, aprovechando múltiples unidades de procesamiento en un sistema. A nivel de software, esto se puede lograr mediante el uso de hilos o procesos paralelos. La concurrencia, en cambio, se refiere a la capacidad de un sistema para manejar varias tareas a la vez, pero no necesariamente en paralelo. En sistemas multinúcleo, la paralelización se convierte en una herramienta esencial para optimizar el rendimiento, al permitir la ejecución simultánea de tareas en distintos núcleos del procesador.[1]

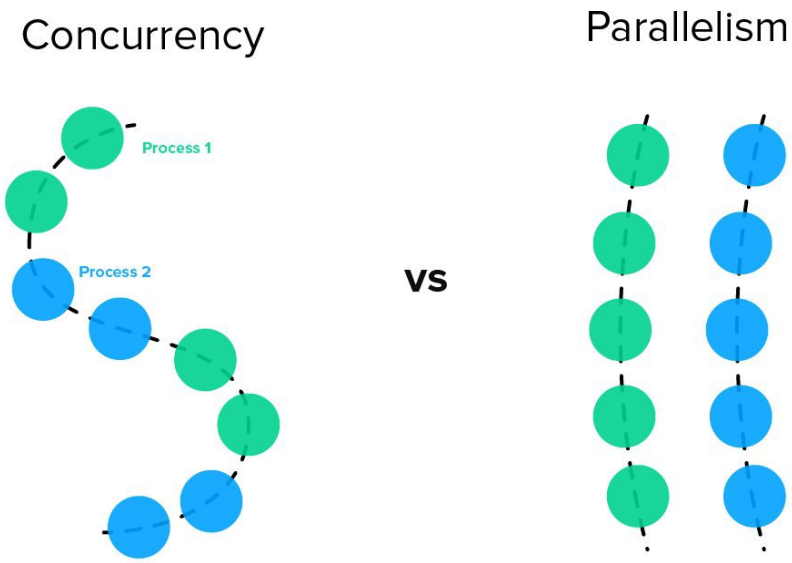

Figura 1: Concurrencia vs Paralelismo

II. ¿En que consiste el procesamiento en paralelo?

De manera general, la computación paralela consiste en el uso simultáneo de múltiples procesadores o núcleos que ejecutan cada uno una serie de instrucciones que conforman las distintas partes en las que se ha descompuesto un problema computacional para resolver. Para poder poner en marcha el cómputo en paralelo:

- El problema computacional debe dividirse en distintos componentes, trabajos o problemas que puedan ser resueltos al mismo tiempo.
- Las instrucciones de estos se deben de poder ejecutar en cualquier momento.
- Debe ser posible resolver los problemas cada vez en menos tiempo cuantos más recursos informáticos estén trabajando a la vez.

Los recursos informáticos que se utilizan en el procesamiento en paralelo son o una computadora con múltiples procesadores/núcleos o múltiples computadoras conectadas en red (computación distribuida).

III. Beneficios de usar procesamiento en paralelo.

Las ventajas principales de la computación paralela radican en que las computadoras pueden ejecutar código de manera más eficiente, lo que supone un ahorro de tiempo y dinero al clasificar el big data más rápido que nunca, además de resolver problemas más complejos.

- Modelos de computación para el mundo real: el mundo que nos rodea no es en serie. Las cosas no suceden una a la vez, esperando que termine un evento antes de que comience el siguiente. Para calcular datos sobre el clima, el tráfico, las finanzas, la industria, la agricultura, los océanos, los casquetes polares y la atención médica es necesaria la computación paralela.

- Ahorrar tiempo: la computación en serie obliga a los procesadores rápidos a hacer cosas de manera ineficiente.
- Ahorrar dinero: al ahorrar tiempo, la computación paralela abarata las cosas. El uso más eficiente de los recursos puede parecer insignificante a pequeña escala, pero cuando se amplía la visión a un sistema a miles de millones de operaciones (software bancario, por ejemplo) se consiguen enormes ahorros de costes.
- Resolución de problemas complejos o grandes: la informática está evolucionando. Con la Inteligencia Artificial (IA) y el big data una aplicación web puede procesar millones de transacciones por segundo. Además, los "grandes desafíos" como asegurar el ciberespacio o hacer que la energía solar sea asequible requerirán petaFLOPS de recursos informáticos. Esto solo es posible con la computación paralela.
- Aprovechar los recursos: los seres humanos crean 2,5 quintillones de bytes de información al día. Con el procesamiento paralelo, varias computadoras con varios núcleos cada una pueden examinar muchas veces más datos en tiempo real.[4]

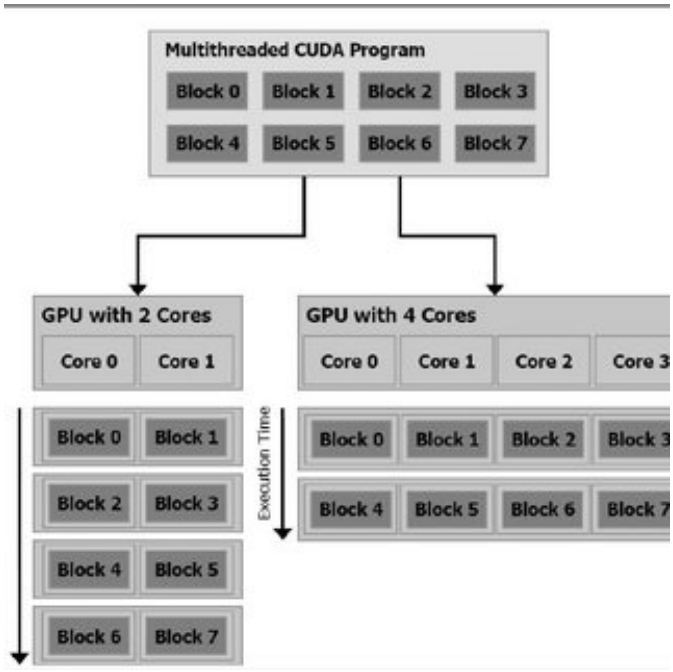

Figura 2: Paralelización multicore.

IV. Multiprocessing.

multiprocessing es un paquete que permite crear procesos (spawning) utilizando una API similar al módulo threading. El paquete multiprocessing ofrece concurrencia tanto local como remota, esquivando el Global Interpreter Lock mediante el uso de subprocesos en lugar de hilos (threads). Debido a esto, el módulo multiprocessing le permite al programador aprovechar al máximo múltiples procesadores en una máquina determinada. Se ejecuta tanto en Unix como en Windows.

El módulo multiprocessing también introduce API que no tienen análogos en el módulo threading. Un buen ejemplo de esto es el objeto Pool que ofrece un medio conveniente de paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos de entrada a través de procesos (paralelismo de datos).[2]

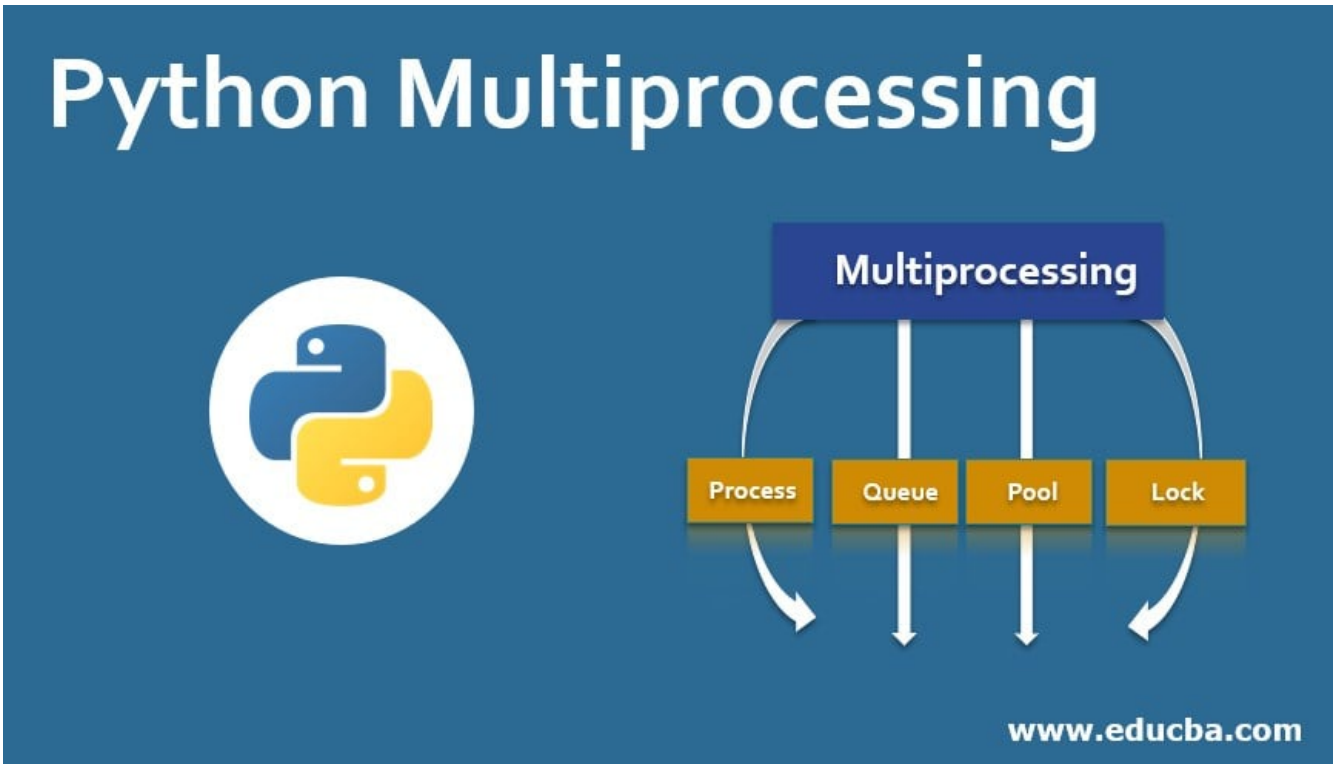

Figura 3: Multiprocessing en python.

V. Concurrent.futures.

Es un módulo de la biblioteca estándar de Python (disponible desde la versión 3.2) y que facilita la ejecución asíncrona y concurrente de funciones. Este módulo proporciona una interfaz de alto nivel para lanzar tareas de forma paralela usando dos clases principales:

- ThreadPoolExecutor: Utiliza un grupo (pool) de hilos y es adecuado para tareas I/O-bound (por ejemplo, operaciones de red o de disco) en las que gran parte del tiempo se espera la respuesta de algún recurso externo.
- ProcessPoolExecutor: Utiliza un grupo de procesos, lo que permite aprovechar múltiples núcleos de la CPU (útil para tareas CPU-bound). Al usar procesos, se evita el Global Interpreter Lock (GIL) que limita la ejecución paralela de hilos en Python.

Ambos ejecutores implementan una interfaz común definida por la clase abstracta Executor y generan objetos del tipo Future. Un Future es una representación del resultado de una operación que se está ejecutando en segundo plano. Con él, puedes consultar si la tarea ya terminó, obtener su resultado (o la excepción que haya podido generar) y, en general, gestionar la ejecución asíncrona de manera sencilla.[5]

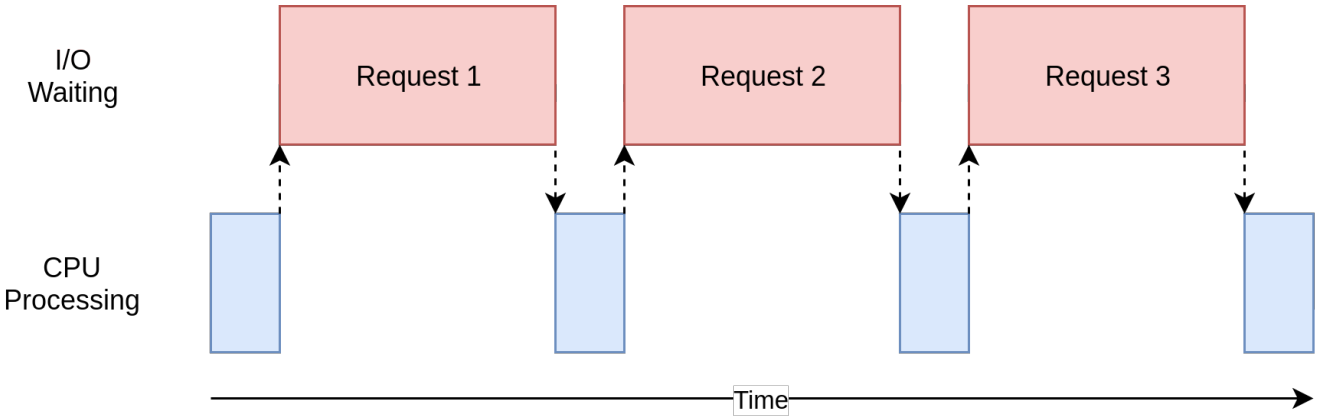

Figura 4: Funcionamiento de librerias para paralelizar.

VI. Redes neuronales.

Una red neuronal es un programa o modelo de aprendizaje automático que toma decisiones de manera similar al cerebro humano, mediante el uso de procesos que imitan la forma en que las neuronas biológicas trabajan juntas para identificar fenómenos, sopesar opciones y llegar a conclusiones. Cada red neuronal consta de capas de nodos o neuronas artificiales: una capa de entrada, una o más capas ocultas y una capa de salida. Cada nodo se conecta a otros y tiene su propia ponderación y umbral asociados. Si la salida de cualquier nodo individual está por encima del valor del umbral especificado, ese nodo se activa y envía datos a la siguiente capa de la red. De lo contrario, no se pasa ningún dato a la siguiente capa de la red. Las redes neuronales se basan en datos de entrenamiento para aprender y mejorar su precisión con el tiempo. Una vez que se ajustan para obtener precisión, son herramientas poderosas en informática e inteligencia artificial, lo que nos permite clasificar y agrupar datos a alta velocidad. Las tareas de reconocimiento de voz o reconocimiento de imágenes pueden tardar minutos en lugar de horas en comparación con la identificación manual por parte de expertos humanos.[3]

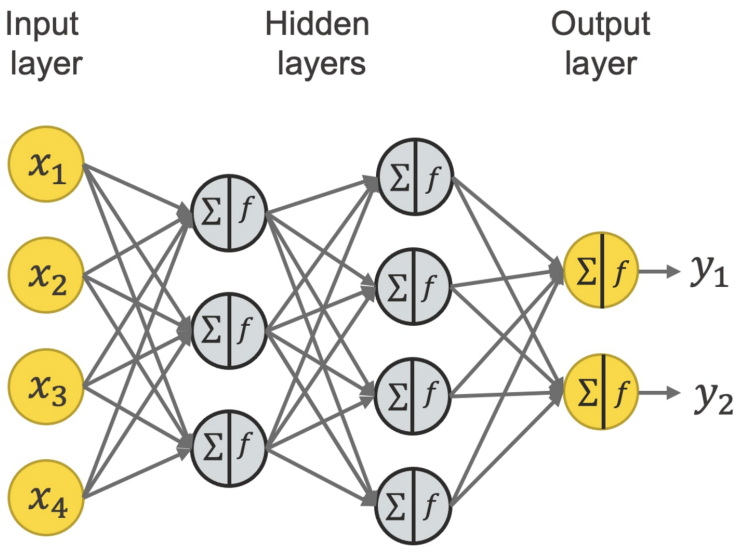

Figura 5: Modelo de una red neuronal

IV. Desarrollo.

Para poder realizar esta comparativa, tuve que crear primero diferentes programas, los cuales, computacionalmente hablando, requirieran de grandes cantidades de recursos. Esto, para que fuera visible el funcionamiento de los hilos, en total se generaron 5 algoritmos diferentes.

I. Algotimo para redes neuronales profundas.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from multiprocessing import Pool
import matplotlib.pyplot as plt
import os

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

def create_model():
    model = models.Sequential([
        layers.Input(shape=(28, 28)),
        layers.Flatten(),
        layers.Dense(128, activation='relu'),
        layers.Dropout(0.2),
        layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

def train_model(start_idx, end_idx, x_train, y_train):
    model = create_model()
    model.fit(x_train[start_idx:end_idx], y_train[start_idx:end_idx], epochs=50, verbose=0)
    accuracy = model.evaluate(x_test, y_test, verbose=0)
    return accuracy[1]

def parallel_training(num_processes, x_train, y_train):
    pool = Pool(processes=num_processes)
    chunk_size = len(x_train) // num_processes
    result = pool.starmap(train_model, [(i * chunk_size, (i + 1) * chunk_size, x_train, y_train) f
    pool.close()
    pool.join()
    return result

def run_parallel_experiments():
    num_cores = os.cpu_count()
    times = []
```

```
    accuracies = []

    for num_processes in range(1, num_cores + 1):
        print(f"Ejecutando con {num_processes} núcleos...")
        import time
        start_time = time.time()
        accuracies_for_run = parallel_training(num_processes, x_train, y_train)
        end_time = time.time()
        execution_time = end_time - start_time
        avg_accuracy = np.mean(accuracies_for_run)
        print(f"Tiempo de ejecución con {num_processes} núcleos: {execution_time:.4f} s")
        print(f"Precisión promedio con {num_processes} núcleos: {avg_accuracy:.4f}")
        times.append(execution_time)
        accuracies.append(avg_accuracy)
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, num_cores + 1), times, marker='o', linestyle='-', color='b', label='Tiempo d
    plt.xlabel('Número de núcleos')
    plt.ylabel('Tiempo de ejecución (segundos)')
    plt.title('Tiempo de Ejecución vs. Número de Núcleos')
    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == '__main__':
    run_parallel_experiments()
```

El código implementa una red neuronal simple para la clasificación de imágenes del conjunto de datos MNIST. El entrenamiento se paraleliza dividiendo el conjunto de datos en subconjuntos, cada uno asignado a un núcleo.

I.1. Carga de Datos

El conjunto de datos MNIST se carga y normaliza:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

I.2. Definición del Modelo

Se define una red neuronal con las siguientes características:

- Capa de entrada con imágenes de 28x28 píxeles.
- Capa oculta con 128 neuronas y activación ReLU.
- Capa de salida con 10 neuronas y activación softmax.
- Optimización con Adam y función de pérdida de entropía cruzada categórica.

I.3. Entrenamiento Paralelizado

El entrenamiento se distribuye entre múltiples núcleos:

- Se divide el conjunto de entrenamiento en subconjuntos iguales.
- Cada proceso entrena el modelo en su subconjunto.
- Se utiliza `Pool.starmap()` para ejecutar el entrenamiento en paralelo.

I.4. Ejecución del Código

Para cada número de núcleos, el código:

- Mide el tiempo de ejecución.
- Calcula la precisión promedio obtenida.
- Muestra los resultados en consola.

I.5. Salida en Consola

Para cada cantidad de núcleos, se imprime:

```
Tiempo de ejecución con X núcleos: Y segundos
Precisión promedio con X núcleos: Z
```

I.6. Gráfica de Rendimiento

Se genera una gráfica donde:

- El eje X representa el número de núcleos utilizados.
- El eje Y representa el tiempo de ejecución en segundos.

II. Algoritmo Montecarlo.

```
1  import time
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import os
5  from multiprocessing import Pool
6
7  def monte_carlo_pi(num_samples):
8      inside_circle = 0
9      for _ in range(num_samples):
10         x, y = np.random.random(), np.random.random()
11         if x**2 + y**2 <= 1:
12             inside_circle += 1
13     return inside_circle
14
15 def parallel_monte_carlo_pi(total_samples, num_cores):
16     samples_per_core = total_samples // num_cores
```

```
    with Pool(processes=num_cores) as pool:
        results = pool.map(monte_carlo_pi, [samples_per_core] * num_cores)
    total_inside_circle = sum(results)
    return (4 * total_inside_circle) / total_samples

total_samples = 10**8
execution_times = []
num_cores = os.cpu_count()

for num_processes in range(1, num_cores + 1):
    start_time = time.time()
    pi_approximation = parallel_monte_carlo_pi(total_samples, num_processes)
    end_time = time.time()
    execution_time = end_time - start_time
    execution_times.append(execution_time)
    print(f"Tiempo de ejecución con {num_processes} núcleos: {execution_time:.4f} s")
    print(f"Aproximación de Pi con {num_processes} núcleos: {pi_approximation:.6f}")

plt.plot(range(1, num_cores + 1), execution_times, marker='o')
plt.xlabel('Número de núcleos')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title('Simulación de Monte Carlo para aproximar Pi')
plt.grid(True)
plt.show()
```

II.1. Explicación del código Algoritmo Montecarlo.

- Genera `100000000` puntos aleatorios (x, y) en el intervalo $[0, 1] \times [0, 1]$.
- Verifica cuántos de estos puntos caen dentro del cuarto de círculo de radio 1, usando la ecuación:

$$x^2 + y^2 \leq 1$$

- Devuelve la cantidad de puntos que están dentro del círculo.

II.2. Función en paralelo

- Divide el número total de muestras entre los núcleos disponibles (`16`).
- Usa `multiprocessing.Pool` para distribuir la tarea en varios procesos.
- La función `pool.map()` ejecuta `monte_carlo_pi` en paralelo, donde cada proceso recibe `samples_per_core` puntos.
- Se suman los puntos dentro del círculo y se estima el valor de π con la fórmula:

$$\pi \approx 4 \times \frac{\text{total\_inside\_circle}}{\text{total\_samples}}$$

- Finalmente, devuelve la estimación de π.

II.3. Bucle Principal

- Se define `total_samples` como 10^8, que es el número de puntos usados en la simulación.
- Se itera desde 1 hasta el número total de núcleos disponibles (`16`).
- Se mide el tiempo de ejecución con cada cantidad de núcleos.
- Se muestra en consola el tiempo y la aproximación de π obtenida.

II.4. Gráfica de Rendimiento

Para visualizar el impacto de la paralelización en el rendimiento, se genera una gráfica donde:

- El eje X representa el número de núcleos utilizados.
- El eje Y representa el tiempo de ejecución en segundos.

Esto permite analizar cómo el uso de múltiples núcleos mejora el rendimiento en tareas intensivas de cómputo.

III. Algoritmo Fibonancci

```
import time
import matplotlib.pyplot as plt
import os
from multiprocessing import Pool

def fibonacci(n, memo={}):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]

def fibonacci_paralelizado(n, num_cores):
    with Pool(processes=num_cores) as pool:
        resultados = pool.map(fibonacci, range(n-7, n+7))
    return resultados

n = 950
execution_times = []
fib_results = []
num_cores = os.cpu_count()
print('El numero de cores en tu computadora es: ' + str(num_cores))
for num_processes in range(1, num_cores + 1):
    start_time = time.time()
    resultados = fibonacci_paralelizado(n, num_processes)
    end_time = time.time()
    execution_time = end_time - start_time
    execution_times.append(execution_time)
```

```
    fib_results.append(resultados)

for i in range(1, num_cores + 1):
    print(f"\nResultados con {i} núcleos:")
    print(fib_results[i-1])

for i in range(1, num_cores + 1):
    print(f"\nTiempo de ejecución con {i} núcleos: {execution_times[i-1]:.4f} s")
plt.plot(range(1, num_cores + 1), execution_times, marker='o')
plt.xlabel('Número de núcleos')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title(f'Tiempo de ejecución de Fibonacci({n}) con diferentes números de núcleos')
plt.grid(True)
plt.show()
```

III.1. Función Fibonacci

III.2. Explicación del Fibonacci.

- Implementa una versión recursiva con memorización para calcular el número de Fibonacci en la posición `n`.
- Si `n` ya ha sido calculado, el valor se obtiene del diccionario `memo`, evitando cálculos redundantes.
- La función sigue la relación de recurrencia estándar:

$$F(n) = F(n-1) + F(n-2)$$

- Los casos base son:

$$F(0) = 0, \quad F(1) = 1$$

III.3. Función `fibonacci_paralelizado(n, num_cores)`

- Utiliza `multiprocessing.Pool` para distribuir el cálculo de Fibonacci en un rango de valores desde $n-7$ hasta $n+7$.
- La función `pool.map()` ejecuta la función `fibonacci()` en paralelo.
- Retorna una lista con los valores de Fibonacci calculados.

III.4. Bucle Principal

- Se define `n = 950`, que es el índice del número de Fibonacci a calcular.
- Se mide el tiempo de ejecución al calcular Fibonacci para el rango definido con diferentes cantidades de núcleos.
- Los resultados se almacenan y se muestran en consola.

III.5. Gráfica de Rendimiento

Para visualizar el impacto de la paralelización en el rendimiento, se genera una gráfica donde:

- El eje X representa el número de núcleos utilizados.
- El eje Y representa el tiempo de ejecución en segundos.

Esto permite analizar cómo el uso de múltiples núcleos mejora el rendimiento en tareas intensivas de cómputo.

IV. Algoritmo Contar números primos

```
import time
import matplotlib.pyplot as plt
import os
import math
from multiprocessing import Pool

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def calculate_primes(start, end):
    primes = [n for n in range(start, end) if is_prime(n)]
    return primes

def prime_calculation_parallel(limit, num_cores):
    chunk_size = limit // num_cores
    ranges = [(i * chunk_size, (i + 1) * chunk_size) for i in range(num_cores)]
    with Pool(processes=num_cores) as pool:
        result = pool.starmap(calculate_primes, ranges)
    primes = [prime for sublist in result for prime in sublist]
    return primes

limit = 10**7
execution_times = []
num_cores = os.cpu_count()
for num_processes in range(1, num_cores + 1):
    start_time = time.time()
    primes = prime_calculation_parallel(limit, num_processes)
    end_time = time.time()
    execution_time = end_time - start_time
    execution_times.append(execution_time)
    print(f"Tiempo de ejecución con {num_processes} núcleos: {execution_time:.4f} s")

```

```
for i in range(1, num_cores + 1):
    print(f"\nResultados con {i} núcleos: {len(primes)} números primos encontrados")
plt.plot(range(1, num_cores + 1), execution_times, marker='o')
plt.xlabel('Número de núcleos')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title(f'Tiempo de ejecución con diferentes núcleos para cálculo de primos hasta {limit}')
plt.grid(True)
plt.show()
```

IV.1. Explicación del código para calcular el numero de numeros primos.

IV.2. Función `is_prime(n)`

- Determina si un número `n` es primo.
- Se utiliza un bucle hasta $\sqrt{n}$ para verificar si es divisible por algún número menor.
- Si es divisible, se retorna `False`, de lo contrario, `True`.

IV.3. Función `calculate_primes(start, end)`

- Genera una lista de números primos en el rango $[start, end)$.
- Utiliza la función `is_prime(n)` para filtrar los números primos.

IV.4. Función `prime_calculation_parallel(limit, num_cores)`

- Divide el rango $[0, \texttt{limit})$ en subrangos para cada núcleo.
- Utiliza `multiprocessing.Pool` para distribuir la carga de trabajo.
- La función `pool.starmap()` se usa para calcular primos en paralelo.
- Retorna una lista combinada con todos los números primos encontrados.

IV.5. Ejecución del Código

- Se establece un límite de búsqueda de 10^7.
- Se ejecuta el cálculo de números primos con diferentes cantidades de núcleos.
- Se mide el tiempo de ejecución y se almacena para su análisis.

IV.6. Salida en Consola

Para cada número de núcleos, el código imprime:

- El tiempo de ejecución en segundos.
- La cantidad de números primos encontrados.

IV.7. Gráfica de Rendimiento

Se genera una gráfica donde:

- El eje X representa el número de núcleos utilizados.
- El eje Y representa el tiempo de ejecución en segundos.

Esto permite analizar cómo la paralelización mejora el rendimiento.

V. Algoritmo multiplicación de matrices

```python
import time
import numpy as np
import matplotlib.pyplot as plt
import os
from multiprocessing import Pool

def multiply_matrices(A, B):
    return np.dot(A, B)

def parallel_matrix_multiplication(A, B, num_cores):
    chunk_size = A.shape[0] // num_cores
    ranges = [(i * chunk_size, (i + 1) * chunk_size) for i in range(num_cores)]

    with Pool(processes=num_cores) as pool:
        result = pool.starmap(multiply_chunk, [(A, B, start, end) for start, end in ranges])
    return np.vstack(result)

def multiply_chunk(A, B, start_row, end_row):
    return np.dot(A[start_row:end_row, :], B)

A = np.random.rand(10000, 10000)
B = np.random.rand(10000, 10000)
execution_times = []
num_cores = os.cpu_count()

for num_processes in range(1, num_cores + 1):
    start_time = time.time()
    result = parallel_matrix_multiplication(A, B, num_processes)
    end_time = time.time()
    execution_time = end_time - start_time
    execution_times.append(execution_time)
    print(f"Tiempo de ejecución con {num_processes} núcleos: {execution_time:.4f} s")

plt.plot(range(1, num_cores + 1), execution_times, marker='o')
plt.xlabel('Número de núcleos')
plt.ylabel('Tiempo de ejecución (segundos)')
plt.title('Tiempo de ejecución para multiplicación de matrices con diferentes núcleos')
```

```
plt.grid(True)
plt.show()
```

V.1. Explicación del código multiplicación matricial.

V.2. Función `multiply_matrices(A, B)`

- Realiza la multiplicación de dos matrices A y B utilizando la función `np.dot()` de NumPy.

V.3. Función `multiply_chunk(A, B, start_row, end_row)`

- Realiza la multiplicación parcial de una submatriz de A con la matriz B.
- Se usa para dividir la carga de trabajo entre múltiples núcleos.

V.4. Función `parallel_matrix_multiplication(A, B, num_cores)`

- Divide la matriz A en segmentos para ser procesados en paralelo.
- Se utiliza `multiprocessing.Pool` para distribuir la carga entre los núcleos.
- Se combinan los resultados parciales con `np.vstack()` para obtener el producto final.

V.5. Ejecución del Código

- Se generan dos matrices aleatorias de 10000×10000.
- Se ejecuta la multiplicación con diferentes cantidades de núcleos.
- Se mide el tiempo de ejecución para cada configuración.

V.6. Salida en Consola

Para cada número de núcleos, el código imprime:

- El tiempo de ejecución en segundos.

V.7. Gráfica de Rendimiento

Se genera una gráfica donde:

- El eje X representa el número de núcleos utilizados.
- El eje Y representa el tiempo de ejecución en segundos.

VI. Arquitectura del hardware.

Una vez que todos los programas fueron generados, lo que realizamos fue un corrimiento de estos programas en la computadora designada. La computadora usada fue una HUAWEI MateBook D 14 Intel , dicha computadora cuenta con las siguientes características:

- Procesador: Intel Core i7-1195G7 de 13 generación
- Graficos: Intel Iris X Graphics

- Memoria RAM: 16 GB DDR4
- Almacenamiento: 1Tb SSD.

VII. Experimentos realizados.

Para profundizar en el análisis, se ejecutaron los programas en la misma computadora utilizando dos sistemas operativos distintos: Windows 11 y Linux Elementary OS, esto para conocer si el sistema operativo afecta de alguna manera el rendimiento. Además, se diseñó un experimento en el que el programa se ejecutaba iterativamente, incrementando el número de núcleos utilizados en cada ciclo hasta alcanzar la totalidad de los núcleos disponibles en el sistema. Por otro lado, y en busca de realizar de manera más robusta el experimento, generamos dos programas con el mismo algoritmo; sin embargo, usando un método/librería de paralelización diferente, eso solamente con la intención de conocer cuál de las dos librerías era mejor entre sí para realizar paralelizaciones.

V. Resultados.

I. Resultado Algoritmo para calcular Numero primos.

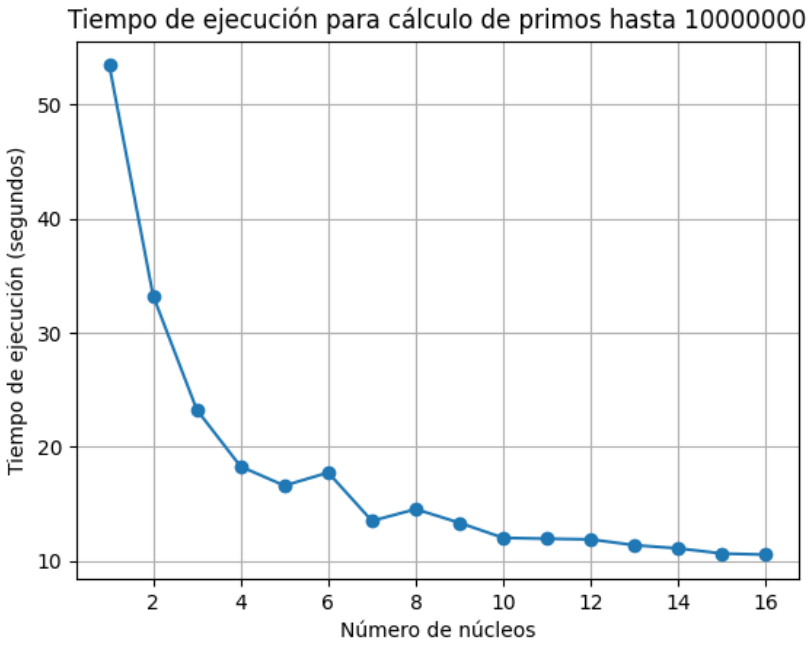

Figura 6: Rendimiento del calculo de números primos comparando cada núcleo usado en Linux

I.1. Impresión en consola (Linux)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 53.4508 segundos
Tiempo de ejecución con 2 núcleos: 33.2690 segundos
Tiempo de ejecución con 3 núcleos: 23.2483 segundos
Tiempo de ejecución con 4 núcleos: 18.2870 segundos
Tiempo de ejecución con 5 núcleos: 16.6136 segundos
Tiempo de ejecución con 6 núcleos: 17.7568 segundos
Tiempo de ejecución con 7 núcleos: 13.5200 segundos

Tiempo de ejecución con 8 núcleos: 14.5507 segundos
Tiempo de ejecución con 9 núcleos: 13.3521 segundos
Tiempo de ejecución con 10 núcleos: 12.0335 segundos
Tiempo de ejecución con 11 núcleos: 11.9579 segundos
Tiempo de ejecución con 12 núcleos: 11.8999 segundos
Tiempo de ejecución con 13 núcleos: 11.3913 segundos
Tiempo de ejecución con 14 núcleos: 11.1126 segundos
Tiempo de ejecución con 15 núcleos: 10.6599 segundos
Tiempo de ejecución con 16 núcleos: 10.5629 segundos

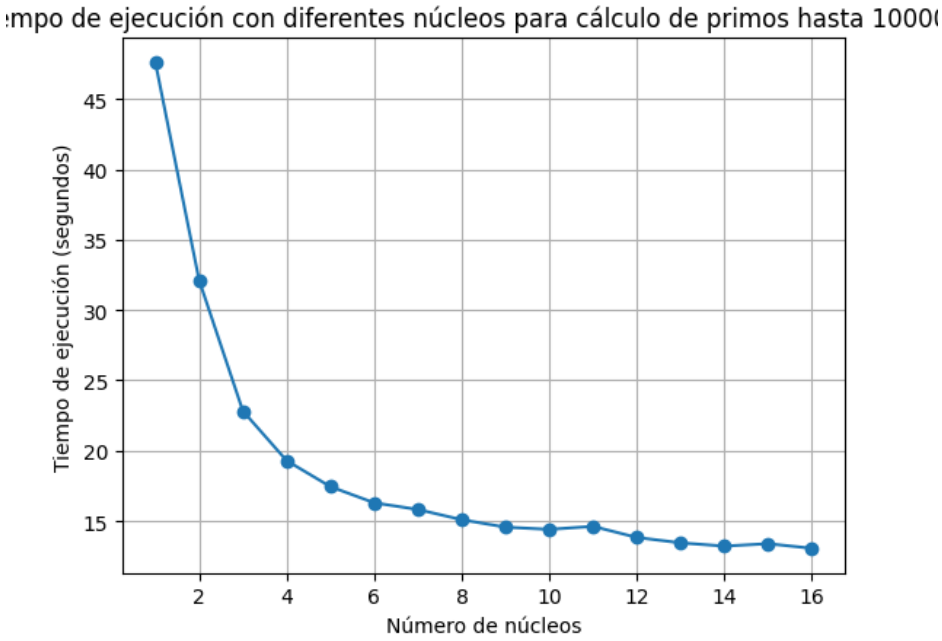

Figura 7: Rendimiento del calculo de números primos comparando cada núcleo usado en Linux

I.2. Impresión en consola (Windows)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 47.6324 segundos
Tiempo de ejecución con 2 núcleos: 32.1304 segundos
Tiempo de ejecución con 3 núcleos: 22.8205 segundos
Tiempo de ejecución con 4 núcleos: 19.2882 segundos
Tiempo de ejecución con 5 núcleos: 17.4389 segundos
Tiempo de ejecución con 6 núcleos: 16.2901 segundos
Tiempo de ejecución con 7 núcleos: 15.8125 segundos
Tiempo de ejecución con 8 núcleos: 15.0867 segundos
Tiempo de ejecución con 9 núcleos: 14.5573 segundos
Tiempo de ejecución con 10 núcleos: 14.4007 segundos
Tiempo de ejecución con 11 núcleos: 14.6142 segundos
Tiempo de ejecución con 12 núcleos: 13.8240 segundos
Tiempo de ejecución con 13 núcleos: 13.4410 segundos

Tiempo de ejecución con 14 núcleos: 13.2113 segundos
Tiempo de ejecución con 15 núcleos: 13.3802 segundos
Tiempo de ejecución con 16 núcleos: 13.0534 segundos

II. Comparativa.

Ambos conjuntos de datos demuestran que la paralelización contribuye a reducir significativamente el tiempo de ejecución. En Linux, el paso de 1 a 16 núcleos permite disminuir el tiempo de ejecución de aproximadamente 53 a 10,5 segundos, mientras que en Windows se observa una reducción de cerca de 48 a 13 segundos. No obstante, se aprecia que los beneficios de agregar más núcleos se hacen menos notorios al superar cierto límite, ya que la mejora en los tiempos se vuelve marginal y en algunos casos incluso se registran pequeñas variaciones o aumentos puntuales (como el comportamiento atípico en Linux al pasar de 5 a 6 núcleos). Además, la comparación sugiere que Linux aprovecha de manera más eficiente el paralelismo en este escenario. En resumen, aunque la paralelización mejora el rendimiento en ambos sistemas, es importante identificar el punto de rendimiento óptimo para evitar sobrecargas o rendimientos decrecientes.

III. Resultados Algoritmo Monte Carlo

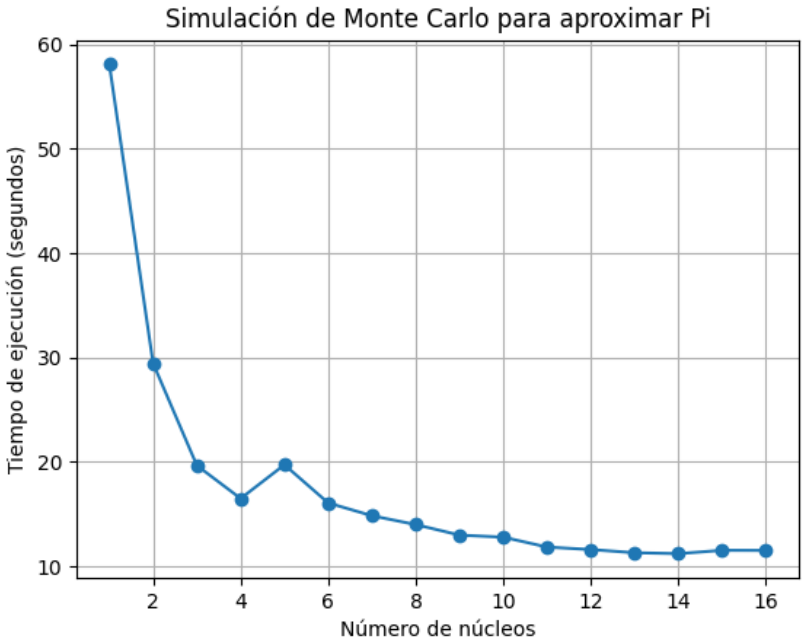

Figura 8: Comparación del tiempo para calcular pi usando el algoritmo Monte Carlo en Linux

III.1. Impresión en consola (Linux)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 58.0768 segundos
Aproximación de Pi con 1 núcleos: 3.141416
Tiempo de ejecución con 2 núcleos: 29.4447 segundos
Aproximación de Pi con 2 núcleos: 3.141381

Tiempo de ejecución con 3 núcleos: 19.6603 segundos
Aproximación de Pi con 3 núcleos: 3.141545
Tiempo de ejecución con 4 núcleos: 16.4982 segundos
Aproximación de Pi con 4 núcleos: 3.141694
Tiempo de ejecución con 5 núcleos: 19.7306 segundos
Aproximación de Pi con 5 núcleos: 3.141849
Tiempo de ejecución con 6 núcleos: 16.0735 segundos
Aproximación de Pi con 6 núcleos: 3.141446
Tiempo de ejecución con 7 núcleos: 14.8484 segundos
Aproximación de Pi con 7 núcleos: 3.141390
Tiempo de ejecución con 8 núcleos: 14.0005 segundos
Aproximación de Pi con 8 núcleos: 3.141805
Tiempo de ejecución con 9 núcleos: 12.9937 segundos
Aproximación de Pi con 9 núcleos: 3.141626
Tiempo de ejecución con 10 núcleos: 12.7966 segundos
Aproximación de Pi con 10 núcleos: 3.141479
Tiempo de ejecución con 11 núcleos: 11.8475 segundos
Aproximación de Pi con 11 núcleos: 3.141673
Tiempo de ejecución con 12 núcleos: 11.6100 segundos
Aproximación de Pi con 12 núcleos: 3.141438
Tiempo de ejecución con 13 núcleos: 11.3054 segundos
Aproximación de Pi con 13 núcleos: 3.141669
Tiempo de ejecución con 14 núcleos: 11.2183 segundos
Aproximación de Pi con 14 núcleos: 3.141370
Tiempo de ejecución con 15 núcleos: 11.5363 segundos
Aproximación de Pi con 15 núcleos: 3.141997
Tiempo de ejecución con 16 núcleos: 11.5257 segundos
Aproximación de Pi con 16 núcleos: 3.141384

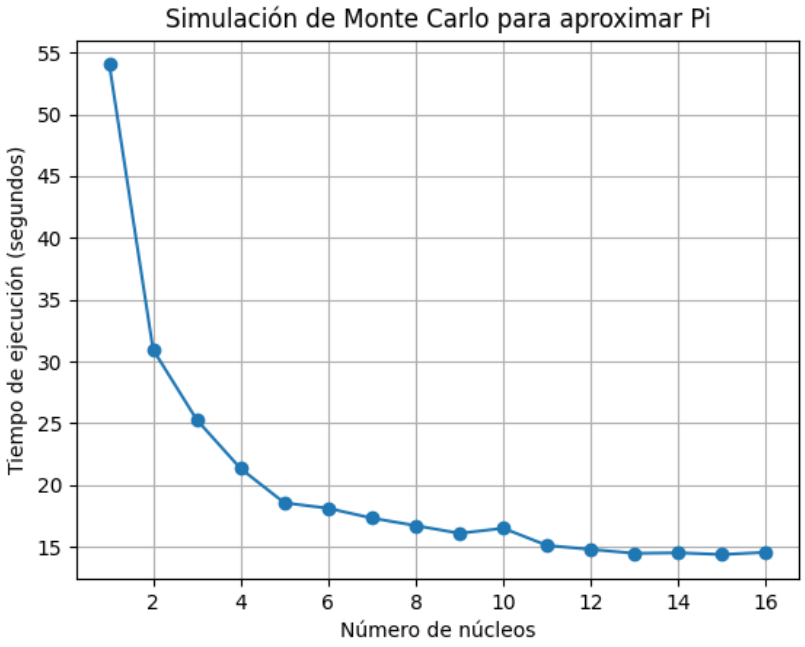

Figura 9: Comparación del tiempo para calcular pi usando el algoritmo Monte Carlo en Windows

III.2. Impresión en consola (Windows)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 54.0423 segundos
Aproximación de Pi con 1 núcleos: 3.141679
Tiempo de ejecución con 2 núcleos: 30.9546 segundos
Aproximación de Pi con 2 núcleos: 3.141500
Tiempo de ejecución con 3 núcleos: 25.2651 segundos
Aproximación de Pi con 3 núcleos: 3.141381
Tiempo de ejecución con 4 núcleos: 21.3261 segundos
Aproximación de Pi con 4 núcleos: 3.141396
Tiempo de ejecución con 5 núcleos: 18.5514 segundos
Aproximación de Pi con 5 núcleos: 3.141467
Tiempo de ejecución con 6 núcleos: 18.1128 segundos
Aproximación de Pi con 6 núcleos: 3.141561
Tiempo de ejecución con 7 núcleos: 17.3164 segundos
Aproximación de Pi con 7 núcleos: 3.141565
Tiempo de ejecución con 8 núcleos: 16.7103 segundos
Aproximación de Pi con 8 núcleos: 3.141696
Tiempo de ejecución con 9 núcleos: 16.0878 segundos
Aproximación de Pi con 9 núcleos: 3.141983
Tiempo de ejecución con 10 núcleos: 16.4954 segundos
Aproximación de Pi con 10 núcleos: 3.141689
Tiempo de ejecución con 11 núcleos: 15.1033 segundos
Aproximación de Pi con 11 núcleos: 3.141666
Tiempo de ejecución con 12 núcleos: 14.7838 segundos

Aproximación de Pi con 12 núcleos: 3.141545
Tiempo de ejecución con 13 núcleos: 14.4650 segundos
Aproximación de Pi con 13 núcleos: 3.141870
Tiempo de ejecución con 14 núcleos: 14.5073 segundos
Aproximación de Pi con 14 núcleos: 3.141691
Tiempo de ejecución con 15 núcleos: 14.3705 segundos
Aproximación de Pi con 15 núcleos: 3.141641
Tiempo de ejecución con 16 núcleos: 14.5483 segundos
Aproximación de Pi con 16 núcleos: 3.141335

IV. Comparativa.

Ambos conjuntos de datos evidencian que la paralelización reduce significativamente el tiempo de ejecución sin afectar la precisión del cálculo de π. En Linux, el tiempo disminuye de aproximadamente 58 segundos con un solo núcleo a cerca de 11.5 segundos con 16 núcleos, aunque se observan algunas fluctuaciones (por ejemplo, un pico de 19.73 segundos con 5 núcleos) que pueden deberse a sobrecostos de sincronización o a variaciones en la carga del sistema. En Windows, se aprecia una tendencia similar: el tiempo disminuye de 54 segundos a alrededor de 14,5 segundos al incrementar el número de núcleos, aunque la reducción es menos marcada en comparación con Linux. Por otro lado, las aproximaciones de π se mantienen prácticamente estables en ambos sistemas, oscilando en un rango muy reducido (aproximadamente entre 3,1413 y 3,1419), lo que indica que el procesamiento en paralelo no compromete la exactitud del resultado. En conclusión, la implementación paralela mejora de manera notable el rendimiento computacional en ambos entornos, siendo Linux el que obtiene tiempos de ejecución inferiores, mientras que la precisión en la aproximación de π se conserva de forma consistente independientemente del número de núcleos empleados.

V. Resultados Algoritmo Multiplicación matricial.

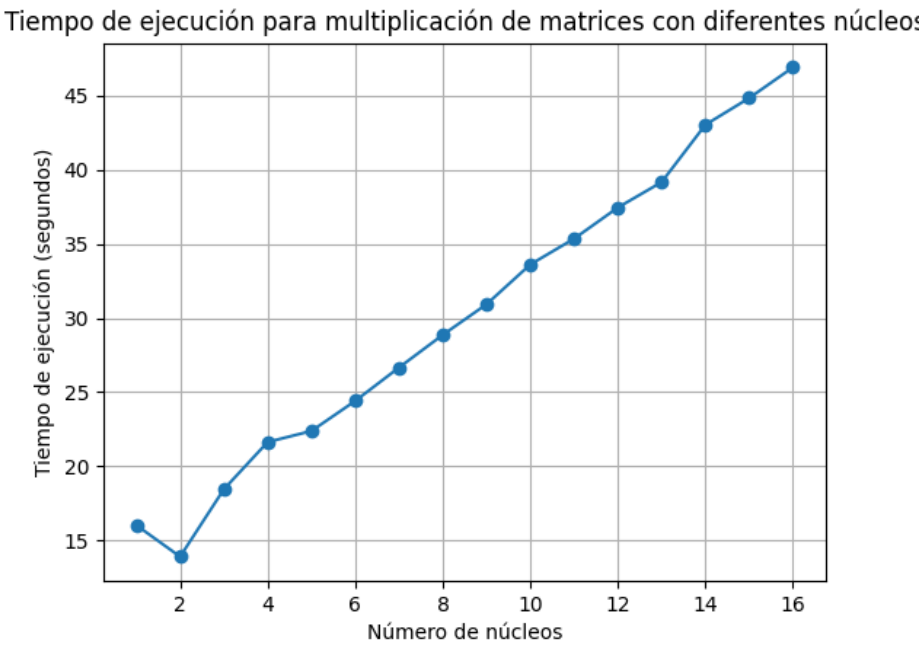

Figura 10: Comparación del tiempo para multiplicar matrices

V.1. Impresión en consola (Linux)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 16.0330 segundos
Tiempo de ejecución con 2 núcleos: 13.9134 segundos
Tiempo de ejecución con 3 núcleos: 18.4576 segundos
Tiempo de ejecución con 4 núcleos: 21.6383 segundos
Tiempo de ejecución con 5 núcleos: 22.3844 segundos
Tiempo de ejecución con 6 núcleos: 24.4109 segundos
Tiempo de ejecución con 7 núcleos: 26.6606 segundos
Tiempo de ejecución con 8 núcleos: 28.8639 segundos
Tiempo de ejecución con 9 núcleos: 30.9278 segundos
Tiempo de ejecución con 10 núcleos: 33.6096 segundos
Tiempo de ejecución con 11 núcleos: 35.3485 segundos
Tiempo de ejecución con 12 núcleos: 37.4487 segundos
Tiempo de ejecución con 13 núcleos: 39.1633 segundos
Tiempo de ejecución con 14 núcleos: 43.0304 segundos
Tiempo de ejecución con 15 núcleos: 44.8312 segundos
Tiempo de ejecución con 16 núcleos: 46.9006 segundos

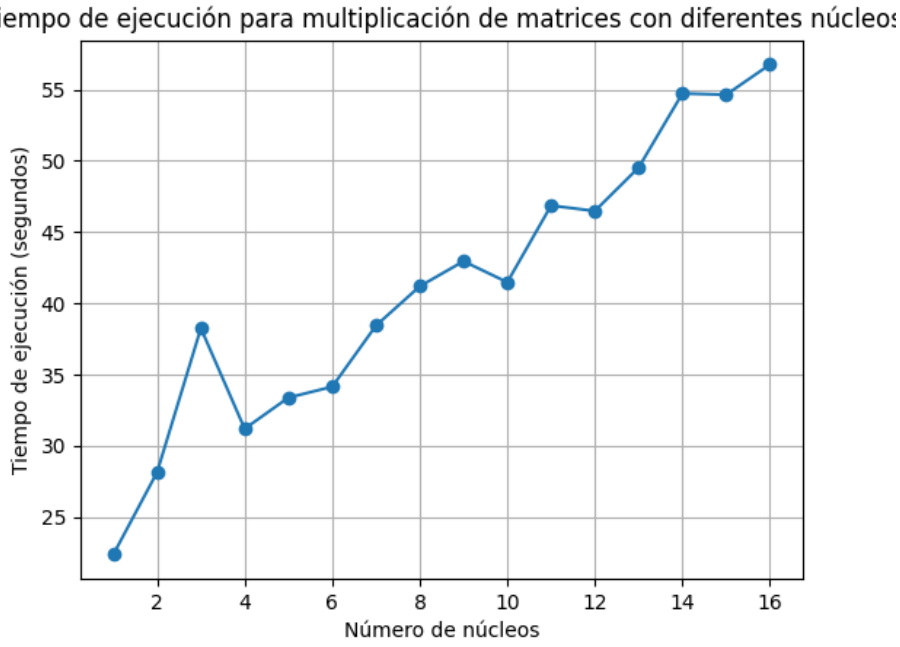

Figura 11: Comparación del tiempo para multiplicar matrices en Windows.

V.2. Impresiones en consola (Windows)

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 22.3788 segundos
Tiempo de ejecución con 2 núcleos: 28.1513 segundos
Tiempo de ejecución con 3 núcleos: 38.2561 segundos
Tiempo de ejecución con 4 núcleos: 31.2016 segundos
Tiempo de ejecución con 5 núcleos: 33.3862 segundos
Tiempo de ejecución con 6 núcleos: 34.1497 segundos
Tiempo de ejecución con 7 núcleos: 38.4395 segundos
Tiempo de ejecución con 8 núcleos: 41.1967 segundos
Tiempo de ejecución con 9 núcleos: 42.9540 segundos
Tiempo de ejecución con 10 núcleos: 41.4832 segundos
Tiempo de ejecución con 11 núcleos: 46.8634 segundos
Tiempo de ejecución con 12 núcleos: 46.4821 segundos
Tiempo de ejecución con 13 núcleos: 49.5273 segundos
Tiempo de ejecución con 14 núcleos: 54.7297 segundos
Tiempo de ejecución con 15 núcleos: 54.6410 segundos
Tiempo de ejecución con 16 núcleos: 56.7603 segundos

VI. Comparativa.

Ambos conjuntos de datos muestran que, en este escenario, la implementación de la paralelización empeora el rendimiento en lugar de mejorarlo. En Linux, el tiempo de ejecución aumenta de 16.03 segundos con 1 núcleo a 46.90 segundos con 16 núcleos, mientras que en Windows se observa un incremento de 22.38

segundos con 1 núcleo a 56.76 segundos con 16 núcleos. Esto evidencia que, a mayor cantidad de núcleos, se incurre en un sobrecosto que ralentiza la ejecución. Existen varias teorías que pueden explicar por qué la paralelización no funcionó en este caso:

- Sobrecarga de sincronización y comunicación.
- Granularidad inadecuada de la tarea.
- Contención de recursos y problemas de memoria (false sharing).
- Problemas de escalabilidad del algoritmo.
- Sobrecarga del sistema operativo y gestión de hilos.

La degradación del rendimiento observada al aumentar el número de núcleos sugiere que, en este caso, los costes asociados a la paralelización (sincronización, gestión de hilos, contención de memoria, entre otros) superan los beneficios potenciales.

VII. Resultados Algoritmo para obtener secuencia Fibonancci.

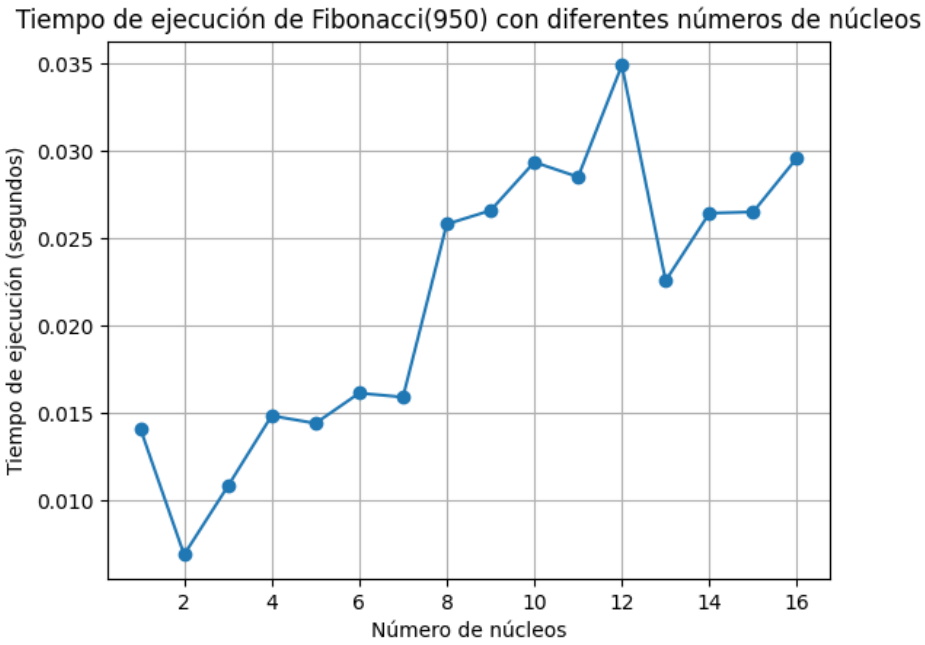

Figura 12: Comparación del tiempo requerido para obtener la secuencia de PI usando paralelización.

VII.1. Impresiónm en consola (Linux).

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 0.0141 segundos
Tiempo de ejecución con 2 núcleos: 0.0069 segundos
Tiempo de ejecución con 3 núcleos: 0.0108 segundos
Tiempo de ejecución con 4 núcleos: 0.0149 segundos
Tiempo de ejecución con 5 núcleos: 0.0144 segundos
Tiempo de ejecución con 6 núcleos: 0.0161 segundos

Tiempo de ejecución con 7 núcleos: 0.0159 segundos
Tiempo de ejecución con 8 núcleos: 0.0258 segundos
Tiempo de ejecución con 9 núcleos: 0.0266 segundos
Tiempo de ejecución con 10 núcleos: 0.0294 segundos
Tiempo de ejecución con 11 núcleos: 0.0285 segundos
Tiempo de ejecución con 12 núcleos: 0.0349 segundos
Tiempo de ejecución con 13 núcleos: 0.0226 segundos
Tiempo de ejecución con 14 núcleos: 0.0264 segundos
Tiempo de ejecución con 15 núcleos: 0.0265 segundos
Tiempo de ejecución con 16 núcleos: 0.0296 segundos

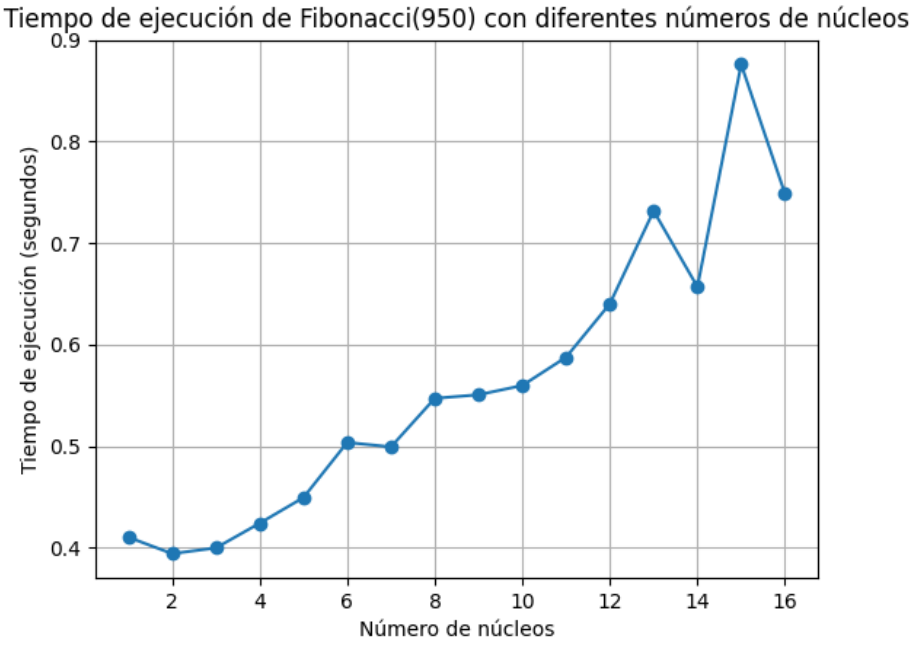

Figura 13: Comparación del tiempo requerido para obtener la secuencia de PI usando paralelización en Windows.

VII.2. Impresión en consola (Windows).

Tiempos de ejecución:
Tiempo de ejecución con 1 núcleos: 0.4104 segundos
Tiempo de ejecución con 2 núcleos: 0.3942 segundos
Tiempo de ejecución con 3 núcleos: 0.3998 segundos
Tiempo de ejecución con 4 núcleos: 0.4241 segundos
Tiempo de ejecución con 5 núcleos: 0.4494 segundos
Tiempo de ejecución con 6 núcleos: 0.5037 segundos
Tiempo de ejecución con 7 núcleos: 0.4993 segundos
Tiempo de ejecución con 8 núcleos: 0.5471 segundos
Tiempo de ejecución con 9 núcleos: 0.5507 segundos
Tiempo de ejecución con 10 núcleos: 0.5598 segundos
Tiempo de ejecución con 11 núcleos: 0.5876 segundos
Tiempo de ejecución con 12 núcleos: 0.6402 segundos

Tiempo de ejecución con 13 núcleos: 0.7318 segundos
Tiempo de ejecución con 14 núcleos: 0.6570 segundos
Tiempo de ejecución con 15 núcleos: 0.8760 segundos
Tiempo de ejecución con 16 núcleos: 0.7487 segundos

VIII. Comparativa.

Los resultados muestran que, en este escenario, la paralelización no aporta mejoras en el rendimiento y, de hecho, tiende a degradarlo a medida que se utilizan más núcleos. En Linux se observa que el tiempo mínimo de ejecución se alcanza con 2 núcleos (0.0069 segundos), pero al aumentar el número de núcleos la ejecución se vuelve más lenta, alcanzando 0.0296 segundos con 16 núcleos. En Windows ocurre algo similar, aunque en valores absolutos mucho mayores: partiendo de 0.4104 segundos con 1 núcleo, el tiempo incrementa hasta 0.7487 segundos con 16 núcleos.

IX. Resultados Algoritmo Redes Neuronales.

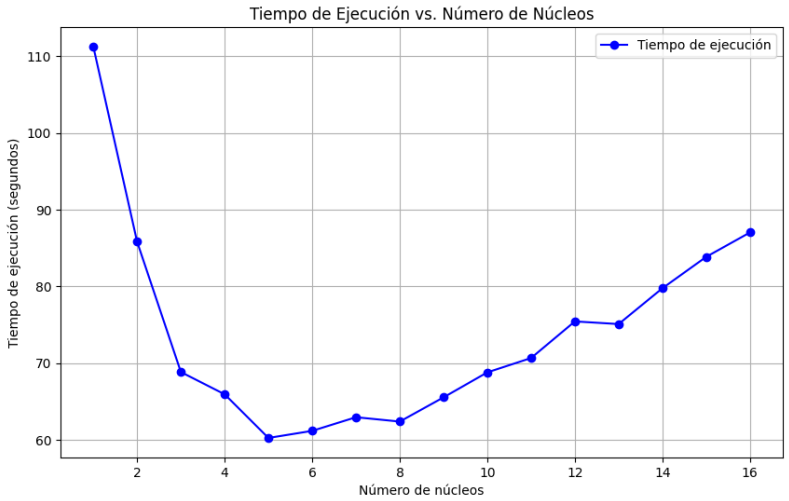

Figura 14: Comparación del tiempo que tarda en ser ejecutado el programa.

IX.1. Impresiones de consola en Windows.

Ejecutando con 1 núcleos...
Tiempo de ejecución con 1 núcleos: 111.2323 segundos
Precisión promedio con 1 núcleos: 0.9828
Ejecutando con 2 núcleos...
Tiempo de ejecución con 2 núcleos: 85.8331 segundos
Precisión promedio con 2 núcleos: 0.9737
Ejecutando con 3 núcleos...
Tiempo de ejecución con 3 núcleos: 68.8677 segundos
Precisión promedio con 3 núcleos: 0.9708
Ejecutando con 4 núcleos...
Tiempo de ejecución con 4 núcleos: 65.9708 segundos
Precisión promedio con 4 núcleos: 0.9688

Ejecutando con 5 núcleos...
Tiempo de ejecución con 5 núcleos: 60.2879 segundos
Precisión promedio con 5 núcleos: 0.9642
Ejecutando con 6 núcleos...
Tiempo de ejecución con 6 núcleos: 61.2111 segundos
Precisión promedio con 6 núcleos: 0.9618
Ejecutando con 7 núcleos...
Tiempo de ejecución con 7 núcleos: 62.9864 segundos
Precisión promedio con 7 núcleos: 0.9591
Ejecutando con 8 núcleos...
Tiempo de ejecución con 8 núcleos: 62.4192 segundos
Precisión promedio con 8 núcleos: 0.9553
Ejecutando con 9 núcleos...
Tiempo de ejecución con 9 núcleos: 65.5715 segundos
Precisión promedio con 9 núcleos: 0.9551
Ejecutando con 10 núcleos...
Tiempo de ejecución con 10 núcleos: 68.8257 segundos
Precisión promedio con 10 núcleos: 0.9510
Ejecutando con 11 núcleos...
Tiempo de ejecución con 11 núcleos: 70.6925 segundos
Precisión promedio con 11 núcleos: 0.9497
Ejecutando con 12 núcleos...
Tiempo de ejecución con 12 núcleos: 75.4600 segundos
Precisión promedio con 12 núcleos: 0.9478
Ejecutando con 13 núcleos...
Tiempo de ejecución con 13 núcleos: 75.1151 segundos
Precisión promedio con 13 núcleos: 0.9449
Ejecutando con 14 núcleos...
Tiempo de ejecución con 14 núcleos: 79.7772 segundos
Precisión promedio con 14 núcleos: 0.9434
Ejecutando con 15 núcleos...
Tiempo de ejecución con 15 núcleos: 83.8617 segundos
Precisión promedio con 15 núcleos: 0.9408
Ejecutando con 16 núcleos...
Tiempo de ejecución con 16 núcleos: 87.0400 segundos
Precisión promedio con 16 núcleos: 0.9386
[Done] exited with code=0 in 1316.087 seconds

Podemos observar que, al paralelizar este programa, su rendimiento no sigue una relación lineal entre el número de núcleos y el tiempo de ejecución. De hecho, el tiempo de ejecución más corto se obtuvo al utilizar 5 núcleos; a partir de ese punto, el programa tardó más en completarse. Además, es importante destacar que este script solo pudo ejecutarse en Windows, ya que en Linux se presentaron problemas de compatibilidad por lo que en este no podemos comparar cual de los SO es mejor para ejecutar el programa.

VI. Conclusión.

Este estudio ha permitido analizar a profundidad el impacto de la paralelización en el tiempo de ejecución de distintos algoritmos en sistemas multinúcleo. Se realizaron experimentos utilizando cálculos

matemáticos, algoritmos de búsqueda y redes neuronales profundas, evaluando el comportamiento del rendimiento al incrementar el número de núcleos empleados. Los resultados confirman que la paralelización puede mejorar significativamente la eficiencia computacional en aplicaciones que requieren un alto poder de cómputo, como el método de Monte Carlo y el entrenamiento de redes neuronales. Sin embargo, también se ha identificado que la relación entre el número de núcleos y el tiempo de ejecución no siempre es lineal ni garantiza una mejora constante. En ciertos casos, como en la multiplicación de matrices o la secuencia de Fibonacci, la paralelización generó un aumento en los tiempos de ejecución debido a la sobrecarga de sincronización, la contención de memoria y la administración de hilos en el sistema operativo. Además, el estudio revela diferencias importantes entre los sistemas operativos evaluados. En general, Linux mostró una mejor optimización en el manejo de múltiples procesos en comparación con Windows, obteniendo tiempos de ejecución más bajos en la mayoría de los experimentos. Esto puede deberse a una gestión más eficiente de los recursos del sistema y a la forma en que cada SO maneja la distribución de tareas en entornos multinúcleo. Otro factor que influyó en el rendimiento fue la gestión de la energía en computadoras portátiles. Se observó que, a medida que la batería se agotaba, los tiempos de ejecución aumentaban, lo que sugiere que el sistema ajusta dinámicamente la capacidad de procesamiento para conservar energía. Esto es un aspecto crucial a considerar en experimentos que dependen de la estabilidad del rendimiento del hardware. En conclusión, la paralelización es una herramienta poderosa para la optimización de programas computacionales, pero su efectividad depende de varios factores, incluyendo la naturaleza del problema, el overhead de sincronización y la arquitectura del sistema. No todos los algoritmos se benefician de un aumento en el número de núcleos, y en algunos casos, la sobrecarga generada puede superar los beneficios esperados. Este estudio proporciona una base para futuras investigaciones en optimización de algoritmos en entornos multinúcleo, destacando la necesidad de evaluar caso por caso si la paralelización es realmente beneficiosa o si una ejecución secuencial es más eficiente.

VII. Recursos.

I. Repositorio en Github.

En el siguiente link pueden encontrar los programas generados para este experimento: https://github.com/OmarGudi/Parallelization_using_Python

Referencias

[1] Hennessy, J. L., & Patterson, D. A. (2019). Computer Architecture: A Quantitative Approach (6th ed.). Elsevier.

[2] multiprocessing — Paralelismo basado en procesos — documentación de Python - 3.9.21. Pythonorg 2024. https://docs.python.org/es/3.9/library/multiprocessing.html

[3] IBM. neural networks. Ibmcom 2024. https://www.ibm.com/mx-es/topics/neural-networks

[4] Vive UNIR. La computación paralela: características, tipos y usos. UNIR 2024. https://www.unir.net/revista/ingenieria/computacion-paralela/#:~:text=La%20computaci%C3%B3n%20paralela%20consiste%20en,ha%20descompuesto%20un%20problema%20computacional.

[5] concurrent.futures — Launching parallel tasks. Python Documentation 2025. https://docs.python.org/es/dev/library/concurrent.futures.html.