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

Albert David Pérez Muñoz, Bryan Mauricio González Giraldo, Luis Fernando Patiño Londoño
Programación Concurrente y Distribuida
albert.1701920351@ucaldas.edu.co
bryan.1701910677@ucaldas.edu.co
luis.1701814700@ucaldas.edu.co

***Resumen*— Este documento presenta un análisis de rendimiento de diversas implementaciones del dotplot, una herramienta gráfica crucial en bioinformática para la comparación de secuencias de ADN y proteínas. El estudio abarca cinco enfoques diferentes: secuencial, multihilos, multiprocessing, MPI utilizando la biblioteca mpi4py y PyCUDA. Se busca implementar y comparar el rendimiento de estos métodos mediante diversas métricas, incluyendo tiempo de ejecución total, tiempo de carga de datos, eficiencia y escalabilidad. El objetivo es proporcionar una visión clara y comparativa de cómo diferentes técnicas de paralelización impactan en el rendimiento del dotplot, ofreciendo una guía para optimizar el uso de recursos computacionales en futuras aplicaciones bioinformáticas. Se utilizaron secuencias del E-Coli y la Salmonella como datos de prueba para asegurar un análisis robusto y significativo.**

Palabras clave: Bioinformática, Dotplot, Comparación de secuencias, Secuencial, Paralelización, Multihilos, Multiprocessing, MPI, mpi4py, PyCUDA, Aceleración, Eficiencia, Escalabilidad, GPU, Análisis de rendimiento.

I. Introducción

El análisis de secuencias de ADN y proteínas es una tarea fundamental en bioinformática, esencial para la comprensión de la evolución, la función genética y la estructura de los organismos. Una técnica clave utilizada en este ámbito es el dotplot, una herramienta gráfica que permite la comparación visual de dos secuencias biológicas. El dotplot facilita la identificación de regiones de similitud y repetición, cruciales para la investigación genética y molecular.

Este proyecto tiene como objetivo implementar y comparar el rendimiento de diversas formas de generar un dotplot. Se propone analizar tres métodos diferentes: una versión secuencial, versiones paralelas utilizando multithreading y la biblioteca multiprocessing de Python, y una versión distribuida empleando mpi4py. Además, se incluye una implementación adicional utilizando pyCuda para aprovechar la capacidad de procesamiento de las unidades de procesamiento gráfico (GPU).

El trabajo no solo se centra en la creación de estas implementaciones, sino también en la evaluación de su desempeño mediante varias métricas. Estas métricas incluyen el tiempo de ejecución total, el tiempo de carga de datos, la eficiencia, la escalabilidad y el tiempo muerto, entre otros. Se utilizarán secuencias del E-Coli y la Salmonella como datos de prueba para asegurar un análisis robusto y significativo.

Para mejorar la visualización de las similitudes y facilitar su interpretación por especialistas, se aplicará un filtrado a la imagen generada por el dotplot. Este filtrado permitirá resaltar las líneas diagonales que indican regiones de alta similitud, haciendo más evidente la relación entre las secuencias comparadas. Este enfoque optimizado tiene como objetivo proporcionar una herramienta más efectiva para los investigadores en bioinformática, permitiéndoles identificar y analizar patrones en las secuencias con mayor precisión.

La finalidad de este proyecto es proporcionar una visión clara y comparativa de las diferentes técnicas de paralelización y su impacto en el rendimiento del dotplot, se presentarán los resultados y se discutirán las ventajas y desventajas de cada enfoque, ofreciendo así una guía valiosa para futuras aplicaciones y estudios en bioinformática.

II. Materiales y métodos

A. Lenguaje de Programación y Bibliotecas Utilizadas

El proyecto se implementó utilizando el lenguaje de programación Python, versión 3.11.9. Python fue elegido por su simplicidad, extensibilidad y amplia adopción en la comunidad científica y de bioinformática. Las bibliotecas utilizadas en el desarrollo del proyecto incluyen:

- Biopython: Utilizada para manejar archivos FASTA y realizar operaciones de manipulación de secuencias. Esta biblioteca facilita la lectura, escritura y análisis de secuencias biológicas.
- Multiprocessing: Parte de la biblioteca estándar de Python, se utilizó para paralelizar el proceso de generación del dotplot, distribuyendo la carga de trabajo entre múltiples procesos.
- Threading: También parte de la biblioteca estándar, se empleó para implementar una versión multihilo del dotplot, mejorando la eficiencia mediante el uso concurrente de los recursos.
- mpi4py: Biblioteca que permite la programación paralela y distribuida utilizando el estándar MPI (Message Passing Interface). Esta biblioteca facilita la ejecución del dotplot en un clúster de computadoras.
- PyCuda: PyCUDA es una biblioteca en Python que

facilita la programación paralela haciendo uso de la GPU (Unidad de Procesamiento Gráfico) para realizar cálculos de manera eficiente. En contraste con las implementaciones que se ejecutan en la CPU, PyCUDA permite aprovechar el enorme poder de procesamiento de la GPU para tareas intensivas en cómputo.

- Matplotlib: Utilizada para la visualización de los resultados y generación de las gráficas del dotplot. Esta biblioteca es ampliamente utilizada en la comunidad científica para crear visualizaciones estáticas, animadas e interactivas en Python.
- NumPy: Biblioteca fundamental para la computación científica en Python. Se utiliza para manejar arreglos multidimensionales y realizar operaciones matemáticas y lógicas sobre estos arreglos de manera eficiente.
- OpenCV (cv2): Biblioteca de visión por computadora utilizada para procesar y filtrar la imagen generada del dotplot. OpenCV permite aplicar técnicas avanzadas de procesamiento de imágenes para detectar y resaltar las similitudes entre las secuencias.

TABLA I
SOFTWARE

Software o Libreria	Versión
Python	***3.11.9***
Biopython	***1.83***
MPI4PY	***3.1.6***
PyCuda	***2024.1***
Matplotlib	***3.9.0***
NumPy	***1.26.4***
Scipy	***1.13.1***

B. Organización del Proyecto

El proyecto está organizado en varias carpetas y archivos para facilitar su mantenimiento y escalabilidad. A continuación, se describe la estructura del proyecto:

- data/: Contiene los archivos de datos en formato FASTA utilizados para las pruebas.
- processing/: Incluye los scripts y módulos encargados de la implementación del dotplot en sus diferentes versiones (secuencial, multihilo, multiproceso, distribuido y GPU).
- results/: Carpeta donde se almacenan los resultados generados por las diferentes implementaciones del dotplot.
- utils/: Contiene scripts auxiliares y funciones de utilidad que se utilizan en varias partes del proyecto.
- main.py: Archivo principal que ejecuta el dotplot utilizando la implementación especificada.
- requirements.txt: Archivo que lista todas las dependencias del proyecto, facilitando la instalación de las bibliotecas necesarias.
- readme: Documento que proporciona una descripción general del proyecto, instrucciones para la instalación y ejecución, así como detalles adicionales sobre la estructura del proyecto.

C. Implementación del Dotplot

La implementación del dotplot se llevó a cabo en varios enfoques para evaluar su rendimiento. Estas implementaciones incluyen:

- Versión Secuencial: La versión secuencial del dotplot se desarrolló como una solución básica para la comparación de secuencias biológicas. En esta implementación, el programa genera una matriz dotplot NxM que representa gráficamente las similitudes entre las secuencias. El proceso incluye la comparación de cada elemento de las secuencias y la generación de la imagen resultante. La comparación se realiza carácter por carácter: si los caracteres coinciden y están en la misma posición, se marca con un valor de 1; si no coinciden, se marca con un valor de 0. Este enfoque sirve como línea base para comparar el rendimiento de las versiones paralelas y distribuidas.
- Versión con Multithreading: Para mejorar el rendimiento y reducir el tiempo de ejecución, se implementó una versión del dotplot utilizando multithreading. En esta versión, el trabajo de comparación de secuencias se divide entre múltiples hilos que se ejecutan de forma concurrente. Cada hilo es responsable de procesar una parte de las secuencias, dividiendo la matriz de similitud en secciones que cada hilo se llena simultáneamente. La comparación sigue la misma lógica que en la versión secuencial (carácter por carácter con los mismos valores de coincidencia). Esto permite un uso más eficiente de los recursos del procesador y acelera el proceso general, especialmente en sistemas con múltiples núcleos, donde los hilos pueden ejecutarse en paralelo.
- Versión con Multiprocessing: La versión con multiprocessing utiliza la biblioteca multiprocessing de Python para paralelizar el proceso de generación del dotplot. A diferencia del multithreading, que utiliza hilos, multiprocessing crea múltiples procesos independientes, cada uno con su propia memoria. Este enfoque puede proporcionar una mayor eficiencia y mejor rendimiento en sistemas con múltiples núcleos, ya que los procesos pueden ejecutarse completamente en paralelo sin interferir entre sí. La implementación implica dividir la tarea de comparación de secuencias en varios procesos, donde cada proceso maneja una parte de la matriz de similitud. La comparación de caracteres sigue la misma lógica (valores de 1 y 0). Al finalizar, los resultados de todos los procesos se combinan para generar la matriz dotplot final.
- Versión Distribuida con mpi4py: MPI4PY es una biblioteca en Python que permite la programación paralela utilizando la interfaz de paso de mensajes MPI (Message Passing Interface). MPI permite a los procesos comunicarse entre sí, facilitando la distribución de tareas entre múltiples procesos que

pueden estar en la misma máquina o en diferentes máquinas, a diferencia de multiprocessing, que crea múltiples procesos independientes, cada uno con su propia memoria, multithreading utiliza hilos dentro de un solo proceso que comparten el mismo espacio de memoria. En la implementación se divide la tarea de comparación de secuencias en varios procesos, cada proceso manejando una parte de la matriz de similitud(Dotplot). En donde el proceso principal cargará y recortará las dos secuencias, los valores de estas secuencias serán intercambiadas por valores enteros, y se distribuirán partes de la primera secuencia a todos los procesos presentes, la comparación se hará caracter por caracter, si los caracteres coinciden se marca con un valor de 1, si es el caso contrario se marcará con un 0. Al finalizar, los resultados de todos los procesos se combinarán para generar la matriz dotplot final.

- Versión con pyCuda: En el proceso de generación de un dotplot utilizando PyCUDA, primero se carga y prepara el archivo de secuencias de ADN. Estas secuencias se convierten en matrices de números, donde cada base de ADN se asigna a un número entero utilizando un mapeo predefinido. La generación del dotplot se realiza en paralelo mediante el uso de un kernel CUDA, que es un pequeño fragmento de código escrito en lenguaje CUDA, un lenguaje similar a C/C++ diseñado para la programación de GPUs. Este kernel se ejecuta en paralelo en la GPU para calcular los valores de similitud entre las secuencias. El kernel CUDA compara cada elemento de las secuencias y calcula un valor de similitud basado en ciertas condiciones, como si los elementos son iguales o diferentes. Estos valores de similitud se almacenan en una matriz, que representa el dotplot final. Una vez completado el cálculo en la GPU, los resultados se copian de vuelta a la memoria principal del sistema.

D. Filtrado y Detección de Similitudes

Para mejorar la interpretación de las similitudes en dotplots y facilitar su análisis por parte de especialistas, se ha desarrollado una función de filtrado de imágenes. Este proceso consta de dos etapas principales:

- Detección de Líneas Diagonales: Se emplea un algoritmo especializado en la detección de bordes y líneas diagonales en la representación visual del dotplot. Esto resalta las áreas de alta similitud entre las secuencias, proporcionando una visualización más clara de los patrones presentes.
- Optimización del Rendimiento: La función de filtrado ha sido optimizada para ejecutarse eficientemente en paralelo. Esto se logra mediante el aprovechamiento de la capacidad de procesamiento paralelo de la biblioteca multiprocessing, lo que acelera significativamente el tiempo de procesamiento.

Este proceso de filtrado permite a los investigadores identificar con mayor precisión patrones y regiones de interés en los dotplots, mejorando su utilidad en aplicaciones bioinformáticas y facilitando la extracción de información relevante para el análisis de secuencias genéticas.

E. Análisis de Rendimiento

Se evaluaron varias métricas de rendimiento para cada implementación del dotplot:

- Tiempo de Ejecución Total: Se midió el tiempo total requerido para generar el dotplot desde la carga de datos hasta la finalización del proceso.
- Tiempo de Carga de Datos: Se registró el tiempo necesario para cargar y preparar los datos de las secuencias en formato FASTA.
- Tiempo de Procesamiento: Se midió el tiempo que tarda el procesamiento, ya sea secuencial o paralelo, en la creación de la matriz del dotplot.
- Tiempo de generación de imagen: Se calculó el tiempo que se tarda en crear la imagen a partir de la matriz del dotplot y lo que tarda el filtrado de la misma.
- Eficiencia: Se calculó la eficiencia de cada método comparando el tiempo de ejecución con el número de hilos/procesos/nodos/cuda cores utilizados.
- Escalabilidad: Se evaluó cómo el rendimiento del dotplot mejora al aumentar el número de hilos, procesos o nodos.
- Tiempo Muerto: Se analizó el tiempo en que los recursos de procesamiento no se utilizaban de manera efectiva durante la generación del dotplot.
- Escalabilidad: Es la capacidad de un algoritmo de mantener sus prestaciones cuando aumenta el número de procesos y/o el tamaño del problema (workload). Esta métrica no aplica para PyCUDA debido a que no podemos limitar la cantidad de cuda cores.

F. Datos de Prueba

Las implementaciones se probaron utilizando secuencias de los cromosomas X del E. coli y la Salmonella. Estos datos proporcionan un conjunto de pruebas adecuado debido a su tamaño y la similitud esperada entre las secuencias.

Dadas las restricciones de hardware, se decidió trabajar únicamente con una porción de los datos, utilizando los primeros 25.000 caracteres de cada secuencia para generar los dotplots mediante varios procesos. Esta elección permite observar los cambios en los tiempos de procesamiento al trabajar con conjuntos de datos más grandes, sin que dichos procesos se vieran afectados significativamente por las limitaciones de los equipos.

Para la generación de la imagen con PyCUDA, se redujo aún más el tamaño de la muestra a 21.000 caracteres de cada secuencia. Esto se debió a que las pruebas de PyCUDA se llevaron a cabo en un entorno virtual WSL, el cual presentaba problemas al procesar grandes cantidades de datos, resultando en la terminación abrupta de los procesos cuando se deseaba generar y filtrar la imagen. Sin embargo, es importante

mencionar que durante los tiempos de procesamiento de PyCUDA, se tuvieron en cuenta la totalidad de los 25000 caracteres. A pesar de ello, no fue posible generar imágenes con este conjunto completo debido a las restricciones de recursos del entorno WSL, que utiliza menos memoria RAM de la disponible en el dispositivo.

G. Hardware

Para llevar a cabo los ejercicios, procesos y obtener los resultados del dotplot, se utilizaron tres computadoras pertenecientes a los integrantes del equipo. A continuación, se detalla la configuración de hardware de cada una de estas máquinas:

- **PC 1**

  Procesador: AMD Ryzen 5 2400G (8 CPUs) 3.8 GHz

  RAM: 16 GB DDR4

  GPU: AMD Radeon RX 480

  Sistema Operativo: Windows 10 Pro

- **PC 2**

  Procesador: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz

  RAM: 8GB

  Sistema Operativo: Windows 11 - 23H2

- **PC 3**

  Procesador: 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz 2.69 GHz

  RAM: 8 GB

  GPU: Nvidia GeForce GTX 1650

  Sistema Operativo: - Windows 11 - 23H2

  - Ubuntu 22.04.3 LTS (WSL)

La combinación de estas tres computadoras permitió al equipo distribuir la carga de trabajo y aprovechar diferentes configuraciones de hardware para optimizar el rendimiento de las implementaciones del dotplot. El uso de procesadores multinúcleo, GPUs avanzadas y sistemas operativos diversificados contribuyó a una ejecución eficiente y rápida de los algoritmos, garantizando resultados precisos y reproducibles.

III. Resultados

Durante la ejecución de nuestro programa de dotplot, nos encontramos con limitaciones significativas debido a la capacidad de memoria disponible en las computadoras utilizadas. A pesar de nuestros esfuerzos por realizar pruebas exhaustivas con matrices de diferentes tamaños, la mayor matriz que logramos procesar exitosamente fue de tamaño 25000x25000. Esta limitación de memoria nos impidió explorar matrices de mayor tamaño, aunque creemos que con un hardware más avanzado y optimizaciones adicionales, sería posible realizar pruebas con matrices más grandes. La limitación de hardware se ve reflejado principalmente en la creación de la imagen y en el filtrado, porque generar únicamente la matriz del dotplot si nos funcionó cantidades de datos más grandes, pero desafortunadamente los recursos no dieron para crear la imagen ni filtrar.

```
Tiempo de procesamiento: 0.975433349609375 segundos
```

Fig. 1 Tiempo secuencial

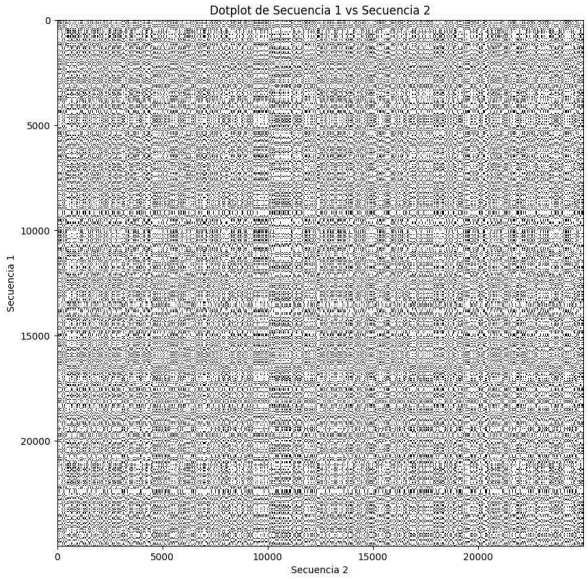

Fig. 2 Dotplot Solución para las cadenas evaluadas

Un dotplot filtrado es una herramienta gráfica utilizada para visualizar la similitud entre dos secuencias de ADN, destacando regiones de alta coincidencia y omitiendo el ruido generado por coincidencias esporádicas o menos significativas. Al aplicar un filtro, se mejora la claridad del gráfico, permitiendo identificar con mayor precisión segmentos de alineación conservada, repeticiones o regiones homólogas.

A continuación, se presenta la gráfica del dotplot filtrado, donde se puede observar claramente las coincidencias significativas entre las dos cadenas de ADN:

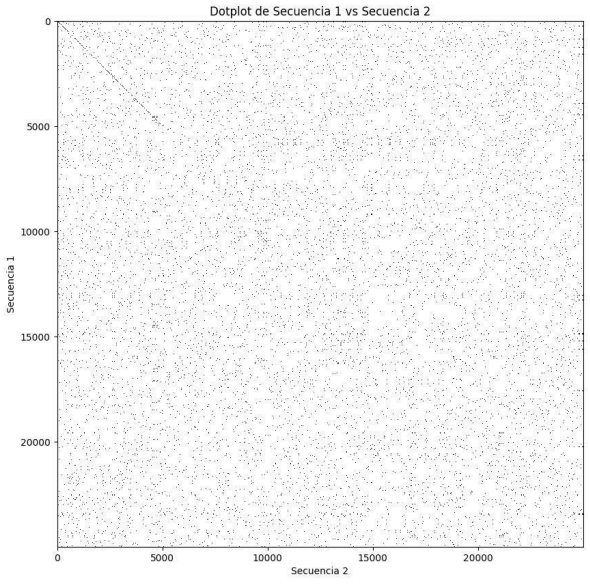

Fig. 3 Dotplot Filtrado para las cadenas evaluadas

Después de realizar el análisis secuencial del dotplot, procedimos a implementar el mismo proceso utilizando un enfoque de multi hilos (multi-threaded). Este método aprovecha la capacidad de procesamiento paralelo del sistema, distribuyendo la carga de trabajo entre múltiples hilos para mejorar la eficiencia y reducir el tiempo de ejecución.

A continuación, se presentan los resultados obtenidos utilizando el proceso multihilos, comparando el rendimiento, la eficiencia y la aceleración con el método secuencial:

ults > multithreaded > ≡ metrics_multithreaded.txt

```
1   Hilos, Tiempo Paralelo, Aceleraci□n, Eficiencia
2   1, 1.4231131076812744, 1, 1
3   2, 1.063080072402954, 1.3386697245340256, 0.6693348622670128
4   4, 0.5970497131347656, 2.3835755656079685, 0.5958938914019921
5   8, 0.48055076599121094, 2.96142095361326, 0.3701776192016575
6
```

Fig. 4 Tiempos para Aceleración, Escalabilidad y Eficiencia - Multithreaded

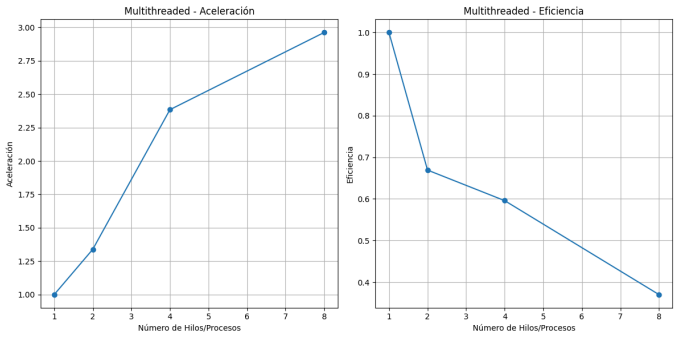

Fig. 5 Gráficas de Aceleración y Eficiencia - Multithreaded

Los resultados indican que, en este caso específico, el sistema muestra una buena escalabilidad fuerte, con una aceleración significativa. La aceleración aumenta de manera notable hasta alcanzar un valor aproximado de 3.

La eficiencia, por otro lado, disminuye rápidamente a medida que se incrementa el número de hilos. La caída es abrupta al pasar de 1 a 2 hilos, y continúa disminuyendo a medida que se añaden más hilos. A 8 hilos, la eficiencia es considerablemente baja, cercana a 0.1, lo que sugiere que la sobrecarga de gestión y coordinación entre hilos empieza a superar los beneficios de paralelismo.

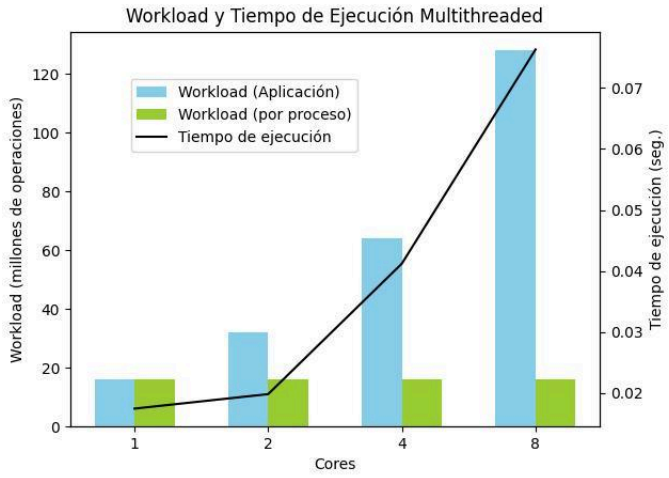

Fig. 6 Gráfica de escalabilidad - Multithreaded

Los resultados del análisis de escalabilidad nos muestra que el enfoque utilizado para multihilos no presenta escalabilidad débil, ya que probamos con duplicar la carga de datos según la cantidad de cores y como vemos en la figura 6 el tiempo no se mantuvo constante, sino que aumentó. Y manteniendo la cantidad de datos pero aumentando la cantidad de procesos vemos que los tiempos disminuyeron, reflejando que si presenta escalabilidad fuerte. Esto puede deberse a la cantidad de información que estamos duplicando y en la forma que paralelizamos el procedimiento.

Después de analizar los resultados obtenidos con el enfoque secuencial y el método multihilos, procedimos a implementar el proceso utilizando la técnica de multiprocessing. Este método permite ejecutar múltiples procesos en paralelo, aprovechando mejor las capacidades de los núcleos de la CPU y minimizando el tiempo total de procesamiento.

A continuación, se presentan los resultados obtenidos utilizando el enfoque de multiprocessing:

```
Procesos, Tiempo Paralelo, Aceleración, Eficiencia

1, 4.85518479347229, 1, 1

2, 4.636387586593628, 1.0471913106469628, 0.5235956565323481

4, 4.967324495315552, 0.9774245266342041, 0.24435613165855102

8, 6.19171142578125, 0.7841426157647001, 0.09801782697058752
```

Fig. 7 Tiempos para Aceleración, Escalabilidad y Eficiencia - Multiprocessing

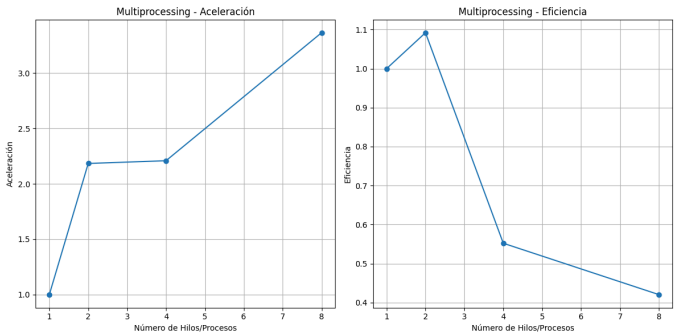

Fig. 8 Gráficas de Aceleración y Eficiencia - Multiprocessing

Según vemos la gráfica 7 de multiprocessing, indica que el cambio de 2 a 4 procesos es mínimo pero de 4 a 8 procesos si hay un aumento notorio en la aceleración. La eficiencia también disminuye con el aumento del número de procesos, lo cual es esperado en sistemas paralelos debido a que mientras más núcleos, menos procesamiento hace cada núcleo. Los tiempos de multiprocessing son más altos. Este comportamiento puede ser atribuido a varias razones. Primero, la sobrecarga de comunicación y sincronización entre procesos aumenta con el número de procesos. Segundo, la contención de recursos como memoria y ancho de banda del bus de datos puede causar que los procesos compitan por los mismos recursos. Tercero, la creación y gestión de múltiples procesos tiene un costo asociado, incluyendo el tiempo necesario para crear, inicializar, y finalizar procesos, así como para gestionar la comunicación entre ellos. Este overhead puede ser significativo, especialmente cuando se utilizan muchos procesos, ya que cada proceso adicional introduce más complejidad y requiere más recursos para la coordinación y la comunicación.

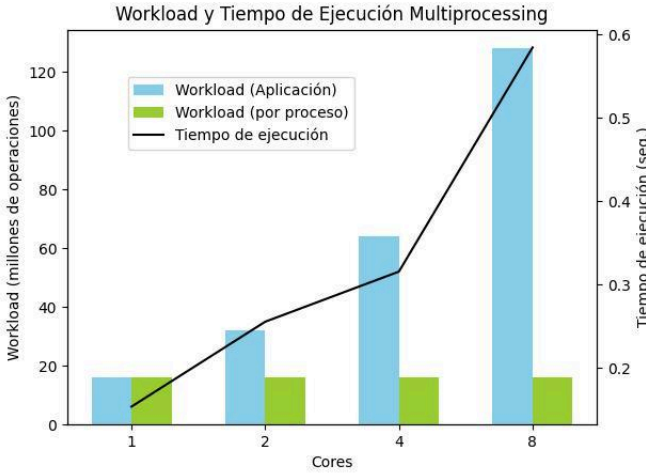

Fig. 9 Gráfica de escalabilidad - Multiprocessing

Los resultados del análisis de escalabilidad de multiprocessing nos muestra que tampoco presenta escalabilidad débil. Pero si nos fijamos en los tiempos de procesamiento al mantener la cantidad de datos y aumentar la cantidad de cores, vemos que si hay escalabilidad fuerte.

Después de realizar el análisis con los métodos secuencial, multihilos y multiprocessing, implementamos el proceso utilizando MPI (Message Passing Interface) con la biblioteca mpi4py. Este enfoque permite la comunicación entre múltiples procesos distribuidos en diferentes nodos, aprovechando al máximo la capacidad de procesamiento paralelo de un clúster de computadoras.

A continuación, se presentan los resultados obtenidos utilizando MPI con mpi4py, comparando el rendimiento y la eficiencia con los métodos previamente utilizados:

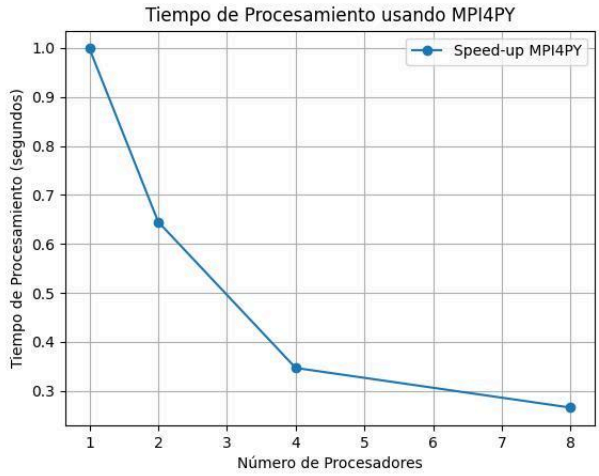

Fig. 10 Tiempos para Aceleración, Escalabilidad y Eficiencia - MPI4PY

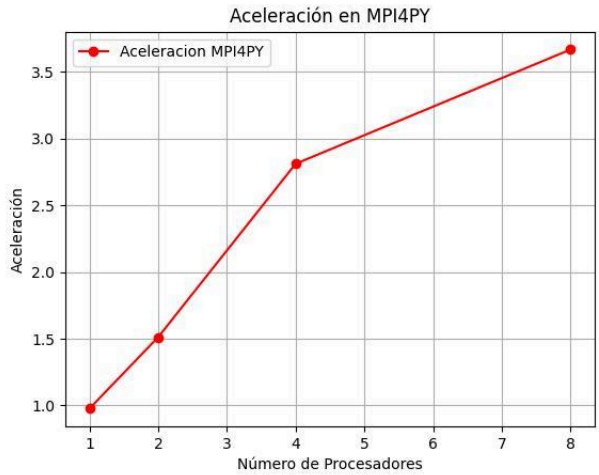

Fig. 11 Gráfica de aceleración - MPI4PY

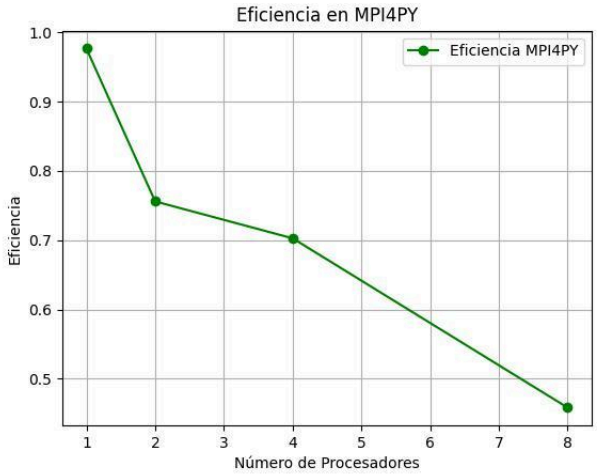

Fig. 12 Gráfica de eficiencia - MPI4PY

Estos resultados nos indican que MPI4PY es efectivo para reducir el tiempo de procesamiento y aumentar la aceleración al utilizar múltiples procesadores. Sin embargo, la eficiencia disminuye con el incremento del número de procesadores, debido a que cada proceso recibe menos carga de trabajo.

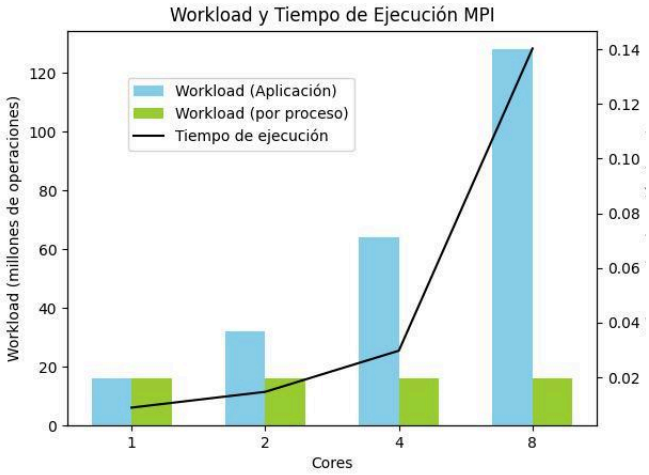

Fig. 13 Gráfica de escalabilidad - MPI4PY

En el análisis de escalabilidad de MPI4PY vemos que tenemos el mismo problema que con los demás enfoques. No hay escalabilidad débil, pero si nos fijamos en los tiempos de procesamiento cuando mantenemos la cantidad de datos y aumentamos los núcleos, el tiempo disminuye significativamente evidenciando que si presenta escalabilidad fuerte.

Después de analizar los resultados obtenidos con los métodos secuencial, multihilos, multiprocessing y MPI, procedimos a implementar el último proceso utilizando PyCUDA. Este enfoque permite aprovechar la capacidad de procesamiento paralelo de las GPU, que son especialmente eficaces para tareas intensivas en cómputo como la generación de dot plots.

A continuación, se presentan los resultados obtenidos utilizando PyCUDA, comparando el rendimiento y la eficiencia con los métodos previamente utilizados:

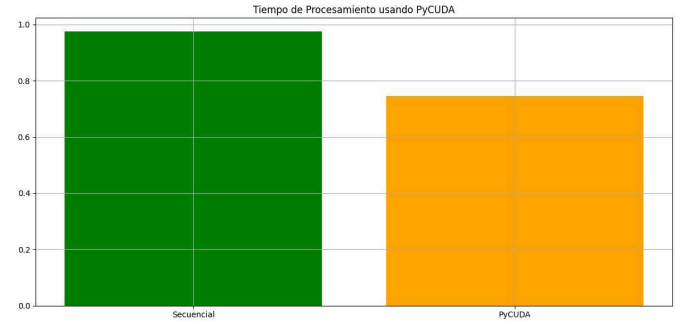

Fig. 14 Gráfica tiempos de procesamiento - PYCUDA

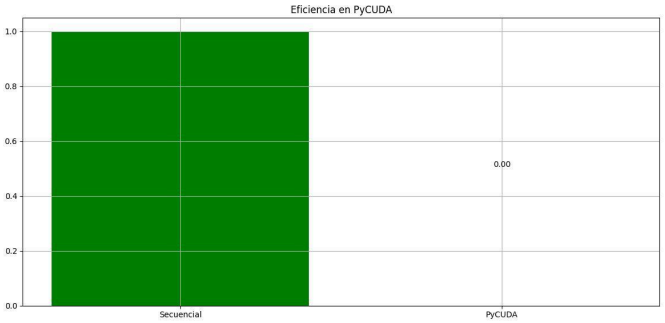

Fig. 15 Gráfica de eficiencia - PYCUDA

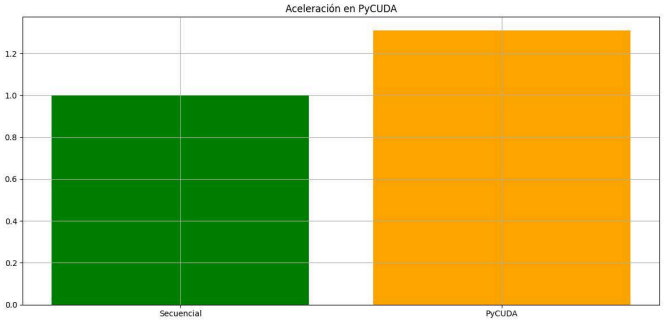

Fig. 16 Gráfica de aceleración - PYCUDA

Estos resultados indican que PyCUDA es un poco mejor en términos de tiempo de procesamiento y aceleración en comparación con el método secuencial. Sin embargo, la eficiencia es muy baja, lo que sugiere que, aunque PyCUDA es extremadamente rápido, no está utilizando los recursos de manera óptima. La baja eficiencia se debe a que no le estamos enviando la carga suficiente de procesos a todos los cuda_cores que posee la GPU. PyCUDA es una opción muy poderosa para tareas que requieren un procesamiento intensivo y pueden beneficiarse de la paralelización masiva proporcionada por las GPU, Si en lugar de 25.000 por secuencia usaramos 80.000 o 100.000 sí sería muy notoria la diferencia significativa en tiempos, pero no pudimos usar estas cantidades de datos debido a la ram que usa el WSL (Linux) ya que linux mata los procesos cuando la ram no es suficiente.

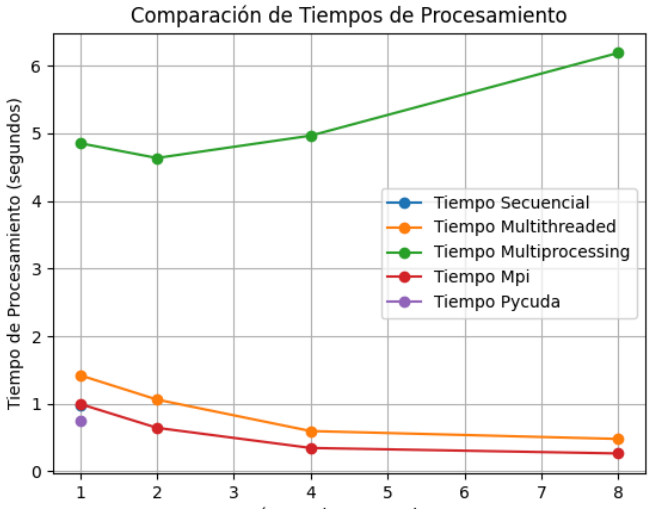

Fig. 17 Tiempos Totales de Procesamiento

En la comparación total de tiempos vemos que la versión más rápida es MPI, que usando 8 núcleos es incluso más rápida que PyCuda y por el lado opuesto vemos que multiprocessing tarda muchísimo más tiempo que las demás versiones.

IV. Conclusiones

El análisis de rendimiento comparativo entre diferentes implementaciones del dotplot (secuencial, multihilos, multiprocessing, MPI y PyCUDA) reveló las siguientes conclusiones clave:

Rendimiento y Tiempo de Ejecución:

- Multiprocessing: Aunque directa, la implementación con multiprocessing resultó ser la más lenta en términos de tiempo de ejecución total, debido a la sobrecarga de gestión entre procesos.

- Versiones Paralelas (Multihilos y MPI): Se observaron mejoras significativas en el rendimiento con las versiones paralelas en comparación con la secuencial. Multihilos y MPI mostraron una reducción considerable en el tiempo de ejecución, demostrando una buena aceleración en entornos adecuadamente configurados.

- PyCUDA: Presenta un rendimiento poderoso para el procesamiento intensivo, aunque se ve limitado por la eficiencia en la transferencia de datos entre la CPU y la GPU, especialmente en entornos como WSL que tienen restricciones significativas.

Eficiencia y Escalabilidad:

- Multihilos: Aunque inicialmente escalable, la eficiencia disminuye con el aumento del número de hilos debido a la sobrecarga de coordinación. El punto óptimo de aceleración se alcanza con 3 hilos, con una disminución significativa en eficiencia más allá de este punto.

- Multiprocessing: Ofrece mejor eficiencia que multihilos debido a la independencia de los procesos. La mejor aceleración se obtiene con 4 procesos, pero la escalabilidad no es un punto fuerte.

- MPI (mpi4py): Proporciona un buen equilibrio entre aceleración y eficiencia en entornos distribuidos. La eficiencia puede disminuir con más procesadores, pero muestra una escalabilidad notable hasta cierto punto.

- PyCUDA: Aunque extremadamente rápido en procesamiento, su eficiencia se ve comprometida por la sobrecarga de gestión de la GPU y la transferencia de datos, especialmente en entornos virtuales como WSL.

Limitaciones y Recursos:

- La capacidad de memoria de las computadoras utilizadas fue una limitación significativa, con la mayor matriz procesada exitosamente siendo de tamaño 25000x25000.

- PyCUDA enfrentó desafíos adicionales en entornos como WSL, donde la gestión de grandes volúmenes de datos resultó en la terminación abrupta de procesos.

Filtrado y Visualización:

- La aplicación de técnicas de filtrado mejoró significativamente la claridad y utilidad de los dotplots, facilitando la identificación precisa de patrones y regiones de similitud.

Recomendaciones para Futuros Estudios:

- Optimización de PyCUDA: Explorar entornos no virtualizados para resolver los problemas de eficiencia observados con PyCUDA, especialmente enfocándose en mejorar la gestión de la GPU y la transferencia de datos.

- Hardware Avanzado: Investigar el uso de hardware más avanzado para manejar matrices más grandes, lo que permitiría evaluar el rendimiento y la escalabilidad en un contexto más amplio.

- Técnicas Adicionales de Optimización: Considerar la implementación de técnicas avanzadas de optimización y gestión de memoria para mejorar el rendimiento general de las implementaciones paralelas y distribuidas del dotplot.

Estas conclusiones y recomendaciones están diseñadas para sintetizar los hallazgos del análisis comparativo de las implementaciones del dotplot, resaltando tanto los puntos fuertes como las áreas para futuras mejoras y estudios adicionales.