Proyecto Final: Análisis de Rendimiento de Dotplot Secuencial vs. Paralelización

Juan Felipe Cortes Castrillon
Ing. en Sistemas y Computación
Universidad de Caldas
Manizales, Colombia
juan.1701721757@ucaldas.edu.co

Lorena Naranjo Arias
Ing. en Sistemas y Computación
Universidad de Caldas
Manizales, Colombia
lorena.1701814799@ucaldas.edu.co

David Salazar García
Ing. en Sistemas y Computación
Universidad de Caldas
Manizales, Colombia
david.1701713821@ucaldas.edu.co

***Abstract*—The objective of this project is to implement and analyze the performance of three ways to perform a dotplot, a technique commonly used in bioinformatics to compare DNA or protein sequences. You must implement a sequential version, a parallel version using the multiprocessing library, and a parallel version using mpi4py. Compare the performance of these three implementations using various metrics.**

***Index Terms*—multiprocessing, secuencial, mpi4py**

I. Introducción

La bioinformática es un campo interdisciplinario que combina la biología y la informática para analizar y comprender datos biológicos complejos. Una tarea común en bioinformática es comparar secuencias de ADN o proteínas para identificar similitudes y patrones relevantes. El dotplot es una técnica ampliamente utilizada para visualizar y analizar estas comparaciones, representando gráficamente la similitud entre dos secuencias en una matriz de puntos.
Este proyecto tiene como objetivo implementar y analizar el rendimiento de tres formas de realizar un dotplot, utilizando diferentes enfoques secuenciales y paralelos. Los resultados obtenidos permitirán determinar la mejor estrategia de implementación y proporcionarán conocimientos útiles en el campo de la bioinformática.

II. Objetivos

1) Implementar tres versiones del dotplot: El objetivo principal es implementar una versión secuencial del dotplot, una versión paralela utilizando la biblioteca multiprocessing de Python y una versión paralela utilizando mpi4py. Cada implementación debe ser capaz de generar un dotplot a partir de dos secuencias de entrada en formato FASTA.

2) Analizar el rendimiento de las tres implementaciones: Se busca comparar el rendimiento de las tres versiones del dotplot en términos de tiempos de ejecución, eficiencia y escalabilidad. Se deben calcular y analizar métricas como los tiempos de ejecución totales y parciales, el tiempo de carga de datos y generación de la imagen, el tiempo muerto, la aceleración y la eficiencia.

3) Visualizar los resultados: Se deben presentar los resultados obtenidos en un informe estilo artículo científico en formato IEEE. Este informe debe incluir gráficas de desempeño, aceleración, eficiencia y escalabilidad, además de describir los hallazgos y conclusiones derivados del análisis de rendimiento.

4) Optimizar el rendimiento mediante la implementación de un filtro de imagen paralelo: Además de generar el dotplot, se debe implementar una función paralela para realizar un filtrado de la imagen generada y detectar las líneas diagonales. Esta función debe estar optimizada para mejorar el rendimiento de la versión paralela del dotplot.

III. Fundamentos teóricos

A continuación se presentan los fundamentos teóricos necesarios para comprender los conceptos clave relacionados con el proyecto.

1) Dotplot:
Representa gráficamente la similitud entre dos secuencias mediante una matriz de puntos, donde cada punto representa una coincidencia entre los elementos de las secuencias en una posición determinada. Los patrones resultantes en el dotplot pueden revelar estructuras repetitivas, regiones conservadas o inversiones en las secuencias.

2) Bioinformática:
Disciplina que combina la biología y la informática para analizar y comprender los datos biológicos. En el contexto de este proyecto, la bioinformática se utiliza para realizar comparaciones de secuencias de ADN o proteínas y extraer información relevante sobre la estructura y función de los organismos vivos. La bioinformática se apoya en algoritmos y herramientas computacionales para realizar análisis a gran escala y obtener resultados significativos.

3) Comparación de secuencias:
Proceso fundamental en bioinformática. Implica la alineación y comparación de secuencias de ADN

o proteínas para identificar similitudes y patrones. Los métodos comunes de comparación de secuencias incluyen el alineamiento de secuencias, que busca encontrar regiones con similitudes y determinar la mejor manera de alinearlas, y la identificación de regiones conservadas, que señala regiones de alta importancia funcional en las secuencias.

4) Programación paralela con multiprocessing:
La programación paralela es una técnica utilizada para acelerar la ejecución de tareas intensivas en CPU mediante la distribución de la carga de trabajo en múltiples procesadores o núcleos. El módulo multiprocessing de Python permite la programación paralela a nivel de procesador, permitiendo que múltiples procesos trabajen en paralelo en una máquina con varios núcleos de CPU. Esto es especialmente útil para mejorar el rendimiento en tareas como la generación de dotplots, donde se pueden aprovechar los recursos de procesamiento múltiple.

5) Programación paralela con mpi4py:
Mpi4py es una biblioteca de Python que implementa el estándar de paso de mensajes MPI (Message Passing Interface). MPI es un estándar ampliamente utilizado en programación paralela en sistemas distribuidos. mpi4py permite la comunicación y la sincronización entre múltiples procesos en diferentes nodos de una red, lo que es esencial para abordar problemas de alta escalabilidad en bioinformática. El uso de mpi4py facilita la implementación de versiones paralelas del dotplot que pueden ejecutarse en múltiples nodos de un supercomputador o en un clúster de computadoras.

6) Optimización de rendimiento:
Proceso mediante el cual se busca mejorar el rendimiento de un programa o algoritmo, minimizando el tiempo de ejecución y maximizando

IV. Resultados y Análisis

- Aplicación de línea de comandos para dotplot:

```
python .\pruebasMulti.py \\
--file1=./data/E_coli.fna \\
--file2=./data/Salmonella.fna\\
--limite=10000 --cores=10
```

- Secuencial:
Para realizar el proceso de manera secuencial se intento de varias maneras ya que por cuestiones de máquina, específicamente memoria RAM, esto ya que a pesar de varios intentos por hacer las comparaciones seguía exigiendo demasiado espacio, entre las opciones que se intentaron fueron: realizar las comparaciones por bloques de datos, usar el método de la matriz dispersa y por último se optó por vectorizar por bloques de información.
Se intentó declarar una matriz haciendo uso de las listas tradicionales de python para almacenar la información del procesamiento de las cadenas, pero no fue posible crear una matriz del tamaño de secuencia1 x secuencia2, también se intento crear un dotplot del mismo lleno de ceros y hacer el calculo sobre el dotplot, pero tampoco fue posible, además de que se necesitaban 23tb de memoria. También se intento con una matriz vacía pero tampoco se podía asignar el espacio de memoria, pues python, incluso para un valor vacío o nulo reserva un espacio de memoria, así que lo que se hizo fue hacer uso de una matriz dispersa que tiene algunas ventajas sobre una matriz densa o normal. Hay varias diferencias entre una matriz dispersa creada con scipy.sparse y una matriz normal de Python utilizando listas. Aquí hay algunas diferencias clave:

  - Eficiencia de almacenamiento: Las matrices dispersas se optimizan para almacenar matrices con una gran cantidad de elementos cero. Almacenar matrices densas en listas de Python puede ser ineficiente en términos de uso de memoria, especialmente cuando la matriz tiene muchas entradas cero. Las matrices dispersas solo almacenan los elementos no cero, lo que puede ahorrar significativamente memoria y mejorar el rendimiento.

  - Eficiencia computacional: Las operaciones matemáticas en matrices dispersas se pueden realizar de manera más eficiente que en matrices densas almacenadas en listas de Python. Esto se debe a que las operaciones en matrices dispersas aprovechan la estructura y las propiedades especiales de estas matrices para reducir la cantidad de cálculos necesarios.

  - Funcionalidad especializada: scipy.sparse proporciona una variedad de formatos de matriz dispersa (como CSR, CSC, COO, entre otros) que están optimizados para diferentes operaciones y patrones de acceso. Estos formatos ofrecen ventajas en términos de acceso eficiente a filas, columnas o elementos específicos de la matriz, lo que puede ser útil en diferentes aplicaciones.

  - Uso de memoria: Las matrices dispersas suelen ocupar menos espacio en memoria que las matrices densas almacenadas en listas de Python. Esto es especialmente relevante cuando se trabaja con matrices grandes o dispersas, ya que puede reducir significativamente los requisitos de memoria y permitir el manejo de conjuntos de datos más grandes.

Para la graficación del resultado, no se sabe con exactitud

el tiempo que este se toma, pues no se halló una forma de medir el tiempo que se toma el matplotlib en hacer una gráfica del tamaño especificado, en una maquina con las siguientes especificaciones:
RAM: 16,0 GB
Procesador: AMD Ryzen 5 3600XT 6-Core Processor
Disco: De estado sólido NVMe M.2
Velocidad de base: 3,79 GHz
Sockets: 1
Núcleos: 6
Procesadores lógicos: 12
Caché L1: 384 kB
Caché L2: 3,0 MB
Caché L3: 32,0 MB

A partir de una matriz de 30000x30000 ya se presentan problemas y no se sabe si la ejecución va a terminar o el programa colapsó.
En las siguientes imágenes se puede visualizar dicha comparación, tomando 30000 datos de cada una de las 2 secuencias a comparar y graficando esta:

- Multiprocessing:
  Para hacer uso de esta librería y poder procesar ambas secuencias en un principio se uso una clase llamada lil matrix que se usa para crear matrices dispersas en formato ”lista de listas”, lo que quiere decir que, cada fila de la matriz se almacena como una lista separada, y las listas se almacenan en otra lista, esta permite agregar y modificar elementos en la matriz fácilmente. Sin embargo, las operaciones matriciales numéricas pueden ser muy lentas en comparación con otros formatos, por consiguiente se optó por usar la biblioteca np.zeros, la cual es una biblioteca de Numpy que se usa para crear un arreglo de valores cero, esta crea un arreglo de la forma y tipo de datos especificados, donde todos los elementos tienen un valor de cero, la cual es útil cuando se desea inicializar un arreglo con ceros antes de llenarlo con otros valores o para cuando se necesite realizar operaciones numéricas, entonces se dejó de usar lil matrix porque aunque se ahorra memoria es menos eficiente procesando cálculos matemáticos entre matrices, por el contrario de la librería de numpy, np.zeros.

  - Además de usar la librería Multiprocessing, se continuó usando la función para leer los archivos FASTA.
  - Se adaptó la función para crear el dotplot usado en el método secuencial, una de estas modificaciones para dicha adaptación fue el uso de un índice, este para saber en que carácter iba de la subsecuencia 1 dividida.
  - Se crea un función de comparación que es la que hace uso de la librería multiprocessing, donde también se crea el pool de procesos, que lo que hace es permitir que las tareas se descarguen a los procesos de trabajo de diferentes maneras.
  - Se divide la primera secuencia (Secuencia1) en subsecuencias con respecto al número de procesos.
  - Para los resultados se usó el método pool.apply async, que lo que hace básicamente es permitir ejecutar funciones en paralelo de manera asíncrona.
  - Se crea una matriz de ceros con numpy y con un tipo de dato uint8(para reducir el espacio en memoria) con dimensiones basadas en las longitudes de ambas secuencias a comparar.
  - Se utiliza tqdm para mostrar la barra de progreso cuando se itera sobre los resultados obtenidos de las tareas asíncronas
  - Se actualiza la parte correspondiente del dotplot utilizando el resultado parcial obtenido, usa la siguiente formula para calcular el rango de los índices : (índice * len(secuencia1) // numProcesos): ((índice + 1) * len(secuencia1) // numProcesos)
  - Por último retorna un dotplot resultante de la comparación entre las 2 secuencias.

- MPI4PY:
  Cuando se intento realizar el proceso de procesamiento de la comparación entre las dos secuencias usando MPI4PY, se encontró que cada proceso no podía almacenar más de 2Gb y esto mataba el proceso de manera inmediata por lo cual se decidió usar métodos de comunicación, específicamente los vistos en clase:

  - Broadcast: Lo que hace es enviar un valor o conjunto de valores desde un proceso o nodo a todos los demás procesos o nodos en un sistema distribuido. En otras palabras, un proceso envía una copia del dato a todos los otros procesos para que puedan acceder a él localmente. Esto es útil cuando se necesita compartir información común a todos los procesos sin necesidad de comunicación adicional.
  - Scatter: Operación inversa del ”gather”. En lugar de recopilar datos de múltiples procesos en un solo proceso, el ”scatter” distribuye datos desde un proceso central a múltiples procesos. El proceso central tiene una estructura de datos que contiene información para cada proceso secundario y se encarga de distribuir los datos a los procesos correspondientes. El ”scatter” se utiliza cuando se desea distribuir datos de manera equitativa o realizar cálculos paralelos donde cada proceso necesita su propio conjunto de datos.
  - Gather: Implica recopilar o recolectar datos de múltiples procesos o nodos en un solo proceso o nodo. En esta operación, cada proceso contribuye con su propio conjunto de datos y un proceso central los recopila en una estructura de datos centralizada. Esencialmente, se combina información dispersa en una sola ubicación. El ”gather” es útil cuando se necesita consolidar datos de múltiples fuentes para su posterior análisis o procesamiento.

  Entonces para solucionar los problemas que se habían

presentado de memoria fue:

- Dividir la secuencia 1 en chunks y los distribuye a los diferentes procesos. Cada proceso calcula su parte del dotplot y se recopilan los resultados parciales en el proceso 0. El proceso 0 combina los resultados parciales en un dotplot completo.
- Se divide la secuencia en subsecuencias de tamaño especifico.
- Se calculó el tamaño en megabytes de una matriz

- Tiempos de ejecución parciales en multiprocessing:

- Calculo de la escalabilidad:
  Escalabilidad = (Tiempo secuencial) / (Tiempo paralelo)

V. Conclusiones

- Después de haber realizado los procesos de comparación de secuencias con diferentes tipos de procesamiento, en este caso: secuencial, usando multiprocessing y usando MPI4PY, se puede observar que mientras se hace de manera secuencial se necesita mucha memoria ya que la consume muy rápidamente , usando multiprocessing se consume más CPU, sin embargo sigue consumiendo mucha memoria RAM pero con la diferencia de que este consumo se mantiene estable.

VI. Anexos

A continuación se encuentra el link del respositorio en Github con el código fuente de lo implementado del proyecto: https://github.com/juanF18/Analisis-Rendimiento-Dotplot.git

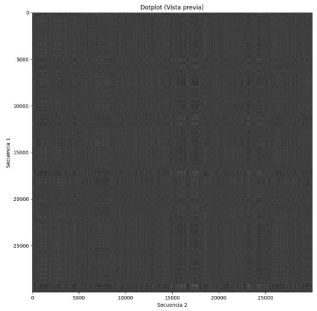

Fig. 1. Matriz de forma secuencial 30000 x 30000

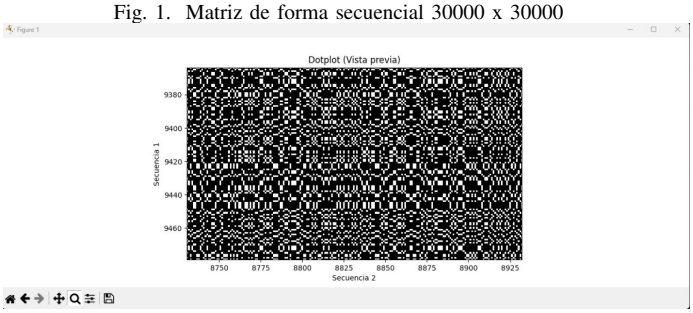

Fig. 2. Matriz de forma secuencial 30000 x 30000 con zoom

Fig. 3. Tiempo de lo que se tarda en procesar la matriz de 30000 x 30000

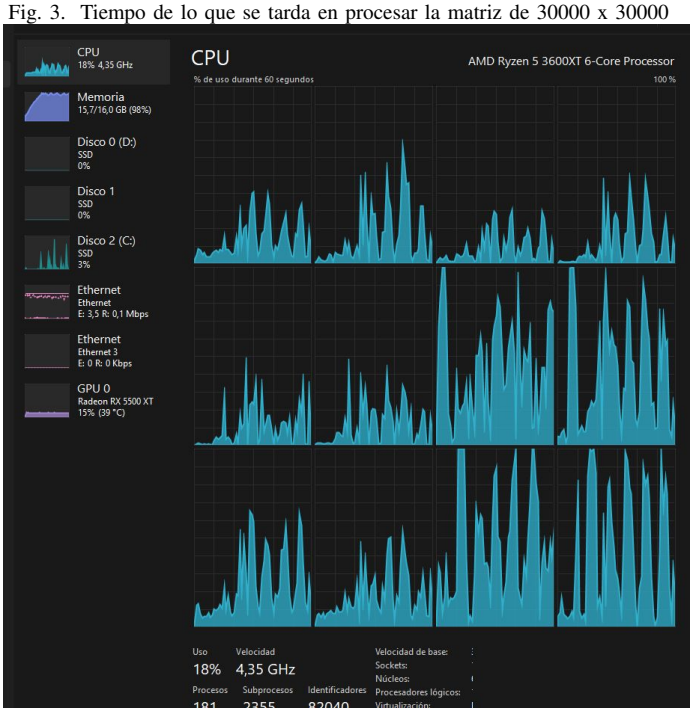

Fig. 4. Estado de la máquina 50000 x 50000, guardando una gráfica de 30000 x 30000

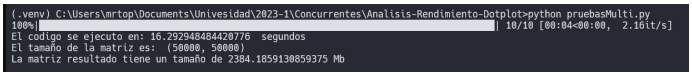

Fig. 5. Tiempo en multiprocessing 50000 x 50000 guardando la imagen

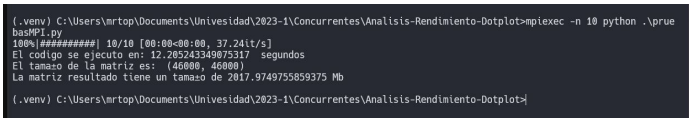

Fig. 6. Tiempo en MPI4PY 46000 x 46000 guardando una gráfica de 30000 x 30000)

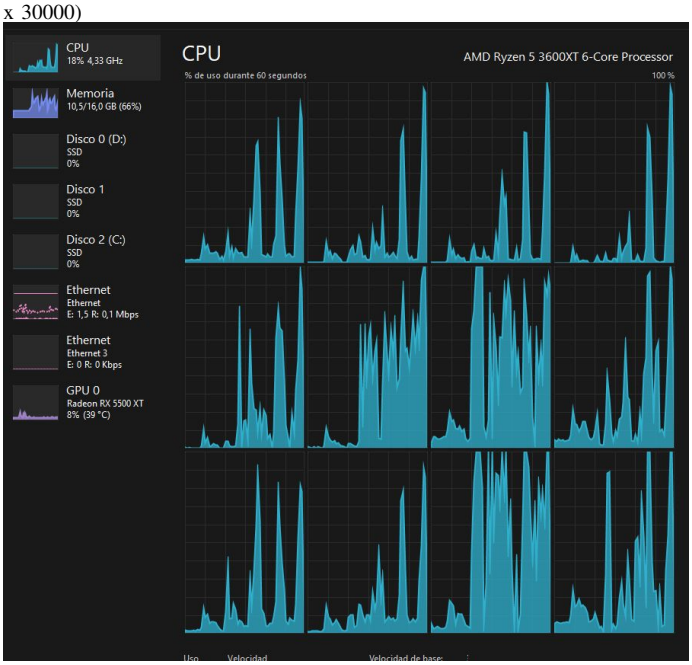

Fig. 7. Pantallazo del estado de la máquina mientras se ejecuta el proceso con multiprocessing)

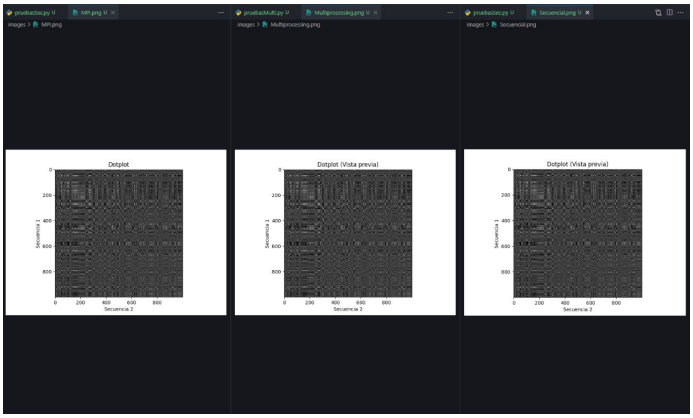

Fig. 8. Gráfica generada por los 3 métodos de procesamiento (Secuencial, multiprocessing y MPI4PY)

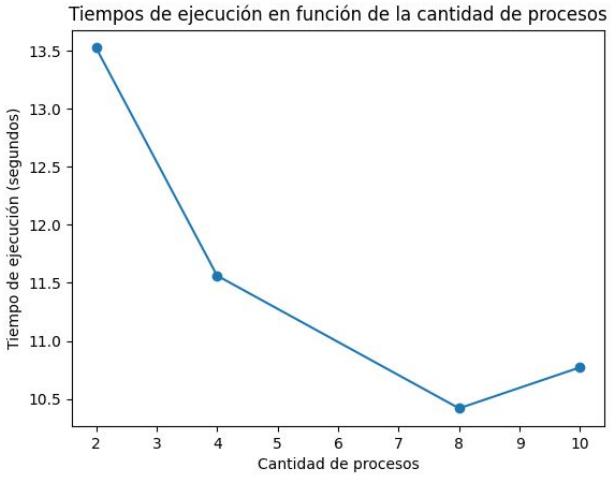

Fig. 9. Gráfica donde se relacionan el tiempo y el número de procesos usando Multiprocessing