UNIVERSIDAD DEL QUINDIO

PROYECTO FINAL:
MULTIPLICACIÓN DE MATRICES GRANDES

INTEGRANTES:

Jhoan Esteban Soler Giraldo
Julian Andres Hoyos Gomez
Johana Paola Palacio Osorio

PROFESOR: CARLOS ANDRÉS FLÓREZ VILLARRAGA

FACULTAD DE INGENIERÍA

ING. DE SISTEMAS

ANÁLISIS DE ALGORITMOS

ARMENIA - QUINDIO

2024

Tabla de índices:

Introducción..... 3
Arquitectura de la aplicación..... 3
1. Estructura General del Proyecto..... 3
2. Estructura de Carpetas..... 3
3. Flujo de datos en la arquitectura propuesta..... 5
Tabla 1. Órdenes de complejidad..... 7
Explicación de cada algoritmo mediante representación gráfica:..... 7
Conclusión..... 7

Introducción

En el ámbito de la computación, la multiplicación de matrices es una operación fundamental en diversos campos como álgebra lineal, procesamiento de imágenes, gráficos por computadora, y aprendizaje automático. Sin embargo, a medida que el tamaño de las matrices aumenta, la eficiencia y el rendimiento de los algoritmos de multiplicación se convierten en factores críticos. Este proyecto se centra en la implementación y análisis de diferentes algoritmos para la multiplicación de matrices grandes, aplicando métodos tanto iterativos como de divide y vencerás. Con este enfoque, se busca comprender el comportamiento de cada algoritmo bajo condiciones de prueba específicas, evaluando su eficiencia y tiempo de ejecución en arquitecturas y lenguajes de programación diversos.

Arquitectura de la aplicación

Para la arquitectura de este programa de multiplicación de matrices, se ha diseñado una estructura modular y bien organizada que permite realizar pruebas, ejecutar algoritmos en distintos lenguajes, almacenar resultados, y visualizar análisis de rendimiento.

1. Estructura General del Proyecto

El proyecto está dividido en dos implementaciones: una en Python y otra en Go. Ambas implementaciones ejecutan y comparan el rendimiento de diez algoritmos de multiplicación de matrices bajo diferentes tamaños de matrices y condiciones de prueba. La arquitectura está organizada en las siguientes carpetas y archivos clave**:**

2. Estructura de Carpetas

- **Documentos/:** Contiene todos los archivos de prueba y resultados.
    - Casos de prueba/: Carpeta con los archivos .txt que representan las matrices de entrada generadas aleatoriamente, usadas para las pruebas tanto en Python como en Go.
        - Caso1:
        - Caso2:
        - Caso3:
        - Caso4:
        - Caso5:
        - Caso6:
        - Caso7:
        - Caso8:

Cada subcarpeta CasoX/ tiene los siguientes componentes:

- **generar_matriz.py o equivalente en Go:** Un script que genera matrices aleatorias de tamaño específico (por ejemplo, 8x8, 16x16, etc.), que corresponden al tamaño designado para cada caso. Este script se ejecuta para crear las matrices de entrada en formato .txt.

- **Archivos de Matrices (matriz_A.txt y matriz_B.txt):**Cada carpeta de caso contiene dos archivos de texto (matriz_A.txt y matriz_B.txt) que representan las matrices de entrada para los algoritmos. Estos archivos .txt almacenan las matrices en un formato que puede ser leído por los scripts de Python y Go, y contienen valores generados aleatoriamente por el script GenerarMatrices.py.

- Resultados/: Carpeta que guarda los resultados de las ejecuciones en formato JSON.
  - go/: Almacena los resultados de la implementación en Go.
  - python/: Almacena los resultados de la implementación en Python.

- **PFAnalisisAlgoritmoGo/:** Carpeta que contiene la implementación en Go.

  - algoritmos/: Incluye los diez algoritmos de multiplicación de matrices en Go.
  - utilidades/: Archivos de soporte que contienen estructuras y métodos de utilidad como CasoMet y ResultadoMet para manejar los datos de los casos de prueba y sus resultados.

- **PFAnalisisAlgoritmosGraficos/:** Contiene los scripts en Python para generar gráficos de comparación, utilizando matplotlib.pyplot para visualizar los resultados de rendimiento de los algoritmos.

- **PFAnalisisAlgoritmosPython/:** Carpeta con la implementación en Python.

  - .vscode/: Archivos de configuración de entorno de desarrollo para facilitar la ejecución y depuración en Visual Studio Code.
  - algoritmos/: Carpeta que contiene los diez algoritmos de multiplicación de matrices implementados en Python.
  - utilidades/: Archivos de soporte, similares a los de Go, para gestionar estructuras y funciones de utilidad.
  - venv/: Entorno virtual para gestionar las dependencias del proyecto en Python, lo que permite aislar paquetes y versiones específicos para este proyecto.

3. Flujo de datos en la arquitectura propuesta

- **Generación de Matrices**

  Ubicación: Las matrices de entrada están en la subcarpeta Casos de prueba dentro de Documentos.

  Proceso de generación: Para cada caso de prueba, un script llamado generar_matriz.py genera dos matrices (matriz_A.txt y matriz_B.txt) que contienen valores aleatorios. Estas matrices tienen tamaños específicos (como 8x8, 16x16, 32x32, etc.), permitiendo pruebas de algoritmos con matrices de diferentes dimensiones.

  Cada caso de prueba se encuentra en una subcarpeta nombrada de forma incremental (por ejemplo, Caso1, Caso2, etc.) y contiene las matrices en formato .txt. Este formato es simple, fácil de leer y compatible con los algoritmos en Go y Python.

  Las matrices generadas en cada archivo .txt siguen una estructura de filas y columnas, donde cada línea representa una fila de la matriz, y los valores de cada elemento están separados por espacios. Este formato permite que ambos lenguajes lean la matriz de forma eficiente.

- **Ejecución de Algoritmos**

  Ubicación: Las implementaciones de los algoritmos están en las carpetas PFAnalisisAlgoritmoGo (para Go) y PFAnalisisAlgoritmosPython (para Python).

  Proceso de ejecución:
  Tanto los algoritmos en Go como en Python están diseñados para leer los archivos de matrices (matriz_A.txt y matriz_B.txt) de cada caso de prueba dentro de Documentos/casos de prueba.

  Ejecución en Go: Los algoritmos de Go, ubicados en PFAnalisisAlgoritmoGo/algoritmos, leen las matrices, ejecutan la operación de multiplicación y calculan métricas como el tiempo de ejecución. Para cada caso, registran estos datos en un archivo JSON que se guarda en Documentos/resultados/go. Esto se hace en el Main de ejecutable de go, y llama a las otras funciones.

  Ejecución en Python: De manera similar, los algoritmos en Python en PFAnalisisAlgoritmosPython/algoritmos leen las mismas matrices y realizan la multiplicación. Los resultados también se almacenan en formato JSON, pero en Documentos/resultados/python.De igual forma se corre el main.py que llama a las otras funciones y métodos de los algoritmos en la carpeta.

  Cada archivo JSON generado contiene:

- **casos:** Lista de casos de prueba con los resultados de cada tamaño de matriz.
- **muestras:** Lista de tiempos de ejecución registrados por cada ejecución del algoritmo.
- **promedio:** Tiempo promedio calculado a partir de las muestras, lo cual permite analizar la eficiencia.
- **tam:** Tamaño de la matriz (por ejemplo, 8, 16, 32, etc.).
- **tamanioBloques:** Tamaño de los bloques utilizados en el procesamiento por bloques.
- **lenguaje**: Lenguaje de programación utilizado (por ejemplo, "go" o "python").
- **nombre:** Nombre específico del algoritmo o método de ejecución utilizado.

- **Análisis y Visualización de Resultados**

  Ubicación: Los scripts para análisis y visualización están en PFAnalisisAlgoritmosGraficos.

  Proceso de análisis: Los archivos JSON generados en la carpeta Documentos/resultados (tanto los de Python como los de Go) son procesados por scripts de visualización en Python usando la biblioteca matplotlib.pyplot.

  Extracción de datos: Los scripts de análisis leen cada archivo JSON, obteniendo datos clave como el tamaño de la matriz, el promedio de tiempo de ejecución y los tamaños de bloque.

  Generación de gráficos:

  - Los gráficos se crean para comparar el rendimiento entre los algoritmos en Go y Python, mostrando el tiempo promedio de ejecución en función del tamaño de las matrices.
  - En los gráficos, cada punto o línea representa el rendimiento de un algoritmo específico para un tamaño de matriz particular, lo que permite observar tendencias de eficiencia en distintos tamaños de matrices.
  - La visualización clasifica los algoritmos y destaca el rendimiento en segundos, haciendo comparaciones directas y mostrando en qué condiciones un algoritmo puede ser más eficiente que otro.

Tabla 1. Órdenes de complejidad

Nota: Todos los órdenes de complejidad son realizados bajo la condición de que las matrices son cuadradas, sino fuera de esta manera algunos algoritmos tendrían una variación en su cálculo de Big(O).

Algoritmo	Órden de complejidad
1. **NaivOnArray**	$O(n^3)$
2. **NaivLoopUnrollingTwo**	$O(n^3)$
3. **NaivLoopUnrollingFour**	$O(n^3)$
4. **WinogradOriginal**	$O(n^3)$
5. **StrassenNaiv**	$O(n^{2.807})$
6. **StrassenWinograd**	$O(n^{2.807})$
7. **III.3 Sequential block**	$O(n^3)$
8. **IV.3 Sequential block**	$O(n^3)$
9. **V.3 Sequential block**	$O(n^3)$
10. **III.4 Parallel Block**	$O(n^3)$

Conclusión

El desarrollo de esta arquitectura organizada en Python y Go ha facilitado un análisis comparativo detallado del rendimiento de algoritmos de multiplicación de matrices, tomando en cuenta distintos tamaños de matriz y configuraciones de bloques. La división en módulos independientes para la generación de casos, ejecución de algoritmos, y visualización de resultados ha permitido un flujo de trabajo eficiente, simplificando tanto la recolección como el análisis de datos.

La estructura empleada, que incluye la generación de matrices aleatorias y el almacenamiento estandarizado de resultados en JSON, permite una fácil expansión del proyecto. Es posible añadir más algoritmos o configuraciones de prueba sin necesidad de cambiar la estructura general. La implementación en dos lenguajes y el análisis gráfico

facilitan la identificación de fortalezas y limitaciones de cada uno, ofreciendo información valiosa para optimizar el rendimiento en futuras aplicaciones.