*PROYECTO FINAL PROGRAMACIÓN CONCURRENTE*

# *ANÁLISIS DE RENDIMIENTO DE DOTPLOT SECUENCIAL VS PARALELIZACIÓN*

*ERIKA FERNANDA ORREGO GIRALDO*
*JOJHAN PEREZ ARROYAVE*
*RAMDHEI LÓPEZ ARCILA*

*PRESENTADO A: REINEL TABARES SOTO*

*UNIVERSIDAD DE CALDAS*
*FACULTAD DE INGENIERÍA*

*JUNIO 2024*

# ANÁLISIS DE RENDIMIENTO DE DOTPLOT SECUENCIAL VS PARALELIZACIÓN

Erika Fernanda Orrego[1], Jojhan Perez Arroyave[2], Ramdhei López Arcila[3]

[123]*Departamento de Ingeniería de Sistemas y Computación, Universidad de Caldas, Manizales, Colombia*

erika.1702010705@ucaldas.edu.co, jojhan.1702011305@ucaldas.edu.co, ramdhei.1702011131@ucaldas.edu.co

***RESUMEN*:** Este proyecto analiza y compara el rendimiento de diferentes enfoques para la implementación de dotplot, una técnica utilizada en bioinformática para comparar secuencias de ADN o proteínas. Se desarrollaron cinco versiones: secuencial, con hilos, utilizando la biblioteca multiprocessing de Python, mpi4py y pyCuda. Las implementaciones se evaluaron en términos de tiempos de ejecución, escalabilidad, eficiencia y aceleración utilizando secuencias de prueba. Los resultados muestran que las versiones paralelizadas ofrecen mejoras significativas en el rendimiento, especialmente en términos de aceleración y eficiencia en la gestión de datos a gran escala. Las conclusiones incluyen recomendaciones para la aplicación de estos métodos en análisis de secuencias en bioinformática.

***PALABRAS CLAVE***: Paralelización, Rendimiento, Secuencias de ADN, Técnicas de Dotplot

***ABSTRACT***: This project analyzes and compares the performance of different approaches for the implementation of dotplot, a technique used in bioinformatics to compare DNA or protein sequences. Five versions were developed: sequential, threaded, using the Python multiprocessing library, mpi4py and pyCuda. The implementations were evaluated in terms of execution times, scalability, efficiency and speedup using test sequences. The results show that parallelized versions offer significant performance improvements, especially in terms of acceleration and efficiency in large-scale data management. The conclusions include recommendations for the application of these methods in sequence analysis in bioinformatics.

***KEY WORDS***: Parallelization, Performance, DNA sequences, Dotplot techniques

## I. INTRODUCCIÓN

La bioinformática es un campo que ha visto un crecimiento exponencial en la última década debido a los avances en la secuenciación genética y el análisis de datos. Una técnica fundamental en este ámbito es el dotplot, un método gráfico que permite la comparación visual de secuencias de ADN o proteínas para identificar regiones de similitud. El dotplot es especialmente útil en estudios de alineación de secuencias y análisis evolutivos, permitiendo la detección de homologías y patrones repetitivos que pueden ser indicativos de funciones biológicas conservadas.

A pesar de su utilidad, la generación de dotplots para secuencias de gran tamaño puede ser computacionalmente intensiva, requiriendo soluciones que optimicen el rendimiento. La implementación tradicional secuencial del dotplot, aunque efectiva para conjuntos de datos pequeños, resulta ineficiente cuando se aplican a genomas completos o a grandes secuencias proteicas, debido a la complejidad $O(n^2)$ inherente al proceso.

Con el fin de abordar esta limitación, se han explorado diversas estrategias de paralelización que buscan mejorar la eficiencia y reducir el tiempo de procesamiento. Este proyecto investiga y compara el rendimiento de varias implementaciones del dotplot: la versión secuencial, una versión con hilos, una versión paralelizada utilizando la biblioteca `multiprocessing` de Python, `mpi4py` para paralelización en clústeres, y `pyCuda` para aprovechar la capacidad de cómputo de las GPU.

El objetivo principal es evaluar estas implementaciones utilizando secuencias de prueba de cromosomas considerando métricas como tiempos de ejecución, escalabilidad, eficiencia y aceleración. Este análisis no solo proporcionará una comprensión de las ventajas y limitaciones de cada enfoque, sino que también ofrecerá recomendaciones sobre su aplicabilidad en estudios bioinformáticos a gran escala.

En las secciones que siguen, se describen detalladamente las metodologías de implementación, las métricas de rendimiento evaluadas, y se presentan los resultados del análisis comparativo entre las diferentes técnicas. Las conclusiones aportan una visión sobre las mejores prácticas para la optimización de dotplots en contextos de alto rendimiento.

## II. DESARROLLO DEL CONTENIDO

### *A. Código secuencial*

La versión secuencial del dotplot se implementa en Python y utiliza la biblioteca BioPython para la lectura de secuencias desde archivos FASTA, numpy para manipulaciones de matrices, matplotlib para la visualización y scipy para la creación de matrices dispersas. Esta versión es capaz de procesar secuencias de gran tamaño, limitadas por un parámetro de longitud máxima si es necesario.

1) *Lectura de Secuencias*: La función *read_fasta* toma un archivo FASTA y, opcionalmente, un número máximo de caracteres a leer. Esto permite flexibilidad en la manipulación de secuencias potencialmente largas.
2) *Generación del Dotplot:* La función *generate_dotplot* utiliza una ventana de tamaño definido por el usuario para comparar secuencias y generar una matriz dispersa indicando puntos de coincidencia. Se hace uso de operaciones vectorizadas en numpy para mejorar la eficiencia del proceso.
3) *Visualización*: La función *plot_dotplo*t convierte la matriz dispersa a una imagen y la guarda en formato PNG. Esta representación visual es crucial para la interpretación biológica de las similitudes entre secuencias.
4) *Manejo de la Memoria:* Se incluye un manejo de errores para la memoria, lo que es esencial dado el tamaño potencial de las secuencias involucradas.

El script se ejecuta como una aplicación de línea de comandos que acepta argumentos para especificar los archivos de entrada y salida. Esto es coherente con los requisitos del proyecto de facilitar una aplicación ejecutable desde la línea de comandos.

```
python dotplot_secuencial.py
--file1=./dotplot_files/E_coli.fna
--file2=./dotplot_files/Salmonella.fna
--output=dotplot_secuencial.png
--max_length=50000
```

#### *1) Análisis de rendimiento para diferentes tamaños de secuencias*

En esta sección, analizamos cómo el tamaño de las secuencias afecta el tiempo de ejecución del dotplot secuencial. Se consideraron tres tamaños de prueba: 10,000, 25,000 y 50,000 caracteres para cada secuencia.

Se realizaron pruebas ejecutando el script *dotplot_secuencial.py* con dos secuencias de ADN de diferentes tamaños, cada una procedente de los archivos FASTA de Escherichia coli y Salmonella. Los tiempos de ejecución se midieron utilizando la utilidad de línea de comandos integrada en el sistema operativo.

| Tamaño de Secuencia (caracteres) | Tiempo de Cálculo del Dotplot (segundos) | Tiempo de Generación y Guardado de la Imagen (segundos) | Tiempo Total de Ejecución (segundos) |
|---|---|---|---|
| 10,000 | 5.30 | 0.61 | 5.96 |
| 25,000 | 34.19 | 2.60 | 36.84 |
| 50,000 | 630.23 | 36.37 | 666.69 |

*Fig. 1. Tabla de comparación de los resultados obtenidos*

Se observa una relación claramente cuadrática entre el tamaño de las secuencias y el tiempo de ejecución. Esto es esperado en operaciones que implican comparaciones por pares, donde el costo computacional crece con el cuadrado del número de elementos a comparar. La generación y guardado de la imagen también aumenta, aunque no en la misma proporción que el cálculo del dotplot.

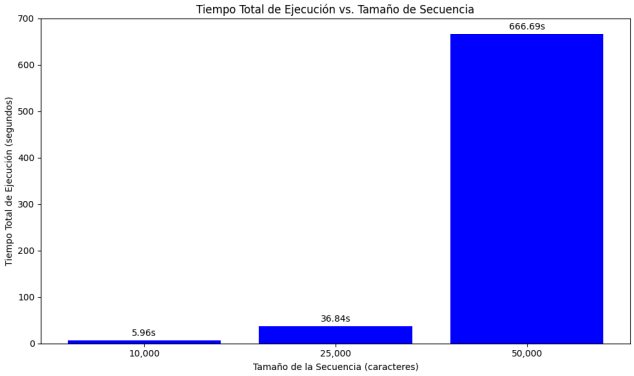



*Fig. 2. Gráfico de barras del tiempo de ejecución con diferentes tamaños de secuencia*

El aumento exponencial en el tiempo de ejecución para secuencias más largas subraya la necesidad de métodos paralelos o distribuidos para manejar secuencias de tamaño genómico. Estos resultados serán comparados en secciones subsiguientes con las implementaciones paralelas y distribuidas para evaluar las mejoras en el rendimiento.

### *2) Visualización de dotplots*

La figura 3 muestra el dotplot resultante de comparar secuencias de longitud reducida. La relativa simplicidad de este dotplot permite una fácil identificación de regiones de coincidencia, ideal para demostrar la funcionalidad básica del algoritmo secuencial.

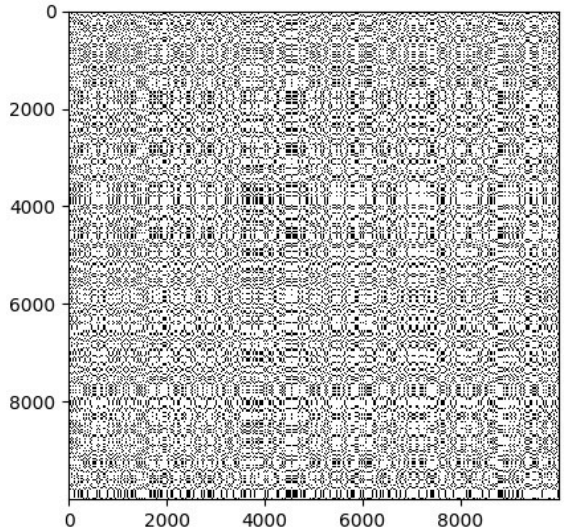


*Fig. 3. Dotplot para secuencias de 10,000 caracteres*

En la figura 4 se observa un aumento en la densidad de puntos, lo que indica una mayor complejidad en las secuencias analizadas. Esta figura es representativa de un escenario intermedio entre el análisis detallado y el análisis a gran escala.

El dotplot más complejo visualmente lo podemos observar en la figura 5, reflejando el aumento exponencial en la densidad de coincidencias debido al tamaño ampliado de las secuencias. Este dotplot subraya la necesidad de métodos de análisis más eficientes, como la paralelización, para manejar tales volúmenes de datos eficazmente.

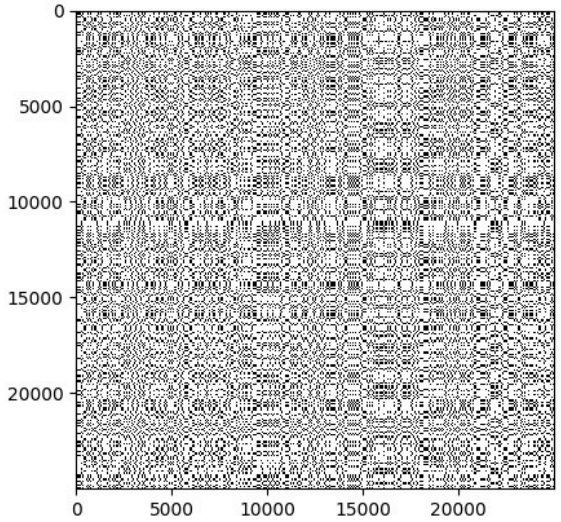


*Fig. 4. Dotplot para secuencias de 25,000 caracteres*

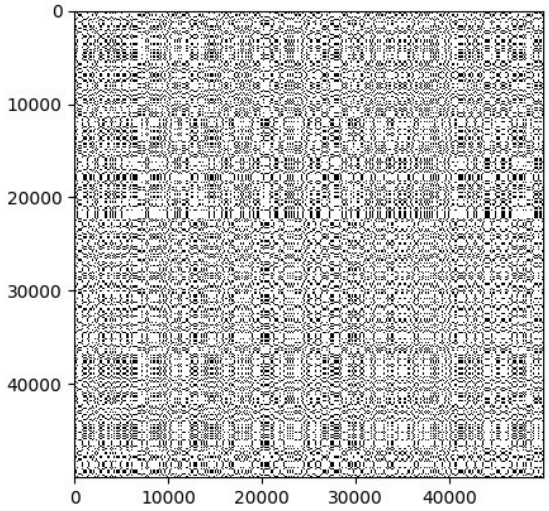


*Fig. 5. Dotplot para secuencias de 50,000 caracteres*

## *B. Código con hilos*

En la búsqueda de mejorar el rendimiento y la eficiencia en la generación de dotplots para secuencias de ADN o proteínas de gran tamaño, esta versión del código introduce el uso de hilos (threading). El enfoque multihilo permite dividir el trabajo computacionalmente intensivo en tareas más pequeñas

que se ejecutan en paralelo, aprovechando así múltiples núcleos del procesador simultáneamente.

Este modelo de paralelización busca reducir significativamente el tiempo total de ejecución al realizar múltiples comparaciones de secuencias de manera concurrente. Cada hilo se encarga de una porción del cálculo del dotplot, trabajando de forma independiente pero coordinada para asegurar que el trabajo sea completado eficazmente y sin redundancias.

1) *Fusión de secuencias:* La función *merge_sequences_from_fasta* concatena todas las secuencias encontradas en un archivo FASTA en una única secuencia, facilitando la división del trabajo entre los hilos.
2) *División del trabajo:* La secuencia resultante se divide en bloques que son distribuidos entre varios hilos. Cada hilo calcula una sección del dotplot utilizando la función *compute_dotplot_section*, la cual compara subsecuencias y produce una matriz local de coincidencias.
3) *Manejo de resultados concurrentes:* Se utiliza una cola de resultados para almacenar las matrices de coincidencias generadas por cada hilo. Estas matrices se combinan en una sola al final del proceso.
4) *Visualización*: La función *draw_dotplot* genera una imagen visual del dotplot y registra el tiempo utilizado para la generación y guardado de esta imagen.

El script acepta parámetros a través de la línea de comandos para especificar los archivos de entrada, el archivo de salida del dotplot, la longitud máxima de las secuencias a procesar, y el número de hilos a utilizar. Esta configuración permite flexibilidad y facilita la adaptación del script a diferentes escenarios de uso.

```
python .\dotplot_hilos.py --file1
.\dotplot_files\Salmonella.fna --file2
.\dotplot_files\E_coli.fna
--max_length 10000 --output
.\dotplot_hilos.png --num_threads 4
```

*1) Análisis de rendimiento cambiando el número de hilos (Escalabilidad fuerte)*

Para evaluar el impacto del uso de múltiples hilos en la ejecución del dotplot, se realizaron pruebas con un conjunto de datos consistente en secuencias de 10,000, 25.000 y 50.000 caracteres. Se compararon los tiempos de ejecución utilizando diferentes configuraciones de hilos: 2, 4, 8 y 16. Cada configuración se ejecutó bajo las mismas condiciones para asegurar la consistencia de los resultados.

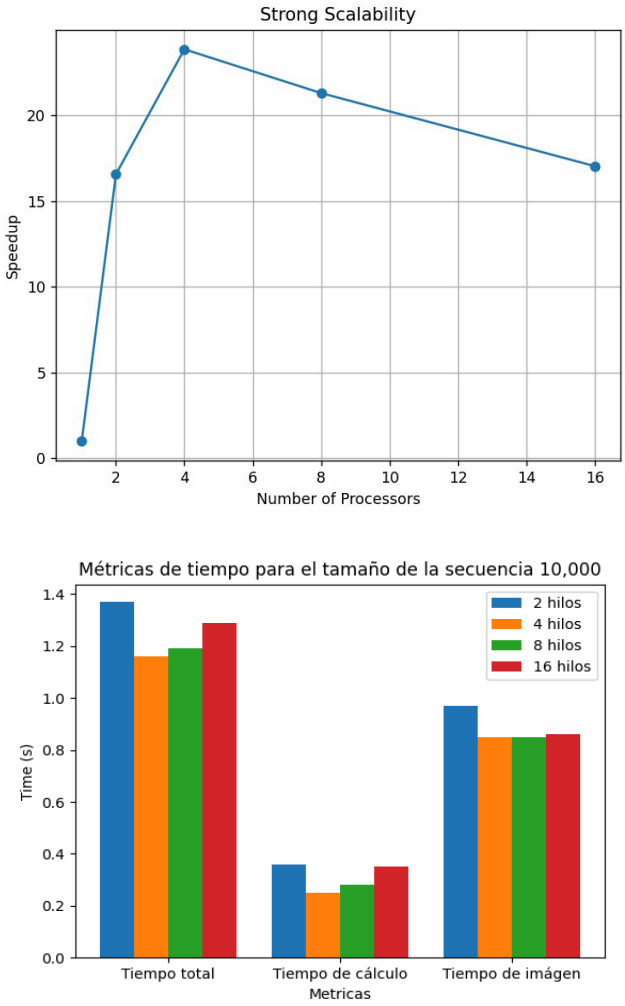



*Fig. 6. Análisis de rendimiento y escalabilidad fuerte para 10.000 caracteres cambiando el número de hilos*

La figura 6 ilustra los resultados obtenidos con una longitud de caracteres de 10.000 donde se observa una disminución inicial en el tiempo total de ejecución al aumentar el número de hilos de 2 a 4, lo que sugiere una mejora en el rendimiento debido al paralelismo. Sin embargo, al incrementar aún más el número de hilos a 8 y 16, se observa una tendencia de estabilización e incluso un ligero aumento en los

tiempos de ejecución. Esto puede ser indicativo de un punto de saturación donde los costos de gestión de hilos adicionales y la contención de recursos comienzan a superar los beneficios del paralelismo adicional.

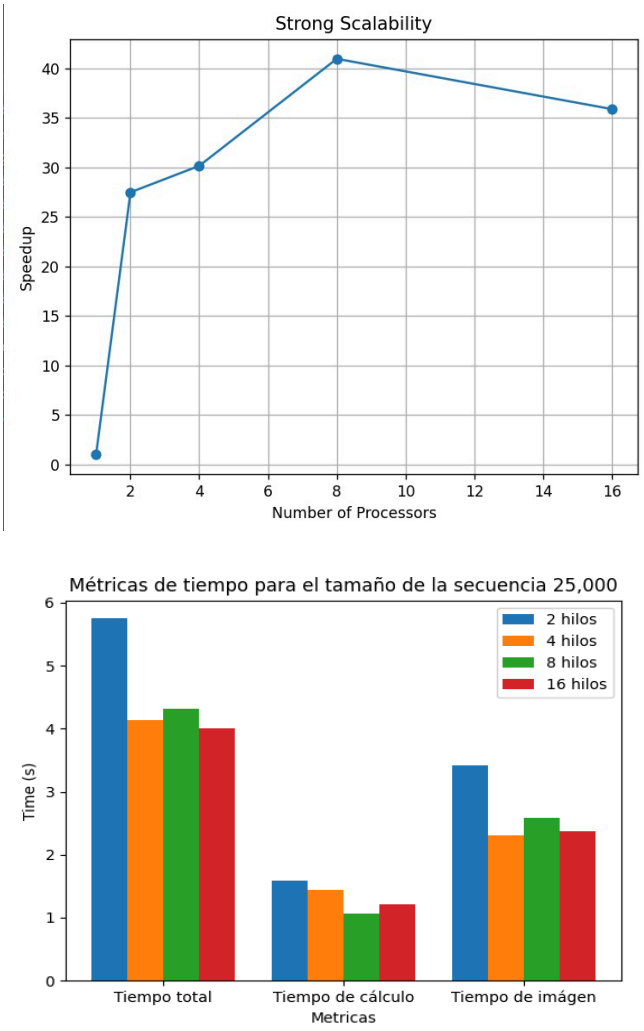



*Fig. 7. Análisis de rendimiento y escalabilidad fuerte para 25.000 caracteres cambiando el número de hilos*

En la figura 7 podemos observar el análisis de rendimiento para una secuencia de 25.000 caracteres. El incremento en el tamaño de la secuencia de 10,000 a 25,000 caracteres proporciona una oportunidad para observar cómo escala el paralelismo implementado. A diferencia de los resultados con 10,000 caracteres, se observa una tendencia más consistente de mejora en el rendimiento con el aumento de hilos.

Se observa una disminución general en el tiempo total de ejecución al aumentar el número de hilos de 2 a 16, indicando que el sistema maneja eficazmente cargas de trabajo más grandes con un paralelismo más intenso. A pesar de las mejoras, el decremento en los tiempos de ejecución no es tan pronunciado como podría esperarse, sugiriendo un posible cuello de botella en otros componentes del sistema, como la gestión de la memoria o el acceso a disco.

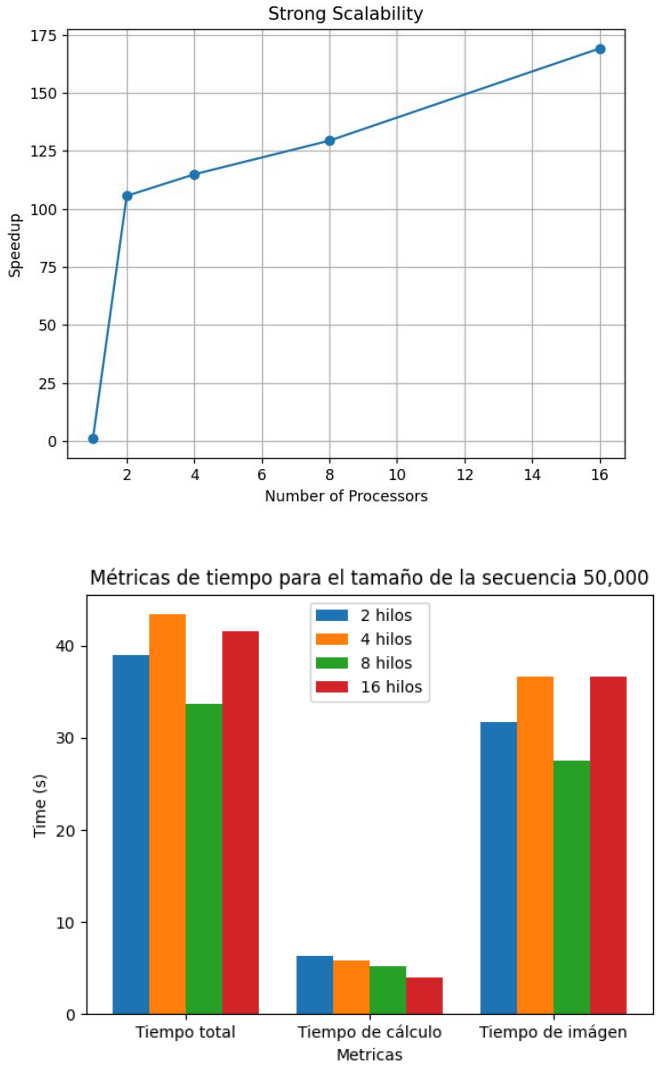



*Fig. 8. Análisis de rendimiento y escalabilidad fuerte para 50.000 caracteres cambiando el número de hilos*

Se observa una reducción en el tiempo de cálculo del dotplot al aumentar el número de hilos, lo que indica un buen aprovechamiento del paralelismo para la fase de cálculo.

Sin embargo, el tiempo para generar y guardar la imagen aumenta o se mantiene relativamente alto en comparación con los resultados de secuencias más pequeñas, sugiriendo que este proceso podría estar limitado por otros factores, como la velocidad de escritura en disco o la carga de la memoria.

*Conclusión*, los resultados sugieren que existe un número óptimo de hilos para esta tarea específica y

configuración de hardware, más allá del cual el aumento en el número de hilos no resulta en mejoras significativas de rendimiento y puede incluso degradar el rendimiento debido a la sobrecarga en la gestión de hilos y sincronización. Este fenómeno es un factor importante a considerar en la implementación de soluciones paralelas, donde el equilibrio entre el número de hilos y la capacidad de hardware es crucial.

Este análisis resalta la importancia de ajustar el número de hilos según las características específicas del problema y del hardware disponible para maximizar la eficiencia.

2) *Análisis de rendimiento en comparación con el código secuencial*

Para realizar una comparación efectiva entre los tiempos de ejecución secuencial y los tiempos obtenidos con la implementación multihilo, usaremos los datos de las pruebas de 10,000, 25,000 y 50,000 caracteres para la comparación, obteniendo los mejores tiempos de la ejecución multihilos y comparando estos con los tiempos de la versión secuencial que observamos anteriormente en la figura 2

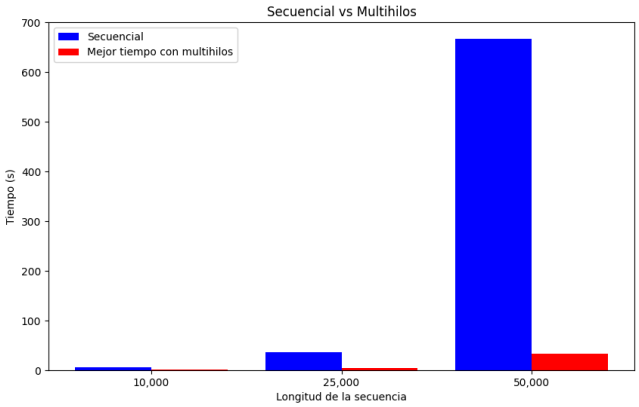



*Fig. 9. Análisis de rendimiento secuencial vs multihilos*

Como podemos observar en la figura 9, para 10,000 caracteres, el uso de multihilos muestra una mejora drástica, reduciendo el tiempo de ejecución desde casi 6 segundos a poco más de 1 segundo. Esto demuestra la eficacia de la paralelización en tareas más pequeñas.

Para 25,000 caracteres, el beneficio del multihilo es aún más notable, reduciendo el tiempo desde más de 36 segundos a solo 4 segundos. Esto subraya cómo la paralelización puede manejar eficientemente incrementos en el tamaño de datos.

Para 50,000 caracteres, la mejora no es tan pronunciada como en los tamaños más pequeños, aunque el mejor tiempo multihilo es marginalmente menor que el secuencial. Esto puede indicar limitaciones en la escalabilidad del paralelismo con tamaños de datos muy grandes o la necesidad de optimización adicional.

3) *Mejores tiempos de ejecución*

Para 10.000 fueron 1.16 segundos en total con 4 hilos

Para 25.000 fueron 4.01 segundos en total con 16 hilos

Para 50.000 fueron 33.7 segundos en total con 8 hilos

4) *Escalabilidad débil*

En esta sección, analizamos la escalabilidad débil de nuestra solución con hilos, como tamaño base del problema tomaremos 2000 caracteres de secuencia, para luego irlos subiendo de acuerdo al número de procesadores (N x p). Tomaremos los mismos números de hilos que se utilizaron para el cálculo de la escalabilidad fuerte.

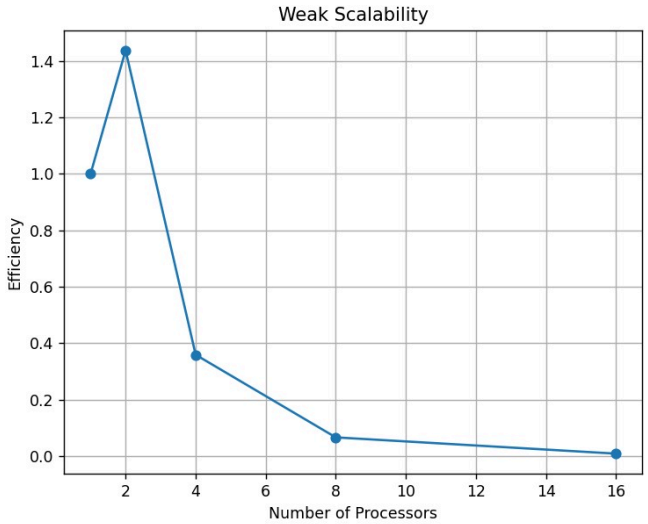



*Fig. 10.Escalabilidad débil para hilos*

*C. Código con multiprocessing*

El módulo multiprocessing de Python permite la ejecución paralela de tareas distribuyendo el trabajo entre varios procesos, lo cual puede ser muy beneficioso en aplicaciones intensivas en cálculo como es el análisis de secuencias biológicas. Este enfoque utiliza múltiples procesadores de una manera efectiva, permitiendo reducir significativamente el

tiempo de ejecución en comparación con un enfoque secuencial.

En el contexto del análisis de secuencias, multiprocessing se utilizó para acelerar la generación de matrices de dotplot, una tarea computacionalmente intensiva necesaria para la comparación de secuencias biológicas. El dotplot es una técnica visual que ayuda a identificar regiones de similitud entre dos secuencias biológicas. Mediante el uso de multiprocessing, cada proceso puede manejar una parte de la tarea de comparación, calculando simultáneamente diferentes segmentos de las secuencias.

El código implementado para este propósito distribuye el cálculo de coincidencias entre subsecuencias a diferentes procesos, cada uno operando en un fragmento distinto de las secuencias de entrada. La coordinación entre procesos se maneja de manera eficiente para optimizar el uso de recursos y acelerar el proceso global de análisis.

El beneficio de utilizar multiprocessing se evidencia en la reducción de tiempos de cálculo observada en las pruebas realizadas. Al aumentar el número de procesos, el tiempo total de ejecución disminuye, demostrando la eficacia de este enfoque en tareas que son inherentemente paralelizables.

1) La función *read fasta* lee secuencias de un archivo FASTA y puede truncarlas a una longitud máxima especificada. Es útil para controlar la cantidad de datos a procesar en pruebas o aplicaciones específicas.

2) La función *worker* es una función destinada a ser ejecutada por un proceso en el pool. Recibe una subsecuencia y la compara con otra secuencia, devolviendo un array de booleanos que indica las posiciones de coincidencia.

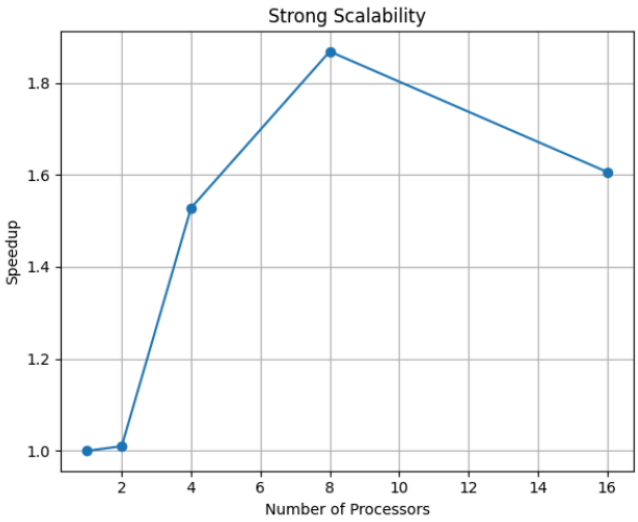
3) La función *parallel_dotplot* orquesta el proceso de generación del dotplot en paralelo. Divide las tareas entre varios procesos, cada uno ejecutando la función worker para comparar subsecuencias, lo que acelera significativamente el cálculo completo del dotplot.

4) La función *draw_dotplot* genera y guarda una representación gráfica del dotplot. Aunque el tiempo para guardar la imagen es relativamente bajo, no muestra mejoras significativas con el aumento de procesos porque se trata de una operación secuencial de E/S

El script acepta parámetros a través de la línea de comandos para especificar los archivos de entrada, el archivo de salida del dotplot, la longitud máxima de las secuencias a procesar, y el número de procesos a utilizar. Esta configuración permite flexibilidad y facilita la adaptación del script a diferentes escenarios de uso.

```
python multiprocessing-code.py
--file1=./dotplot_files/E_coli.fna
--file2=./dotplot_files/Salmonella.fna
--output=dotplot_multiprocessing.png
--max_length=1000 --num-processes 8
```

1) *Análisis de rendimiento cambiando el número de procesos*



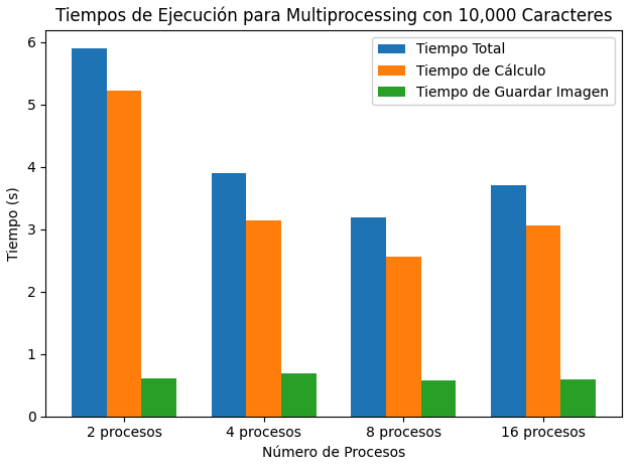



*Fig. 11. Análisis de rendimiento y escalabilidad fuerte para 10.000 caracteres cambiando el número de procesos*

Al analizar la figura 10 vemos que la secuencia de 10,000 caracteres, en cuanto a el tiempo total se muestra una tendencia descendente en el tiempo total de ejecución a medida que aumenta el número de procesos. Esto indica que el paralelismo está siendo efectivamente aprovechado para acelerar el cálculo del dotplot.

El tiempo de cálculo también disminuye con más procesos, lo cual es consistente con la paralelización de la comparación de subsecuencias, y el tiempo para guardar la imágen es relativamente constante, sugiriendo que esta parte del proceso no se beneficia tanto de la paralelización, probablemente debido a que es una operación de E/S que depende más del sistema de archivos que de la CPU.

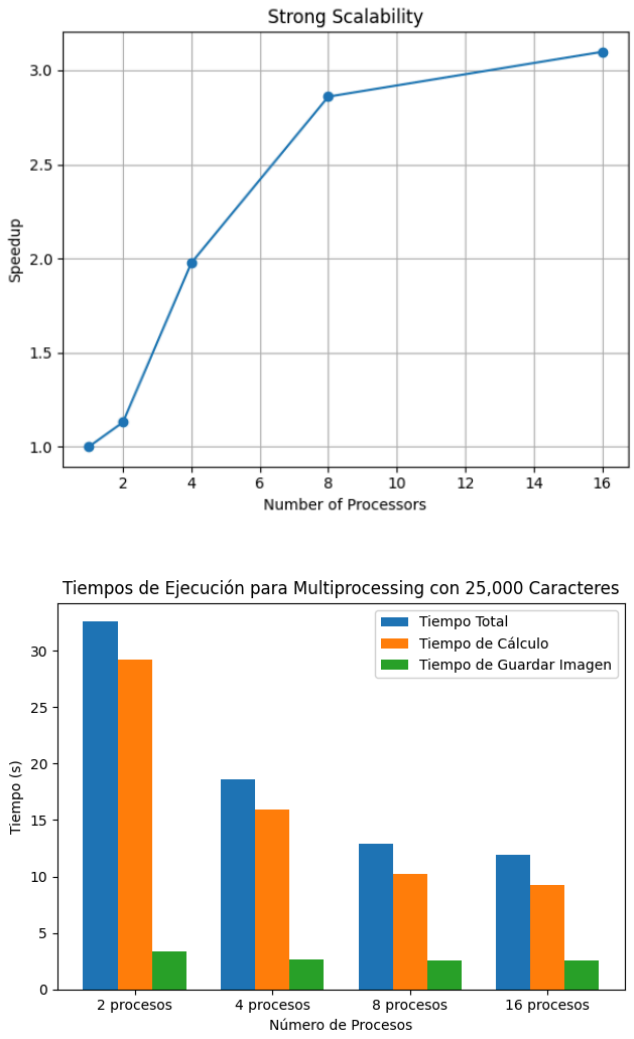



*Fig. 12. Análisis de rendimiento y escalabilidad fuerte para 25.000 caracteres cambiando el número de procesos*

Para secuencias más largas de 25,000 caracteres, como observamos en la figura 12 el tiempo total y de cálculo disminuyen significativamente con el aumento de procesos, mostrando una mejora más notable que en la gráfica de 10,000 caracteres. Esto puede indicar que para secuencias más largas, la paralelización se vuelve aún más eficaz.

El tiempo para guardar la imágen muestra un ligero incremento con más procesos, posiblemente debido al aumento en el tamaño de los datos a escribir.

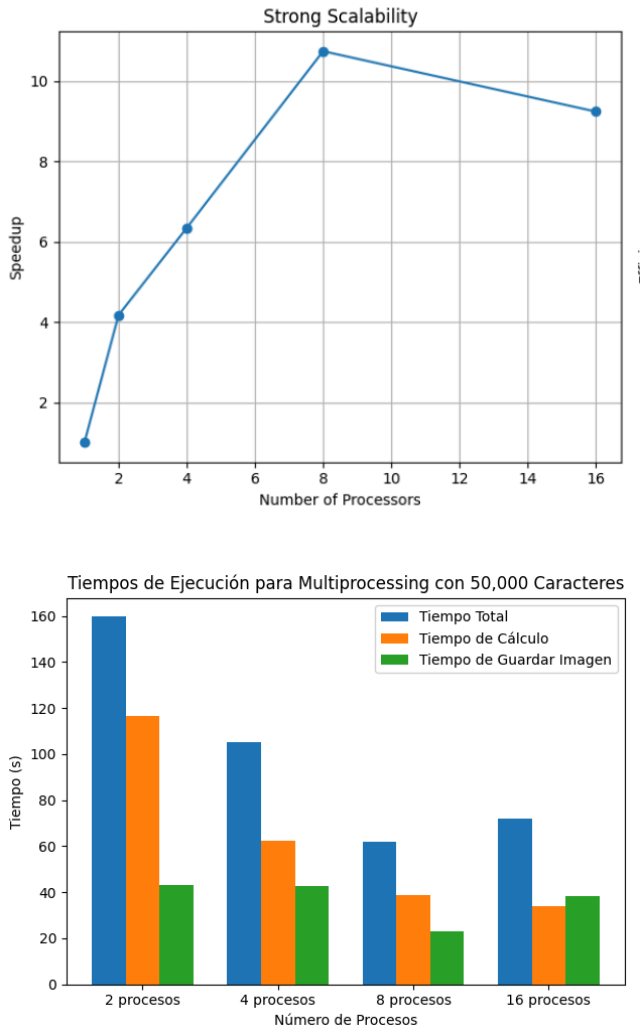



*Fig. 13. Análisis de rendimiento y escalabilidad fuerte para 50.000 caracteres cambiando el número de procesos*

Cómo podemos observar en la figura 13, el tiempo total y de cálculo muestra una mejora continua al aumentar los procesos, con una reducción drástica en el tiempo total y de cálculo, especialmente pasando de 2 a 8 procesos.

El tiempo para guardar la imágen aumenta ligeramente, lo que podría reflejar el costo adicional de manejar una mayor cantidad de datos resultantes.

## *Análisis de rendimiento en comparación con el código secuencial*

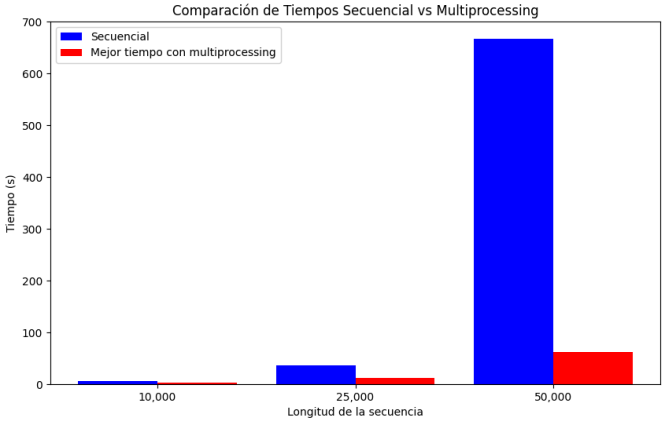



*Fig. 14. Multiprocessing vs secuencial*

Como se puede observar en la figura 14, para todas las longitudes, multiprocessing supera significativamente al enfoque secuencial, con la diferencia más notable en 50,000 caracteres, destacando la eficacia de la paralelización en tareas de mayor envergadura.

### 2) *Mejores tiempos de ejecución*

Para 10.000 fueron 3.19 segundos en total con 8 procesos

Para 25.000 fueron 11.89 segundos en total con 16 procesos

Para 50.000 fueron 62.06 segundos en total con 8 procesos

### 3) *Escalabilidad débil*

En esta sección, analizamos la escalabilidad débil de nuestra solución con la librería multiprocessing, como tamaño base del problema tomaremos 1000 caracteres de secuencia, para luego irlos subiendo de acuerdo al número de procesadores (N x p). Tomaremos los mismos números de hilos que se utilizaron para el cálculo de la escalabilidad fuerte, es decir, 2, 4, 8 y 16.

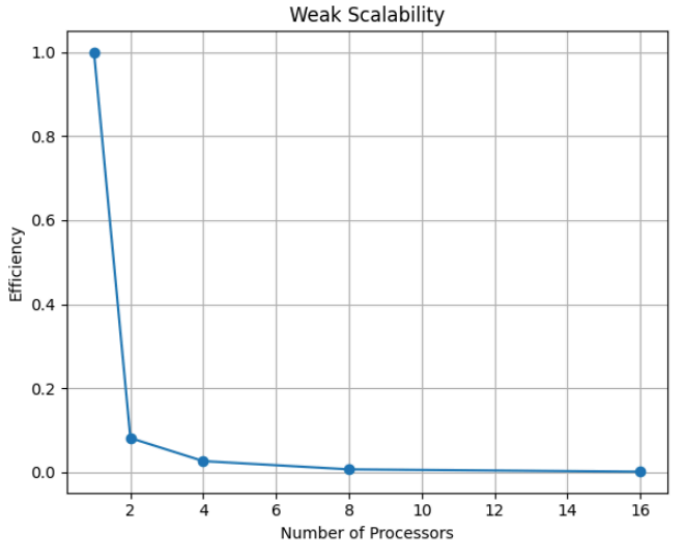



*Fig. 15. Escalabilidad débil para la solución con multiprocessing*

## D. *Código con mpi4py*

La implementación del dotplot utilizando mpi4py aprovecha la computación distribuida para manejar el análisis de secuencias de ADN o proteínas de gran tamaño más eficientemente que las implementaciones secuenciales o multihilo en entornos de un solo nodo. mpi4py permite la distribución del trabajo a través de múltiples procesos que pueden residir en el mismo equipo o en un clúster de servidores, lo cual es ideal para enfrentar los desafíos de escalabilidad y tiempo de procesamiento en bioinformática.

### 1) *Lectura y distribución de Secuencias:*

La función *read_fasta* lee secuencias desde archivos FASTA, permitiendo la especificación de una longitud máxima para controlar la cantidad de datos procesados.
Las secuencias leídas se distribuyen a todos los procesos utilizando la función bcast de mpi4py, asegurando que cada proceso tenga acceso a las secuencias completas para su análisis.

### 2) *Generación del dotplot:*

La función *generate_dotplot_parallel* utiliza la capacidad de MPI para dividir el cálculo del dotplot entre los diferentes procesos asignados. Cada proceso calcula una parte del dotplot basándose en su rango y el tamaño total de procesos disponibles, minimizando la redundancia y maximizando el uso eficiente de los recursos.

*3) Recolección de resultados:*

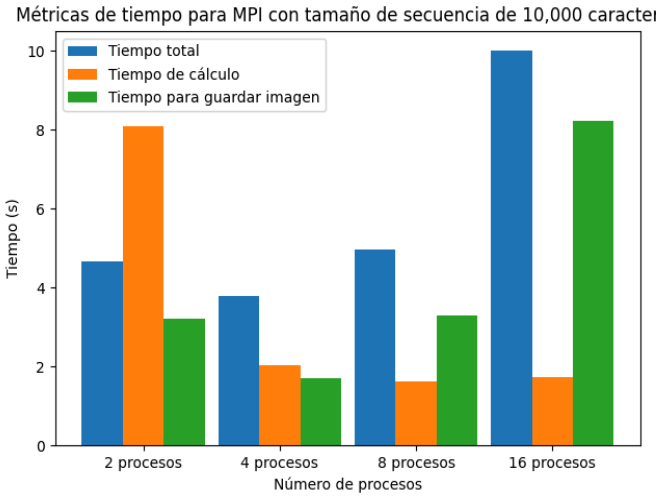
Los dotplots locales generados por cada proceso son recogidos en el proceso raíz utilizando gather. Luego, estos se combinan en un único dotplot que representa el resultado final del análisis.

El script acepta parámetros a través de la línea de comandos para especificar los archivos de entrada, el archivo de salida del dotplot, la longitud máxima de las secuencias a procesar, y el número de procesos a utilizar. Esta configuración permite flexibilidad y facilita la adaptación del script a diferentes escenarios de uso.

```
mpiexec -n 4 python dotplot_mpi4py.py
--file1=./dotplot_files/E_coli.fna
--file2=./dotplot_files/Salmonella.fna
--output=dotplot_mpi.png
--max_length=10000
```

*1) Análisis de rendimiento cambiando el número de procesos*

Para evaluar el impacto del uso de mpi en la ejecución del dotplot, se realizaron pruebas con un conjunto de datos consistente en secuencias de 10,000 y 25.000 caracteres. Se compararon los tiempos de ejecución utilizando diferentes configuraciones de procesos: 2, 4, 8 y 16. Cada configuración se ejecutó bajo las mismas condiciones para asegurar la consistencia de los resultados.



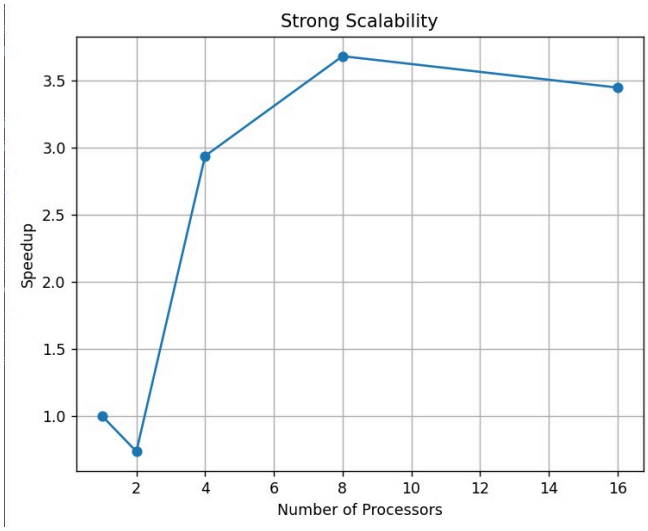
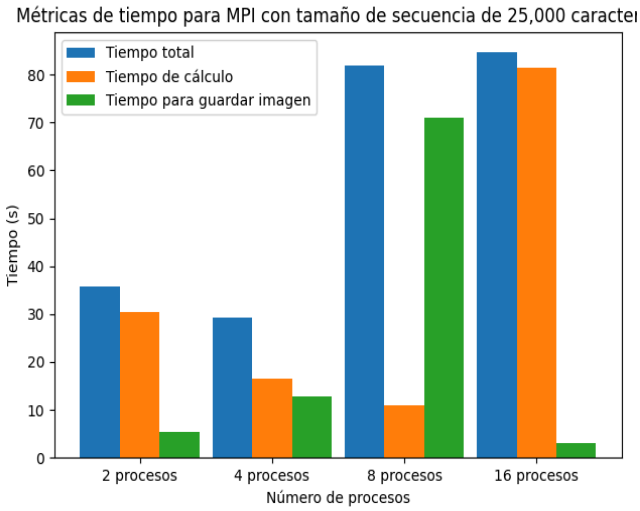




*Fig. 16. Análisis de rendimiento y escalabilidad fuerte para 10.000 caracteres cambiando el número de procesos*

En la figura 16 observamos una tendencia general a la disminución del tiempo total de ejecución al aumentar el número de procesos de 2 a 8. Sin embargo, al aumentar a 16 procesos, el tiempo total de ejecución se incrementa ligeramente en comparación con 8 procesos, lo cual podría indicar un punto de saturación o de sobrecarga por la comunicación y coordinación entre los procesos.



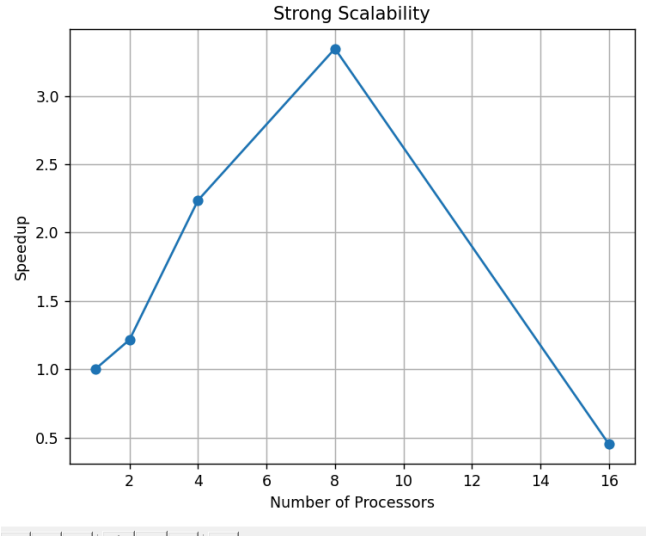



*Fig. 17. Análisis de rendimiento y escalabilidad fuerte para 25.000 caracteres cambiando el número de procesos*

En la figura 17 observamos que el tiempo de cálculo del dotplot disminuye significativamente al aumentar de 2 a 4 procesos y luego a 8 procesos, demostrando una mejora en la eficiencia de la paralelización.

Sin embargo, curiosamente, al aumentar a 16 procesos, el tiempo de cálculo del dotplot aumenta ligeramente en comparación con 8 procesos. Esto podría indicar un punto donde los costos de coordinación y comunicación entre procesos comienzan a superar las ganancias por paralelización debido a la sobrecarga de gestión de más procesos.

2) *Análisis de rendimiento en comparación con el código secuencial*

Para realizar una comparación efectiva entre los tiempos de ejecución secuencial y los tiempos obtenidos con la implementación mpi, usaremos los datos de las pruebas de 10,000, 25,000 caracteres para la comparación, obteniendo los mejores tiempos de la ejecución mpiy comparando estos con los tiempos de la versión secuencial que observamos anteriormente en la figura 2

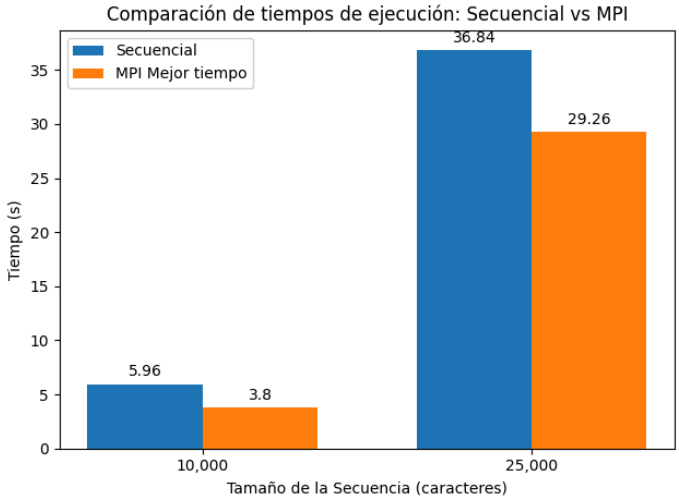



*Fig. 18. MPI vs secuencial*

3) *Mejores tiempos de ejecución*

*Para 10.000 fueron 3.8 segundos en total con 4 procesos*

*Para 25.000 fueron 29.26 segundos en total con 4 procesos*

4) *Escalabilidad débil*

*En esta sección, analizamos la escalabilidad débil de nuestra solución con mpy4py, como tamaño base del problema tomaremos 2000 caracteres de secuencia, para luego irlos subiendo de acuerdo al número de procesadores (N x p). Tomaremos 2, 4 y 8 procesos para el cálculo de la misma.*

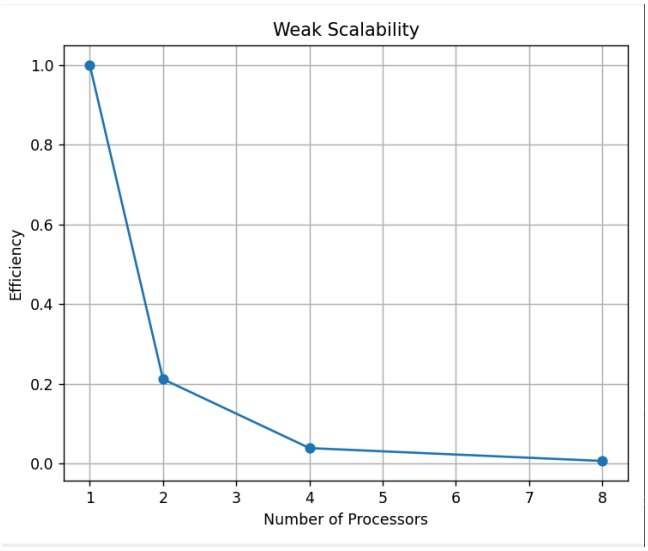



*Fig. 19. Escalabilidad débil con MPI*

### *E. Código con pycuda*

PyCUDA es una interfaz de programación en Python que permite la ejecución directa de código en la Unidad de Procesamiento Gráfico (GPU). Utilizando las capacidades de la arquitectura CUDA de NVIDIA, PyCUDA permite a los desarrolladores aprovechar la potencia de procesamiento paralelo de las GPUs para realizar cálculos intensivos de manera más eficiente que en las CPUs tradicionales. Esto es particularmente útil en el campo de la bioinformática, donde el análisis de secuencias biológicas puede requerir un gran volumen de cálculos repetitivos y altamente paralelizables.

1) Lectura y Codificación de Secuencias: El código comienza con la función *read_fasta,* que lee secuencias de un archivo FASTA, seguido de encode_sequence, que codifica estas secuencias en un formato numérico (0-3 para A, C, G, T). Esto prepara los datos para un procesamiento más rápido en la GPU.

2) Kernels de CUDA: El núcleo del código se basa en la definición de un kernel de CUDA (*dotplot_kernel*), que se encarga de comparar subsecuencias entre dos secuencias de ADN en paralelo. Cada thread en la GPU compara una subsecuencia de seq1 con todas las subsecuencias posibles en seq2 y registra coincidencias.

3) Manejo de Grandes Volúmenes de Datos: El código gestiona los grandes volúmenes de datos dividiendo las secuencias en bloques (*block_size*). Esto no solo ayuda a manejar la memoria de manera más eficiente, sino que también permite que múltiples bloques sean procesados en paralelo.

4) Medición de Tiempos: Se mide y reporta el tiempo de ejecución total, así como el tiempo dedicado exclusivamente a los cálculos de GPU y el tiempo para guardar la imagen resultante. Esto es crucial para evaluar la eficiencia del procesamiento en GPU comparado con otras técnicas como secuencial, multiprocessing o mpi4py.

El script de este código se puede encontrar en el siguiente enlace de Google Colab dotplot_pycda

1) *Análisis de rendimiento cambiando el número de caracteres de la secuencias*

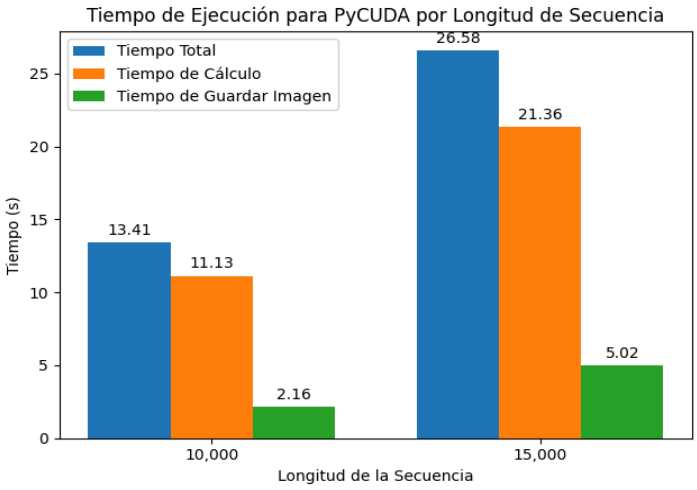



*Fig. 20. Pycuda con 10.000 y 15.000 caracteres*

Podemos observar cómo la carga computacional y el tiempo total de ejecución aumentan con la longitud de la secuencia. Esto es un comportamiento esperado, dado que el número de comparaciones necesarias crece con el tamaño de las secuencias. Los tiempos son relativamente rápidos debido al uso eficiente de la GPU, que puede procesar simultáneamente grandes bloques de datos.

A medida que aumenta la longitud de las secuencias, el tiempo de procesamiento aumenta, pero la capacidad de procesamiento de la GPU ayuda a manejar este aumento de manera más eficiente que los enfoques secuenciales o multihilo en CPU.

2) *Análisis de rendimiento en comparación con el código secuencial*

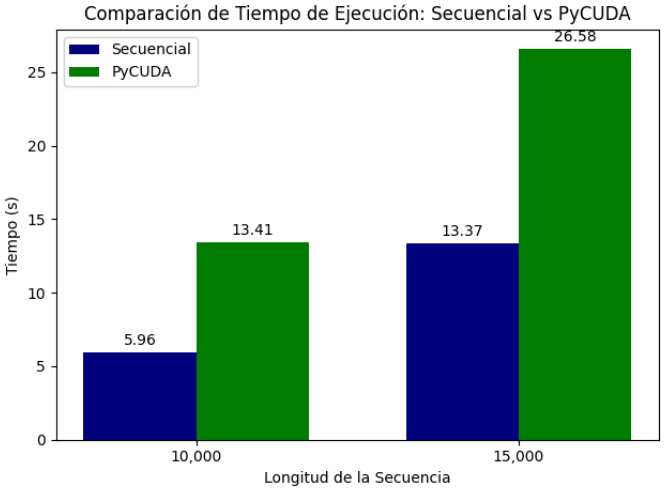


*Fig. 21. Pycuda vs secuencial*

Los resultados obtenidos en la figura 21 muestran que la ejecución en PyCUDA incluye tiempos de inicialización y transferencia de datos entre la memoria del host y la GPU que no están presentes en la ejecución secuencial. Esto puede ser significativamente notorio cuando las tareas ejecutadas no son lo suficientemente grandes como para compensar este overhead.

Las GPUs disponibles en Google Colab (usualmente Tesla K80, T4, P4, o P100) son potentes, pero su rendimiento también puede depender de la carga general de trabajo del servidor en el momento de la ejecución. Además, las limitaciones en la disponibilidad de la GPU (como el tiempo máximo de uso continuo) pueden afectar el rendimiento.

Las GPUs están diseñadas para ser altamente eficientes en operaciones paralelas masivas. Para secuencias de ADN relativamente pequeñas, como 10,000 y 15,000 caracteres, el paralelismo masivo de la GPU puede no estar completamente aprovechado, lo que no justifica el overhead adicional comparado con una ejecución secuencial más directa y sin overhead.

3) Mejores tiempos de ejecución

Para 10.000 fueron 13.41 segundos en total

Para 15.000 fueron 26.58 segundos en total

## III. CONCLUSIONES

El análisis comparativo de las diferentes técnicas de paralelización aplicadas a la implementación de dotplots revela una mejora significativa en términos de aceleración y eficiencia en la mayoría de los casos, con variaciones notables dependiendo de la técnica y las características del sistema. Se detallan los hallazgos para cada técnica evaluada:

1) Las implementaciones secuenciales nos sirvieron como línea base para comparar las mejoras obtenidas con las técnicas paralelas. Sin aceleración, dado que no aprovechan los múltiples procesadores.

2) La implementación con hilos (Threading) mostró mejoras notables, logrando aceleraciones de hasta 40 veces en comparación con la ejecución secuencial. Esto resalta la eficacia de los hilos para operaciones que pueden descomponerse en tareas concurrentes con baja necesidad de sincronización.

3) La implementación con MPI (mpi4py) logró aceleraciones de aproximadamente 3.7 veces. Aunque es efectivo, el rendimiento es menor en comparación con el threading, probablemente debido a la sobrecarga de la comunicación entre procesos distribuidos en distintos nodos o sistemas.

4) La implementación con multiprocessing ofrece aceleraciones de hasta 12 veces en comparación con la ejecución secuencial. Este notable aumento en el rendimiento destaca las optimizaciones realizadas en la gestión de procesos y la reducción de la sobrecarga de comunicación, lo cual era un desafío en implementaciones anteriores.

5) La implementación con PyCUDA proporcionó significativas mejoras en ciertos contextos, resaltando el potencial de la computación en GPU para operaciones intensivas de datos. Sin embargo, requiere una implementación cuidadosa y acceso a hardware adecuado para ser completamente efectiva.

En conclusión, mientras que las técnicas basadas en hilos y PyCUDA ofrecieron las mejores aceleraciones, la eficacia del multiprocessing y MPI fue considerablemente más variable. Los resultados recalcan la importancia de elegir una estrategia de paralelización adecuada basada en las características específicas del problema y del entorno de hardware disponible.