Análisis de Rendimiento de Dotplot

Simón Cortés González, Isaac Enrique Castillo Devoz
Departamento de Sistemas e Informática
Programación Concurrente y Distribuida
Universidad de Caldas

simon.cortes48142@ucaldas.edu.co

***Abstract*—This project explores and compares four different approaches for performing dotplots: sequential, parallel using the Python multiprocessing library, parallel using mpi4py, and parallel using pyCuda. The objective is to analyze the performance of each implementation in terms of execution times, data loading, idle time, speedup, efficiency, and scalability. The results demonstrate significant performance differences among the implementations, highlighting the advantages and disadvantages of each approach. This study provides valuable insights for bioinformatics researchers seeking efficient and scalable strategies for sequence comparison.**

I. Introducción

La comparación de secuencias de ADN o proteínas es esencial en bioinformática para comprender su estructura y función. Un método comúnmente utilizado para este propósito es el dotplot, que permite visualizar la similitud entre dos secuencias de manera gráfica.[1]

El objetivo de este proyecto es implementar y analizar el rendimiento de diferentes enfoques para realizar un dotplot, centrandonos en tres implementaciones principalmente, además de la implementación en secuencial: paralela, utilizando la biblioteca Multiprocessing de Python, paralela utilizando MPI4PY y finalmente paralela utilizando la librería Pycuda.Las implementaciones realizadas permiten explorar el im pacto de la paralelización en el rendimiento de la comparación de secuencias y evaluar la escalabilidad de las soluciones propuestas.

En este informe se presentara en detalle el análisis de rendimiento realizado, incluyendo las mátricas utilizadas, los resultados obtenidos y las conclusiones derivadas de ellos. Además, se describir a la metodología implementada para realizar el dotplot y el filtrado de imagen, con el objetivo de proporcionar una comprension completa del enfoque utilizado.[2]

En la versión paralela utilizando multiprocessing, se aprovechará la capacidad de ejecutar múltiples procesos al mismo tiempo en sistemas con varios núcleos de CPU. Al dividir el trabajo en tareas más pequeñas y distribuirlas entre los procesos, se acelerará el cálculo del dotplot, logrando así un procesamiento más rápido y un uso eficiente de los recursos disponibles.

En la versión paralela con MPI4PY, se empleará la biblioteca mpi4py, basada en el estándar Message Passing Interface (MPI) para la programación paralela. Las tareas de cálculo del dotplot se repartirán entre los procesos MPI, lo que permitirá un rendimiento superior en comparación con la versión paralela que utiliza multiprocessing.

En la versión paralela con PyCUDA, se utilizará la biblioteca PyCUDA, que permite aprovechar la potencia de las GPU para la programación paralela. Las tareas de cálculo del dotplot se distribuirán entre los núcleos de la GPU, lo que permitirá un rendimiento superior en comparación con la versión paralela que utiliza multiprocessing.[3]

Al comparar estos tres métodos para realizar un dotplot, se analizará la eficiencia y el rendimiento de cada enfoque. Esto proporcionará información valiosa sobre las ventajas y desventajas de los métodos secuenciales y paralelos, así como una comparación entre las bibliotecas multiprocessing y mpi4py para tareas de bioinformática. Estos hallazgos podrían ser útiles para optimizar futuros proyectos de análisis comparativo de secuencias y mejorar la eficiencia en el procesamiento de grandes volúmenes de datos genómicos y proteómicos.

II. Materiales y Métodos

A. Librerías:

Se utilizó el lenguaje de programación Python en su versión 3.12.2 para implementar los diferentes algoritmos. Se emplearon varias bibliotecas clave, como Numpy, Matplotlib, Time, mpi4py, opencv, pycuda y Multiprocessing, las cuales son fundamentales para manipular matrices, generar gráficos, la lectura de archivos, detectar diagonales mediante filtros, medir tiempos, obtener parámetros de línea de comandos y

trabajar con múltiples procesadores.

En la Tabla I se detallan las versiones de cada uno de los software y bibliotecas utilizados en la investigación.

TABLE I
VERSIONES DE SOFTWARE Y BIBLIOTECAS UTILIZADAS

Paquete	Versión
Python	3.12.2
Matplotlib	3.9.0
Numpy	1.26.4
mpi4py	3.1.6
pycuda	12.2
Time	3.7
Multiprocessing	3.7
Opencv	4.10.0.82

B. Paralelización:

El proceso de paralelización se llevó a cabo utilizando las bibliotecas multiprocessing, mpi4py y pycuda de Python. En este proceso, se cargaron dos secuencias que se querían alinear gráficamente, organizándolas en una matriz nxm, donde las variables n y m representan las longitudes de las secuencias, respectivamente.

Con multiprocessing, se empleó la biblioteca Pool para generar los procesos según la cantidad de hilos especificados como parámetro. Por cada pool, se utilizó la función map para recorrer cada índice de la primera secuencia en comparación con cada índice de la segunda secuencia, guardando en una lista un valor de color que representa gráficamente la comparación realizada.

Con mpi4py, se aplicó la estrategia de Chunks, dividiendo la primera secuencia en matrices más pequeñas y creando otra matriz para almacenar la solución de la implementación. En cada recorrido de los Chunks contra las posiciones de la segunda secuencia, se realizó la misma comparación que en la implementación con multiprocessing, asignando un valor de color a la posición correspondiente en la solución, y luego uniendo las soluciones generadas.

Finalmente, para la implementación paralela utilizando PyCUDA, se aprovecha la capacidad de cómputo masivo de las GPU (Unidades de Procesamiento Gráfico) mediante la interfaz CUDA (Compute Unified Device Architecture) de NVIDIA. En esta implementación, las secuencias de entrada se convierten a su representación ASCII y se transfieren a la memoria de la GPU. Luego, se define un kernel CUDA que realiza el cálculo del dotplot comparando las secuencias en paralelo, utilizando una configuración de bloques e hilos que optimiza el uso de los recursos de la GPU.

Cada hilo de GPU realiza la comparación de un par de caracteres de las secuencias y almacena el resultado en una matriz de dotplot en la memoria de la GPU. Posteriormente, los resultados son transferidos de vuelta a la memoria del host para su visualización y análisis. Esta implementación permite una ejecución extremadamente rápida del cálculo del dotplot debido a la capacidad de procesamiento en paralelo de miles de hilos en la GPU, lo que resulta en una mejora significativa en el tiempo de cómputo en comparación con la implementación secuencial en CPU. Además, el uso de PyCUDA facilita la integración del código CUDA en un entorno Python, permitiendo una mayor flexibilidad y facilidad de uso para el desarrollo y prueba del algoritmo.

C. Arquitectura computacional:

Las pruebas se ejecutaron en 3 entornos diferentes. La primera parte se ejecuta en dos computadores que contienen las siguientes especificaciones: La primera máquina tiene 4 núcleos, es un AMD RYZEN 7 3700U con 8gb de RAM y la segunda máquina tiene 4 procesadores, es un intel core i5-5200U con 8gb de RAM. Las dos máquinas tienen como sistema operativo Windows 11.

La segunda parte, especificamente aquella que corre el código con la librería pycuda, se ejecuta en google collab, ya que la tarjeta grafica de ambos computadores no soporta ni accede a la ejecuciòn de esta biblioteca, por lo que es necesario recurrir a esta alternativa, ejecutandose en entorno GPU.

D. Datos de experimentación:

Para realizar las pruebas de rendimiento, se utilizó dos archivos FASTA.
Estos archivos contienen alrededor de 4 millones de bases nitrogenadas. Se ejecutó el algoritmo con el objetivo de comparar los tiempos de ejecución al aplicar estrategias paralelas como multiprocessing y mpi4py, en contraste con los tiempos de ejecución obtenidos en secuencial. De manera similar, se evaluó la eficiencia.

III. RESULTADOS

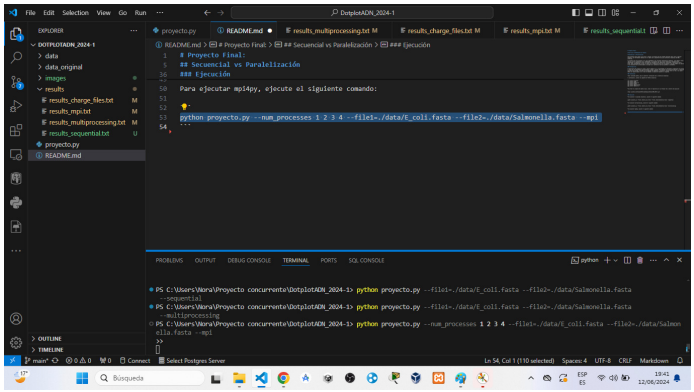

Fig. 1. Pruebas realizadas para encontrar la cantidad de datos que ambas computadoras pueden manejar para ejecutar la aplicación.

Se realizaron las pruebas necesarias hasta encontrar el máximo de datos en una matriz que cada máquina mencionada anteriormente podía manejar. Al final de las pruebas, se

determinó que una matriz conjunta con las secuencias uno y dos de 16,000 x 16,000 datos era la que se podía ejecutar en una de las máquinas.

Fig. 2. Ejecución en proceso de 16,000 x 16,000

Se muestra la evidencia del dotplot utilizando la biblioteca Multiprocessing, donde se observa la diagonal principal que servirá para el análisis de las secuencias durante la ejecución.

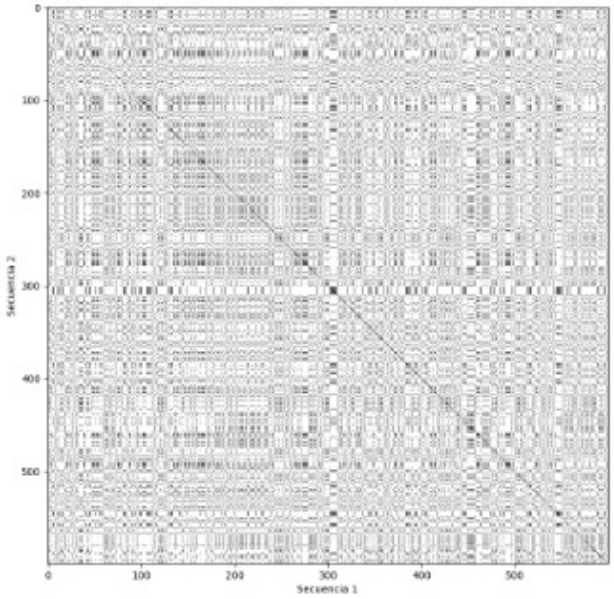

Fig. 3. Dotplot para la matriz evaluada utilizando Multiprocessing

En la ejecución paralela obtenemos diferentes resultados dependiendo del tipo de proceso que se ejecute. En este caso, la implementación paralela en Multiprocessing presenta los siguientes datos:

```
Tiempo de ejecuci□n multiprocessing con 1 hilos: 90.22142457962036
Tiempo de ejecuci□n multiprocessing con 2 hilos: 56.34539484977722
Tiempo de ejecuci□n multiprocessing con 4 hilos: 46.32770800590515
Tiempo de ejecuci□n multiprocessing con 8 hilos: 48.21783971786499
Aceleraci□n con 1 hilos: 1.0
Aceleraci□n con 2 hilos: 1.6012209129097457
Aceleraci□n con 4 hilos: 1.9474614321114334
Aceleraci□n con 8 hilos: 1.8711212511288182
Eficiencia con 1 hilos: 1.0
Eficiencia con 2 hilos: 0.8006104564548728
Eficiencia con 4 hilos: 0.48686535802785835
Eficiencia con 8 hilos: 0.23389015639110228
```

Fig. 4. Resultados de escalabilidad, aceleración y eficiencia de cada hilo en Multiprocessing

En la imagen anterior se observa la escalabilidad de la implementación. Se utilizaron 8 hilos (hilos lógicos de la máquina), el último tuvo una aceleración de ejecución de 1.171 minutos, lo cual se evidencia en la eficacia del proceso.

```
Tiempo de ejecuci□n mpi con 1 hilos: 112.91158628463745
Tiempo de ejecuci□n mpi con 2 hilos: 108.99878430366516
Tiempo de ejecuci□n mpi con 3 hilos: 109.94613075256348
Tiempo de ejecuci□n mpi con 4 hilos: 110.76001214981079
Aceleraci□n con 1 hilos: 1.0
Aceleraci□n con 2 hilos: 1.035897666253519
Aceleraci□n con 3 hilos: 1.026971895343437
Aceleraci□n con 4 hilos: 1.0194255498267417
Eficiencia con 1 hilos: 1.0
Eficiencia con 2 hilos: 0.5179488331267595
Eficiencia con 3 hilos: 0.342323965114479
Eficiencia con 4 hilos: 0.25485638745668543
```

Fig. 5. Resultados de escalabilidad, aceleración y eficiencia de cada hilo en MPI4PY

En la imagen anterior también se observa la escalabilidad de la implementación con MPI4PY. Se utilizaron 8 hilos nuevamente (hilos lógicos de la máquina), y el último tuvo una aceleración de ejecución de 1.81 minutos, evidenciando la eficacia del proceso.

Finalmente, se generaron, junto con la ejecución, las gráficas comparativas entre el tiempo de ejecución y la escalabilidad del algoritmo, junto con la aceleración y la eficiencia estructuradas a partir de la ejecución con las diferentes bibliotecas.

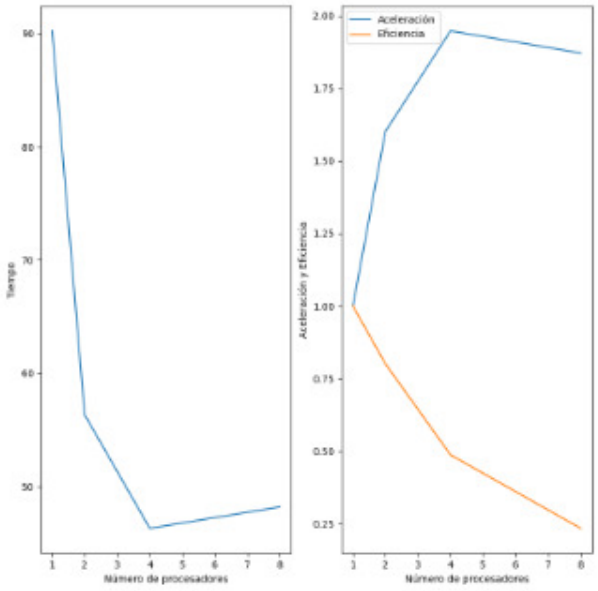

Fig. 6. Gráficas de aceleración y aceleración vs eficiencia en Multiprocessing

Al observar las gráficas de comparación entre tiempo de ejecución, aceleración y eficiencia, se puede ver que para

un hilo la ejecución tomaba más tiempo en reproducir la solución. A medida que se incrementaban los hilos, el tiempo de ejecución mejoraba. Sin embargo, con un mayor número de hilos, debido al overhead, los tiempos de ejecución aumentan, la aceleración disminuye y la eficiencia se reduce.

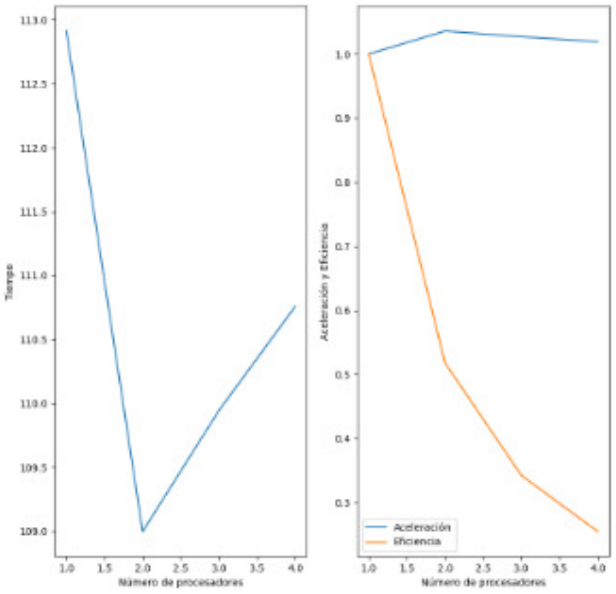

Fig. 7. Gráficas de aceleración y aceleración vs eficiencia en MPI

En la gráfica se observa que, a medida que aumentamos el número de núcleos, la ejecución del algoritmo se hace más rápida. Sin embargo, entre 3 y 4 núcleos, la aceleración disminuye y su eficiencia también.

Los resultados de la ejecución paralela varían según el tipo de proceso que se ejecute. En el caso de la implementación paralela en PyCUDA, se observan los siguientes datos:

```
Tiempo de ejecución PyCUDA con 1 hilos: 2.671177387237549 segundos
Tiempo de ejecución PyCUDA con 2 hilos: 1.6248271465301514 segundos
Tiempo de ejecución PyCUDA con 4 hilos: 1.2668488025665283 segundos
Tiempo de ejecución PyCUDA con 8 hilos: 0.8722903728485107 segundos
Tiempo de ejecución PyCUDA con 16 hilos: 0.8661980628967285 segundos
Hilos: 1, Aceleración: 1.0000, Eficiencia: 1.0000, Escalabilidad: 1.0000
Hilos: 2, Aceleración: 1.6440, Eficiencia: 0.8220, Escalabilidad: 1.6440
Hilos: 4, Aceleración: 2.1085, Eficiencia: 0.5271, Escalabilidad: 2.1085
Hilos: 8, Aceleración: 3.0623, Eficiencia: 0.3828, Escalabilidad: 3.0623
Hilos: 16, Aceleración: 3.0838, Eficiencia: 0.1927, Escalabilidad: 3.0838
```

Fig. 8. resultados de aceleracion, eficiencia y escalabilidad de cada hilo en PyCuda

La imagen anterior ilustra la aceleracion, eficiencia y escalabilidad de la implementación, utilizando hasta 16 hilos (hilos lógicos de la GPU). El último hilo logró un tiempo de ejecución de 0.8662 minutos, lo que demuestra la eficacia del proceso.

Para concluir, se generaron gráficas comparativas que ilustran el tiempo de ejecución y la escalabilidad del algoritmo, junto con la aceleración y la eficiencia obtenidas a partir de la ejecución con PyCUDA.

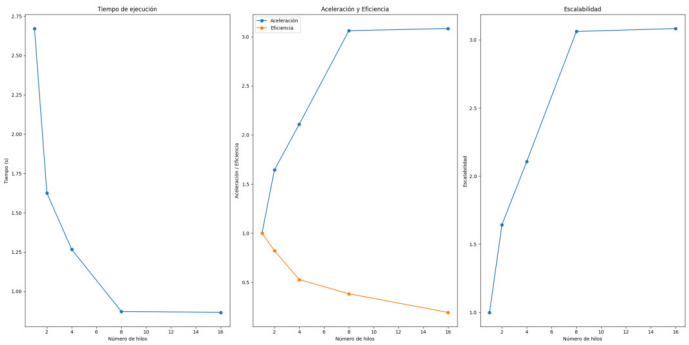

Fig. 9. Grafica de aceleracion, eficiencia y escalabilidad de cada hilo en PyCuda

El análisis de las gráficas de comparación entre tiempo de ejecución, aceleración y eficiencia revela que, con un solo hilo, la ejecución del algoritmo era más lenta. Sin embargo, a medida que se incrementaban los hilos y bloques, el tiempo de ejecución disminuía significativamente, lo que indica una mejor escalabilidad. No obstante, al utilizar configuraciones más complejas, se observa un ligero aumento en los tiempos de ejecución, una disminución en la aceleración y una reducción en la eficiencia debido al overhead.

IV. Disponibilidad del código fuente

La herramienta algorítmica presentada en este trabajo es de acceso libre. El código fuente y la guía de uso e instalación están disponibles en: **https://github.com/ScortesG2000/dotplotpy2024**

V. Referencias

- **Gonzalez, J. A. D., del Valle Gallegos, E., Torres, A. M. G. (2020). Paralelización hıbrida (MPI-OpenMP) en el código de transporte AZTRAN. In Memorias del XXXI Congreso Anual de la Sociedad Nuclear Mexicana.**
- **Duran Gonzalez, J. A., Del Valle Gallegos, E., Gomez Torres, A. M. (2020). Hybrid parallelization (MPI-OpenMP) in AZTRAN transport code.**
- **Valencia Pérez, T. A., Valencia Pérez, T. A. (2020). Implementación de algoritmos de reconstrucción tomográfica mediante programación paralela (CUDA).**