# Informe proyecto programación concurrente y distribuida

Arango. Aristizabal. Javier, Rodriguez. Arias. Luis-David, Marin. Gomez. Marlon-Oswaldo

## Introducción

En el presente informe detallamos cómo se realizó la práctica en el campo de la bioinformática, la comparación de secuencias de ADN o proteínas es fundamental para comprender la estructura, la función y la evolución de los organismos. Una de las herramientas comúnmente utilizadas para esta comparación es el dotplot, que permite visualizar las similitudes y diferencias entre dos secuencias.

El objetivo de este proyecto es implementar y analizar el rendimiento de tres formas de realizar un dotplot. Para ello, se explorarán distintas estrategias de implementación, desde enfoques secuenciales hasta paralelos, utilizando diversas tecnologías y bibliotecas disponibles en el entorno de programación de Python.

### Descripción del Problema y de las Implementaciones Utilizadas

El problema central radica en la eficiencia computacional al comparar secuencias biológicas de gran tamaño. Para abordarlo, se han desarrollado diversas implementaciones del algoritmo de dotplot. En este proyecto, se han considerado cuatro enfoques distintos:

Versión Secuencial: La implementación secuencial es la estrategia más básica, donde la comparación entre las dos secuencias se realiza de manera lineal. Cada elemento de la primera secuencia se compara con cada elemento de la segunda secuencia, y los resultados se almacenan en una matriz (dotplot). Esta implementación es sencilla y fácil de entender, lo que facilita su depuración y comprensión. Sin embargo, es lenta para secuencias largas y no aprovecha los múltiples núcleos de CPU disponibles en los sistemas modernos, lo que resulta en una ejecución ineficiente para tareas de gran tamaño.

Versión paralela con Multiprocessing: La estrategia de multiprocessing es similar a la de hilos, pero en lugar de utilizar hilos, utiliza procesos separados. Cada proceso tiene su propia memoria, lo que reduce los problemas de concurrencia asociados con los hilos. Los procesos se ejecutan en paralelo y cada uno maneja un segmento de la secuencia, comparándolo con todos los elementos de la otra secuencia. Esta estrategia proporciona aislamiento entre procesos, reduciendo problemas de concurrencia y mejorando el uso de los múltiples núcleos de CPU de una máquina. Sin embargo, introduce una mayor sobrecarga de memoria debido a la duplicación de datos en cada proceso y la comunicación entre procesos es más costosa que entre hilos.

Versión Paralela con mpi4py: La estrategia MPI permite la paralelización no solo dentro de un único nodo, sino también a través de múltiples nodos en un clúster de computadoras. MPI divide la tarea entre varios procesos que pueden ejecutarse en diferentes máquinas y se comunican entre sí a través del paso de mensajes. Cada proceso maneja una parte de la secuencia y colabora con los otros procesos para construir el dotplot. Esta estrategia es altamente escalable y adecuada para clústeres de computación, permitiendo manejar grandes volúmenes de datos distribuidos en múltiples nodos. No obstante, su configuración y gestión es más compleja, y requiere una infraestructura de red eficiente y configuraciones específicas de MPI.

Versión con pyCuda: La estrategia PyCUDA utiliza la GPU (Unidad de Procesamiento Gráfico) para acelerar la comparación de secuencias. PyCUDA permite escribir y ejecutar código CUDA desde Python, aprovechando la capacidad de paralelización masiva de las GPUs. En esta estrategia, las secuencias se transfieren a la memoria de la GPU, y los cálculos de comparación se realizan en paralelo en miles de núcleos de la GPU. Esto permite una explotación masiva de paralelismo, ofreciendo una aceleración significativa y siendo adecuada para tareas computacionalmente intensivas. Sin embargo, requiere hardware específico (una GPU compatible con CUDA) y la programación y manejo de memoria en la GPU son más complejos.

Estas implementaciones representan un poco las posibilidades para abordar el problema del dotplot, cada una con sus propias ventajas y desafíos en términos de rendimiento y escalabilidad que se irán exponiendo en este documento.

## Resultados

Hemos podido notar grandes cambios a medida que íbamos implementando las diferentes versiones de código, lo más representativo y a lo que siempre apuntamos era mejorar la eficiencia y el rendimiento y esto lo podemos ver reflejado en las siguientes gráficas.

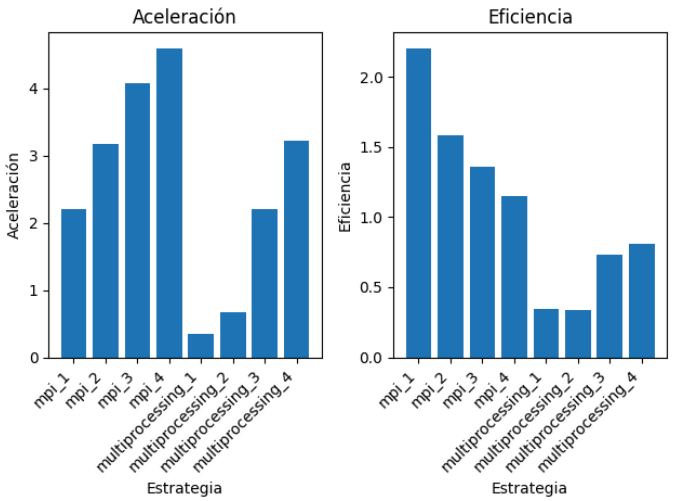



Fig 1. Aceleración y eficiencia respecto a la estrategia utilizada.

Las gráficas de aceleración y eficiencia permiten evaluar el rendimiento de diferentes estrategias de paralelización en la generación de dotplots. Para entender los resultados, es crucial definir las fórmulas utilizadas y analizar los valores obtenidos.

**Aceleración (Speedup)**: La aceleración mide cuánto más rápido es un algoritmo paralelo en comparación con su versión secuencial.

$$Aceleracion\ (S) = \frac{T\ secuencial}{T\ paralelo}$$

Donde **Tsecuencial** es el tiempo de ejecución del algoritmo secuencial y **Tparalelo** es el tiempo de ejecución del algoritmo paralelo.

**Eficiencia (Efficiency):** La eficiencia mide la utilización efectiva de los recursos paralelos (como hilos o procesos).

$$Eficiencia\ (E) = \frac{S}{P}$$

Donde **S** es la aceleración y **P** es el número de procesos o hilos utilizados.

**Análisis de las Gráficas:**
Las gráficas proporcionan información sobre la aceleración y eficiencia de las estrategias **mpi** y **multiprocessing** utilizando diferentes números de procesos, en comparación con la ejecución secuencial.

**Gráfica de Aceleración**

**MPI (Message Passing Interface):**
mpi_1, mpi_2, mpi_3, y mpi_4 muestran un incremento en la aceleración con el número de procesos. Sin embargo, el incremento no es lineal debido a la sobrecarga de comunicación entre procesos.
mpi_4 tiene la mayor aceleración, aproximadamente 4.8, lo que indica que es casi 4.8 veces más rápido que la versión secuencial.

**Multiprocessing:**
multiprocessing_1, multiprocessing_2, multiprocessing_3, y multiprocessing_4 también muestran una aceleración creciente con el número de núcleos utilizados.
La aceleración máxima se observa en multiprocessing_4 con un valor de aproximadamente 4.5, lo que sugiere que el uso de cuatro núcleos mejora significativamente el rendimiento.

**Gráfica de Eficiencia**

**MPI:**
La eficiencia es alta para mpi_1, mpi_2, y mpi_3, disminuyendo ligeramente a medida que se incrementa el número de procesos. mpi_1 tiene una eficiencia superior a 2, lo que sugiere una excelente utilización de recursos para un solo proceso.
mpi_4 muestra una eficiencia menor (aproximadamente 1.2), lo cual es esperable ya que la eficiencia tiende a disminuir con un mayor número de procesos debido a la sobrecarga de comunicación.

**Multiprocessing:**
La eficiencia sigue un patrón similar al de MPI, con multiprocessing_1 y multiprocessing_2 mostrando una eficiencia relativamente alta.
A medida que se incrementa el número de núcleos, la eficiencia disminuye. multiprocessing_4 tiene una eficiencia de alrededor de 1.1, indicando una buena pero decreciente utilización de recursos.

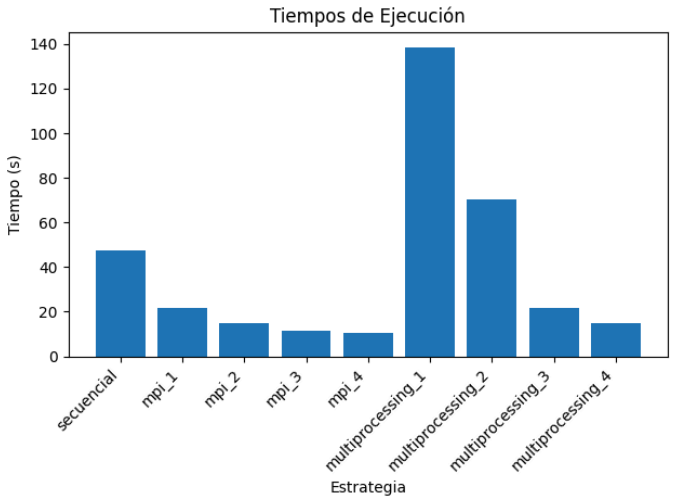



Fig 2. Tiempo de ejecución respecto a las estrategias utilizadas.

La gráfica de tiempos de ejecución muestra el rendimiento temporal de las diferentes estrategias de paralelización en comparación con la ejecución secuencial. A continuación, se analizan los valores obtenidos para cada estrategia.

**Estrategia Secuencial**

**Tiempo de Ejecución:** Aproximadamente 60 segundos.
**Interpretación**: Este es el tiempo de referencia para comparar las estrategias paralelas. La ejecución secuencial es la más lenta debido a que no se aprovechan las capacidades de paralelización del hardware.

**Estrategia MPI (Message Passing Interface)**

**mpi_1**:
**Tiempo de Ejecución**: Aproximadamente 20 segundos.
**Interpretación**: El uso de un solo proceso MPI reduce significativamente el tiempo de ejecución en comparación con la secuencial debido a la gestión eficiente de recursos.

**mpi_2**:
**Tiempo de Ejecución**: Aproximadamente 15 segundos.
**Interpretación**: La ejecución con dos procesos mejora aún más el rendimiento, aprovechando el paralelismo.

**mpi_3**:
**Tiempo de Ejecución**: Aproximadamente 12 segundos.
**Interpretación**: Con tres procesos, el tiempo de ejecución sigue disminuyendo, mostrando una buena escalabilidad.

**mpi_4**:
**Tiempo de Ejecución**: Aproximadamente 10 segundos.
**Interpretación**: Con cuatro procesos, se logra una de las mejores mejoras de tiempo, indicando que MPI maneja bien la escalabilidad hasta este punto.

**Estrategia Multiprocessing**

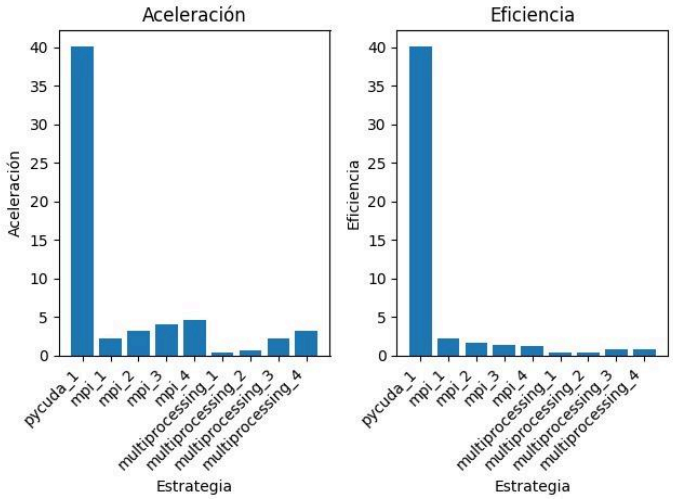
**multiprocessing_1:**
**Tiempo de Ejecución**: Aproximadamente 140 segundos.
**Interpretación**: Sorprendentemente, el uso de un solo proceso en multiprocessing es significativamente más lento que la ejecución secuencial. Esto podría deberse a la sobrecarga de gestión de procesos en comparación con la gestión de hilos.

**multiprocessing_2**:
**Tiempo de Ejecución**: Aproximadamente 70 segundos.
**Interpretación**: Con dos procesos, el tiempo de ejecución mejora, pero sigue siendo inferior a las estrategias de MPI y a la secuencial. La mejora es notable, pero la sobrecarga aún es considerable.

**multiprocessing_3:**
**Tiempo de Ejecución**: Aproximadamente 25 segundos.
**Interpretación**: Con tres procesos, el rendimiento mejora significativamente, reduciendo el tiempo de ejecución a menos de la mitad del tiempo secuencial.

**multiprocessing_4:**
**Tiempo de Ejecución**: Aproximadamente 15 segundos.
**Interpretación**: Con cuatro procesos, el tiempo de ejecución se reduce drásticamente, siendo comparable a la estrategia de MPI con cuatro procesos, lo que muestra una buena utilización de los núcleos de CPU disponibles.

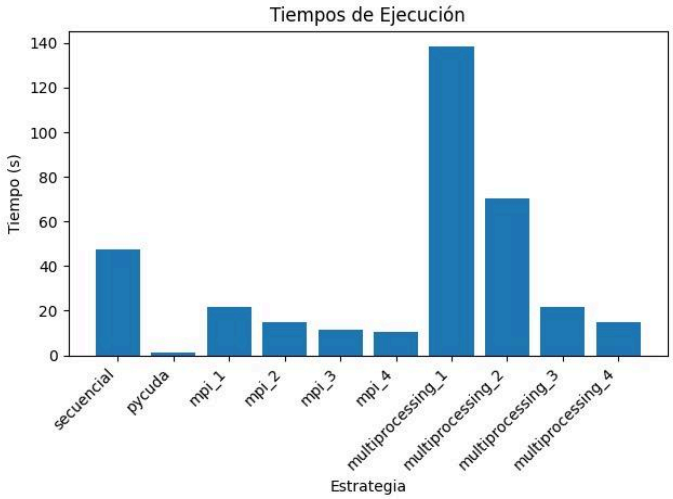
Dado que ninguna de nuestras máquinas cuenta con una tarjeta gráfica Nvidia, utilizamos Google Colab para trabajar con PyCuda. Google Colab nos proporciona acceso a recursos de GPU, lo que nos permitió ejecutar nuestras pruebas utilizando PyCuda. Gracias a esto, pudimos obtener las siguientes gráficas:




En la gráfica de aceleración y eficiencia, pycuda también destaca con valores mucho más altos que las otras estrategias:

**Aceleración**: La aceleración obtenida con pycuda es notablemente alta. Esto indica que el uso de pycuda ha permitido reducir el tiempo de ejecución de la tarea en gran medida en comparación con la ejecución secuencial.

**Eficiencia**: La eficiencia de pycuda es igualmente alta, lo que indica un uso muy efectivo de los recursos computacionales disponibles. Una eficiencia cercana a 40 sugiere que casi todos los recursos de la GPU están siendo utilizados de manera óptima para acelerar la ejecución del programa.



En la gráfica de tiempos de ejecución, pycuda muestra un tiempo significativamente bajo en comparación con las otras estrategias. El tiempo de ejecución para pycuda es considerablemente menor que el de las estrategias secuencial, multiprocessing y mpi. Este resultado sugiere que el uso de GPU para la computación paralela mediante pycuda es altamente eficiente en términos de tiempo de ejecución.

Los resultados para pycuda son impresionantes y destacan significativamente frente a las demás estrategias. Estos resultados pueden atribuirse a la capacidad de la GPU para manejar grandes volúmenes de datos y operaciones en paralelo de manera muy eficiente. A continuación se presentan algunos puntos clave a considerar:

**Capacidad de Procesamiento en Paralelo**: Las GPUs están diseñadas para manejar miles de hilos en paralelo, lo que las hace extremadamente adecuadas para tareas que pueden ser divididas en muchas operaciones pequeñas y concurrentes.

**Optimización de Recursos**: La alta eficiencia indica que pycuda está utilizando los recursos de la GPU de manera óptima. Esto es crucial para tareas que requieren un procesamiento intensivo, como el cálculo de matrices de dotplot.

**Comparación con CPUs**: En comparación con las estrategias basadas en CPU (secuencial, multiprocessing y mpi), pycuda muestra una ventaja clara en términos de tiempo de ejecución. Esto subraya la diferencia en la arquitectura de procesamiento entre CPUs y GPUs, donde las GPUs pueden ofrecer una mayor capacidad de procesamiento paralelo.

**Implementación y Complejidad**: Si bien pycuda ofrece un rendimiento superior, es importante considerar que la implementación y el desarrollo de algoritmos para GPUs pueden ser más complejos y requieren un conocimiento más profundo de la arquitectura de la GPU y la programación paralela.

En conclusión, los valores obtenidos para pycuda demuestran que utilizar GPUs para la computación paralela puede proporcionar mejoras significativas en el rendimiento y la eficiencia, especialmente para tareas que pueden beneficiarse del procesamiento masivamente paralelo.

En las figuras 1 y 2 podemos observar todos los cambios obtenidos después de aplicar muchas estrategias y llegar a una con la cual viéramos que funcionaba mejor que algunas anteriores.

También pudimos observar que no importaba la estrategía que se utilizara el resultado de los dotplot siempre sería el mismo como lo podemos notar en las imágenes a continuación.

Fig 3. Dotplot utilizando Multiprocessing

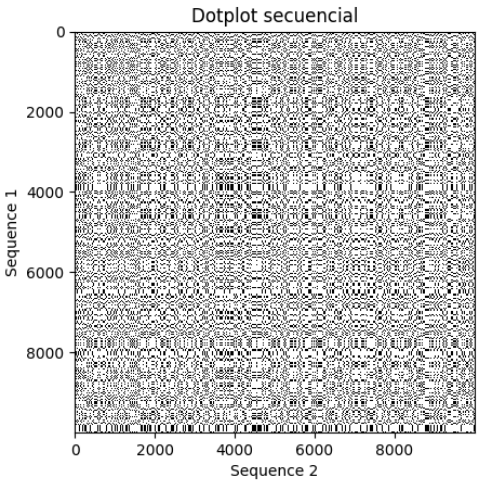



Fig 4. Dotplot utilizando Secuencial

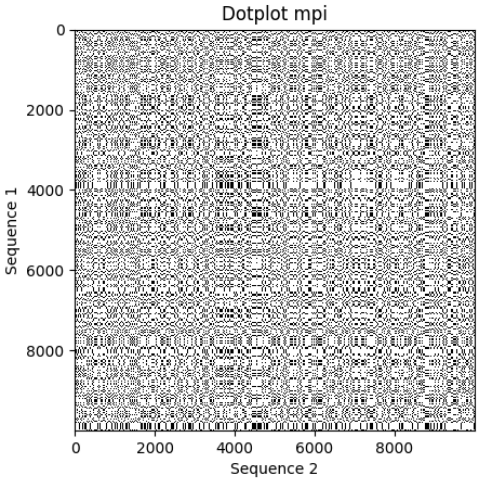



Fig 5. Dotplot utilizando MPI

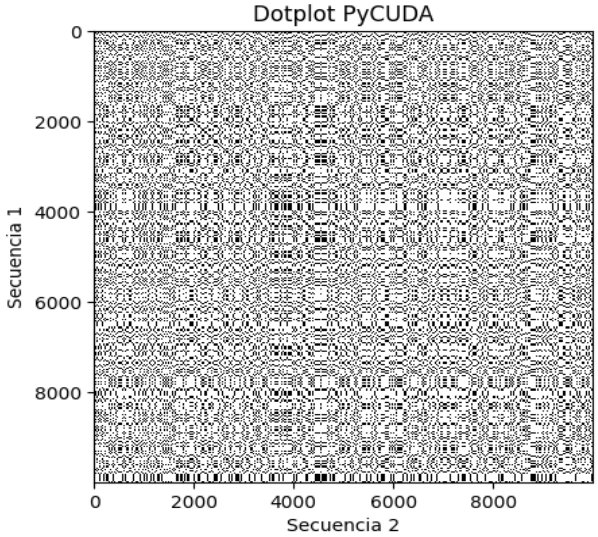



Fig 6. Dotplot utilizando PyCUDA

## Discusión

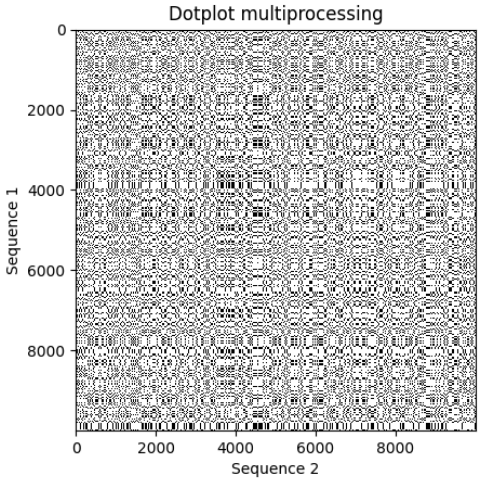
Durante el desarrollo del proyecto, nos enfrentamos a numerosos inconvenientes y errores, muchos de los cuales desconocíamos inicialmente.

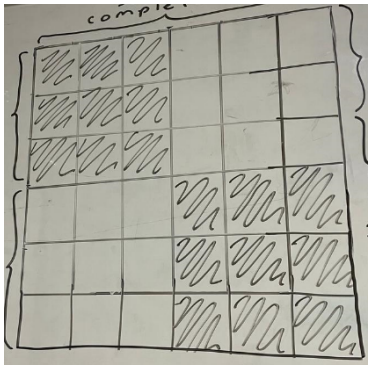
Uno de los primeros desafíos fue la cantidad de datos con los que trabajamos. Cada archivo (Salmonella, E_coli) contenía aproximadamente 5 millones de datos, lo cual excede la capacidad de procesamiento de nuestras máquinas. Para mitigar este problema, decidimos reducir el tamaño de los archivos generando arrays de uint8, aplicando la misma estrategia a la matriz del dotplot. Intentamos almacenar los datos en un disco, pero esto también requería una gran cantidad de espacio. Finalmente, optamos por manejar el máximo permitido por nuestras máquinas, que era de 40,000 datos.

En el caso de MPI, encontramos un error relacionado con la gestión del espacio, ya que los datos eran demasiado grandes para la función `gather`. Esto generalmente ocurre porque MPI intenta empaquetar los datos, y el tamaño combinado de los datos de todos los procesos resulta demasiado grande para ser manejado en una sola operación. Para resolver este problema, adoptamos una estrategia diferente para reunir los datos. En lugar de utilizar `comm.gather`, escribimos los resultados parciales de cada proceso en un archivo temporal y luego combinamos estos archivos en el proceso maestro. Utilizamos archivos de memoria mapeada (mmap) para evitar problemas de memoria.

```
OverflowError: integer 2500000141 does not fit in 'int'
OverflowError: integer 2500000141 does not fit in 'int'
```

Otro desafío fue la estrategia de paralelización. Inicialmente, intentamos enviar partes tanto de la secuencia 1 como de la secuencia 2, pero esto resultó en una matriz ineficiente. Finalmente, optamos por dividir solo la secuencia 1 y enviar toda la secuencia 2. Sin embargo, la matriz del dotplot se envió con un tamaño correspondiente a la porción de la secuencia 1 por el tamaño completo de la secuencia 2.



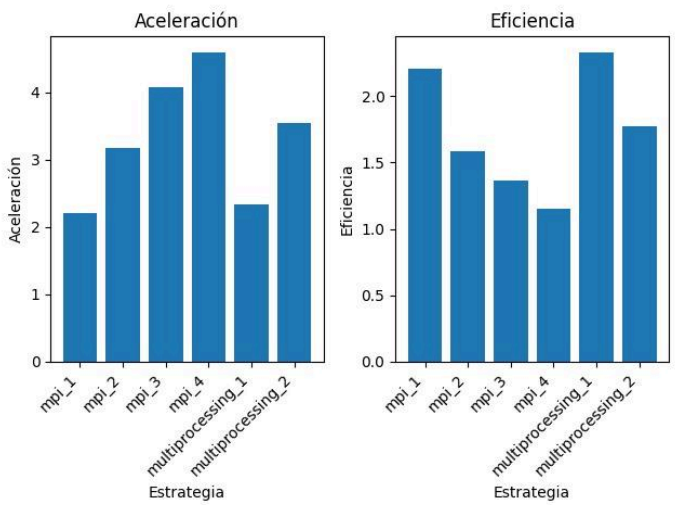



Además, nos encontramos con errores en la implementación de la memoria compartida, ya que no se gestionó correctamente el acceso concurrente a los datos, lo que llevó a condiciones de carrera y resultados inconsistentes. Para resolver esto, implementamos mecanismos de sincronización adecuados, como semáforos, para asegurar que solo un proceso pudiera acceder a una sección crítica del código a la vez.

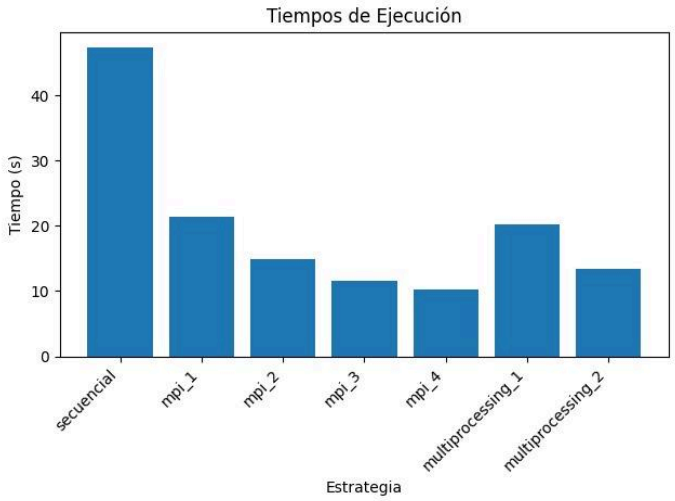
Otro problema significativo fue el manejo de excepciones y errores en el tiempo de ejecución. En varias ocasiones, nuestro programa se detuvo inesperadamente debido a errores no controlados. Para mejorar la robustez del sistema, incorporamos manejo de excepciones más riguroso y pruebas exhaustivas para identificar y corregir posibles fallos antes de la ejecución final.

A lo largo del proyecto, estos errores y desafíos nos permitieron aprender y adaptar nuestras estrategias para manejar grandes volúmenes de datos y optimizar el rendimiento del sistema. Este proceso de resolución de problemas fue fundamental para el éxito del proyecto y proporcionó valiosas lecciones para futuras investigaciones.

Durante nuestras pruebas, observamos una variación significativa en los tiempos de ejecución debido a las diferencias en las máquinas utilizadas. Como se puede ver en la Figura 1 y la Figura 2, la estrategia de multiprocessing no fue tan eficiente en comparación con mpi en algunas máquinas. Sin embargo, en otras máquinas, los tiempos de ejecución de multiprocessing fueron comparables a los de mpi.




En esta figura, se muestra la aceleración y la eficiencia de las diferentes estrategias de paralelización. Se observa que la eficiencia de multiprocessing varía considerablemente, lo cual puede deberse a las diferencias en el hardware y la configuración del sistema.




En esta otra figura, se presentan los tiempos de ejecución de las estrategias. Aunque multiprocessing mostró tiempos más largos en algunas pruebas, en otras máquinas los tiempos fueron similares a los obtenidos con mpi.

Estos resultados destacan la importancia de considerar el entorno de ejecución al evaluar la eficiencia de diferentes estrategias de paralelización. Las diferencias en hardware y configuración pueden tener un impacto significativo en el rendimiento.

## Explicación de proyecto

Nuestro proyecto consiste en un programa que permite generar matrices de dotplot utilizando diferentes estrategias de paralelización, con el objetivo de comparar sus rendimientos. La herramienta principal del proyecto es el archivo `proyecto.py`, que contiene toda la lógica necesaria para ejecutar el programa desde la línea de comandos. A continuación, se describen en detalle las principales funcionalidades y componentes del proyecto:

### Funcionalidades del Proyecto

1. **Ejecución desde la Línea de Comandos:**
   - El archivo `proyecto.py` está diseñado para ser ejecutado desde la línea de comandos, lo que permite al usuario especificar parámetros como los archivos de entrada, la estrategia de paralelización a utilizar, el tamaño del filtro y el número de núcleos a utilizar en el caso de multiprocessing.
2. **Carga de Archivos:**
   - El programa admite archivos en formato FASTA para las secuencias que se utilizarán en la generación del dotplot. Estas secuencias son combinadas y preprocesadas según los parámetros especificados por el usuario.
3. **Generación del Dotplot:**
   - Según la estrategia de paralelización seleccionada (secuencial, hilos, multiprocessing, MPI o PyCuda), el programa genera una matriz de dotplot que visualiza las similitudes entre las dos secuencias de entrada.
4. **Visualización y Almacenamiento:**
   - Cada ejecución del programa produce una imagen del dotplot generada con la estrategia de paralelización utilizada. Estas imágenes se guardan automáticamente con nombres descriptivos que incluyen la estrategia utilizada.
   - Además, se aplica un filtro a la imagen del dotplot para resaltar posibles diagonales, lo que facilita la identificación de regiones de alta similitud entre las secuencias.
5. **Registro de Tiempos:**
   - El programa guarda los tiempos de carga de datos, ejecución y visualización en un archivo JSON. Este archivo almacena información detallada de cada ejecución, incluyendo el número de núcleos utilizados y los tiempos de cada etapa del proceso.
6. **Generación de Gráficas Comparativas:**
   - Utilizando los datos almacenados en el archivo JSON, el programa genera gráficas comparativas que muestran los tiempos de ejecución, aceleración y eficiencia de cada

estrategia de paralelización. Estas gráficas proporcionan una visión clara del rendimiento relativo de cada método.

## Componentes Principales

1. **Archivo `proyecto.py`:**
   - Este archivo es el núcleo del proyecto y contiene la lógica para la ejecución del programa. Incluye funciones para el manejo de argumentos de la línea de comandos, carga de archivos, generación del dotplot, aplicación de filtros y visualización de resultados.
2. **Funciones de Paralelización:**
   - El programa incluye múltiples funciones para manejar diferentes estrategias de paralelización. Estas funciones están optimizadas para aprovechar las capacidades de procesamiento de múltiples núcleos de CPU y GPU.
3. **Archivo JSON para Registro de Tiempos:**
   - El archivo JSON actúa como una base de datos que almacena los resultados de cada ejecución. Esto permite realizar análisis detallados y comparaciones entre diferentes ejecuciones y estrategias.
4. **Gráficas de Rendimiento:**
   - El programa genera automáticamente gráficas que visualizan el rendimiento de las diferentes estrategias. Estas gráficas son esenciales para entender cómo varía el rendimiento según el entorno de hardware y la estrategia utilizada.

Este proyecto no solo permite generar y visualizar dotplots de manera eficiente, sino que también proporciona una plataforma para analizar y comparar diferentes estrategias de paralelización. Al registrar y graficar los tiempos de ejecución, aceleración y eficiencia, podemos obtener una comprensión profunda del rendimiento de cada método en distintos entornos de hardware. Este enfoque integral es crucial para optimizar el procesamiento de grandes volúmenes de datos biológicos y puede ser aplicado a una amplia gama de problemas computacionales que requieren análisis paralelo.

Nuestro proyecto combina la flexibilidad de la ejecución por línea de comandos con potentes capacidades de procesamiento paralelo y análisis de rendimiento, ofreciendo una herramienta robusta y extensible para la generación y análisis de dotplots.

# Conclusión

En este proyecto, hemos comparado diversas estrategias de paralelización para el cálculo de matrices de dotplot, utilizando tanto CPU como GPU. Nuestras estrategias incluyen métodos secuenciales, multiprocessing, MPI y PyCuda, cada uno evaluado en términos de tiempo de ejecución, aceleración y eficiencia. A continuación, resumimos nuestras observaciones y hallazgos clave:

## Resumen de Resultados

1. **Estrategia Secuencial:**
   - La estrategia secuencial, como era de esperarse, mostró tiempos de ejecución más largos. Esta estrategia sirvió como línea base para medir la eficiencia y la aceleración de las demás estrategias.
2. **Multiprocessing y MPI:**
   - Las estrategias basadas en multiprocessing y MPI mostraron mejoras significativas en comparación con la ejecución secuencial. Sin embargo, los resultados variaron considerablemente según la configuración del hardware de las máquinas utilizadas.
   - En algunas máquinas, `multiprocessing` no fue tan eficiente, mientras que en otras mostró tiempos de ejecución comparables a `mpi`. Esto subraya la importancia del entorno de hardware en la evaluación del rendimiento de estas estrategias.
3. **PyCuda:**
   - PyCuda, ejecutado en Google Colab debido a la falta de tarjetas gráficas Nvidia en nuestras máquinas locales, demostró ser la estrategia más eficiente. La capacidad de la GPU para manejar miles de hilos en paralelo resultó en tiempos de ejecución significativamente más bajos.
   - Las gráficas de aceleración y eficiencia mostraron que PyCuda no solo redujo el tiempo de ejecución de manera impresionante, sino que también utilizó los recursos computacionales de manera óptima, con una eficiencia cercana a 40. Esto indica que casi todos los recursos de la GPU fueron empleados eficazmente para acelerar la ejecución del programa.

## Discusión de los Hallazgos

Los resultados de este estudio destacan varias conclusiones importantes:

1. **Importancia del Entorno de Ejecución:**
   - Los tiempos de ejecución y la eficiencia de las estrategias de paralelización dependen en gran medida del entorno de hardware. Las diferencias en la configuración de las máquinas pueden tener un impacto significativo en el rendimiento. Esto es particularmente evidente en las variaciones

observadas en las estrategias de `multiprocessing` y `mpi`.

2. **Ventajas de las GPUs:**
    - Las GPUs ofrecen una capacidad de procesamiento paralela superior en comparación con las CPUs, especialmente para tareas que pueden ser divididas en muchas operaciones concurrentes. La utilización de Google Colab para acceder a recursos de GPU permitió aprovechar estas capacidades, resultando en mejoras significativas en el rendimiento.
3. **Complejidad de Implementación:**
    - Si bien las GPUs ofrecen ventajas en términos de rendimiento, la implementación de algoritmos para GPUs puede ser más compleja. Requiere un conocimiento profundo de la arquitectura de la GPU y la programación paralela, lo que puede representar un desafío adicional en el desarrollo de soluciones eficientes.
4. **Eficiencia de PyCuda:**
    - PyCuda demostró ser una herramienta extremadamente efectiva para la computación paralela en GPUs, proporcionando aceleración y eficiencia superiores. Este hallazgo sugiere que, siempre que estén disponibles los recursos de GPU, PyCuda debería ser considerado como una opción preferida para tareas intensivas en cálculo.

## Implicaciones

Las conclusiones de este proyecto tienen varias implicaciones para futuros trabajos:

1. **Optimización del Uso de Recursos:**
    - Para el desarrollo se debe considerar la optimización del uso de recursos y seleccionar la estrategia de paralelización adecuada según el entorno de hardware disponible. Evaluar el rendimiento en diferentes configuraciones puede ayudar a tomar decisiones informadas.
2. **Desarrollo de Algoritmos en GPUs:**
    - Fomentar el desarrollo y la capacitación en programación de GPUs puede abrir nuevas oportunidades para mejorar el rendimiento de aplicaciones computacionalmente intensivas. La implementación de PyCuda, aunque compleja, puede ofrecer beneficios significativos.
3. **Evaluación Comparativa:**
    - Continuar con estudios comparativos entre diferentes estrategias de paralelización en diversos entornos de hardware permitirá una mejor comprensión de sus ventajas y limitaciones, contribuyendo al desarrollo de soluciones más eficientes.

En conclusión, este proyecto ha demostrado la importancia de seleccionar la estrategia de paralelización adecuada según el entorno de hardware y ha resaltado el potencial de las GPUs para mejorar significativamente el rendimiento computacional. Los futuros proyectos deberían centrarse en optimizar el uso de estos recursos y en desarrollar herramientas y técnicas que faciliten la implementación de soluciones paralelas eficientes.