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

Dany Orlando Guancha Tarapues
Jhair Alexander Peña Aguirre
Jefferson Henao Cano

I. Abstract

Este proyecto implementa y analiza el rendimiento de cuatro enfoques diferentes para realizar un dotplot: una versión secuencial, una versión paralela utilizando la biblioteca multiprocessing de Python, una versión paralela utilizando mpi4py, y una versión paralela utilizando PyCUDA. Se comparan estas implementaciones en términos de eficiencia y rendimiento, proporcionando información valiosa sobre las ventajas y desventajas de los enfoques secuenciales y paralelos en tareas de bioinformática.

II. Introducción

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

En la implementación secuencial, se desarrollará un algoritmo que realiza el dotplot de forma lineal, procesando uno a uno los elementos de las secuencias. Aunque sencillo, este enfoque puede ser lento cuando se trabaja con secuencias grandes.

En la versión paralela con multiprocessing, se aprovechará la capacidad de ejecutar múltiples procesos simultáneamente en sistemas con varios núcleos de CPU. Dividiendo el trabajo en tareas más pequeñas y distribuyéndolas entre los procesos, se acelerará el cálculo del dotplot, logrando un procesamiento más rápido y un uso eficiente de los recursos.

En la versión paralela con mpi4py, se utilizará la biblioteca mpi4py, que se basa en el estándar Message Passing Interface (MPI) para la programación paralela. Se distribuirán las tareas de cálculo del dotplot entre los procesos MPI, permitiendo un rendimiento aún mayor en comparación con la versión paralela con multiprocessing.

En la versión paralela con PyCUDA, se aprovechará la capacidad de las GPUs para realizar cálculos masivos en paralelo. Usando la biblioteca PyCUDA, se implementará el dotplot aprovechando el paralelismo masivo que ofrecen las tarjetas gráficas, lo que puede resultar en una aceleración significativa del proceso.

III. Materiales y funciones

III-A. Librerías

Se utilizó el lenguaje de programación Python en su versión 3.10 para implementar el algoritmo. Se emplearon varias bibliotecas clave, como Numpy, Matplotlib, Time, mpi4py, opencv, Multiprocessing y PyCUDA, las cuales son fundamentales para manipular matrices, generar gráficos, leer archivos, detectar diagonales mediante filtros, medir tiempos, obtener parámetros de línea de comandos y trabajar con múltiples procesadores y GPU. En la Tabla I se detallan las versiones de cada uno de los software y bibliotecas utilizados en la investigación.

Cuadro I
Paquetes Utilizados

Paquete
Python
Matplotlib
Numpy
mpi4py
Time
Multiprocessing
PyCUDA
OpenCV
Tqdm
BioPython

III-B. Paralelización

El proceso de paralelización se realizó por medio de las librerías multiprocessing, mpi4py y PyCUDA de Python. Para este proceso, como entradas se cargaron dos secuencias que se querían alinear gráficamente, organizándose en una matriz NxM (donde N y M son las longitudes de las secuencias respectivamente).

- **Multiprocessing:** Se utilizó la librería Pool para generar los procesos de acuerdo con la cantidad de threads que se recibían por parámetro, y por cada pool se manejaba la función map para recorrer cada índice de la primera secuencia por cada índice de la segunda secuencia y realizar la comparación de estos para guardar en una lista un número de representación de color, que sería la representación gráfica que queremos obtener.
- **mpi4py:** Se utilizó la estrategia de Chunks, para dividir la primera secuencia en matrices más pequeñas y se creó otra matriz donde se guardará la solución de la implementación. En cada recorrido de los Chunks contra las posiciones de la segunda secuencia, se efectúa la misma comparación de la implementación con multiprocessing, para darle un valor de color a la posición de la solución, luego se unen las soluciones generadas.

- **PyCUDA:** Se aprovechó la capacidad de las GPUs para realizar cálculos masivos en paralelo. Utilizando PyCUDA, se implementó el dotplot distribuyendo las operaciones de comparación entre los núcleos de la GPU, acelerando significativamente el procesamiento en comparación con CPU.

III-C. Datos de experimentación

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

III-D. Disponibilidad del algoritmo

La herramienta presentada en este trabajo es de acceso libre. El código fuente y la guía de uso e instalación están disponibles en: https://github.com/AguirreCode/concurrent_project.git

IV. Resultados

```
Tiempo de ejecución secuencial: 98.63680791854858
```

Figura 1. Tiempo secuencial

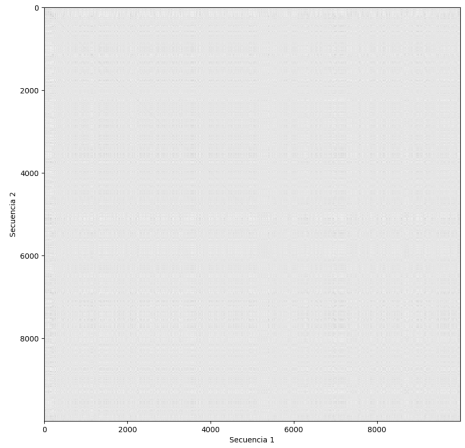

Figura 2. Dotplot para la solución de la matriz evaluada

Con un dotplot solución, en el que se evidencia la diagonal principal que servirá para el análisis de las secuencias.

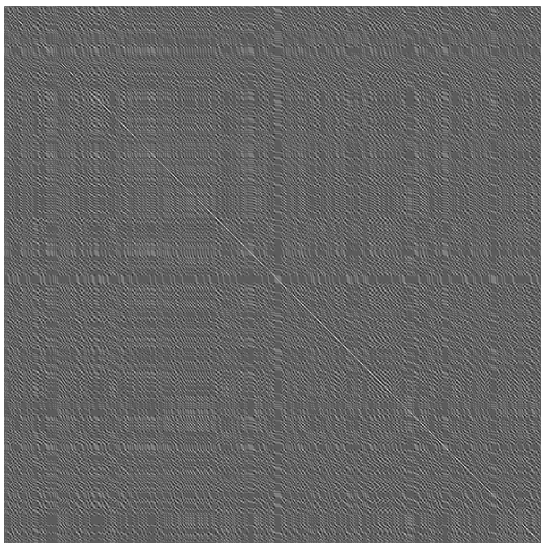

Figura 3. Dotplot filtrado para la solución de la matriz evaluada

Terminado el proceso secuencial, se vuelve a correr el archivo, pero con la implementación paralela de multiprocessing, la cual presenta los siguientes datos:

```
Tiempo de ejecución multiprocessing con 1 hilos: 103.43224573135376
Tiempo de ejecución multiprocessing con 2 hilos: 57.64410185813904
Tiempo de ejecución multiprocessing con 4 hilos: 31.244438648223877
Tiempo de ejecución multiprocessing con 8 hilos: 21.64829993247986
Aceleración con 1 hilos: 1.0
Aceleración con 2 hilos: 1.7943249449099472
Aceleración con 4 hilos: 3.3104214927212716
Aceleración con 8 hilos: 4.7778471944412395
Eficiencia con 1 hilos: 1.0
Eficiencia con 2 hilos: 0.8971624724549736
Eficiencia con 4 hilos: 0.8276053731803179
Eficiencia con 8 hilos: 0.5972308993051549
```

Figura 4. Tiempos para la Escalabilidad, Aceleración y Eficiencia multiprocessing

En la imagen anterior se observa la escalabilidad de la implementación. Se utilizaron 8 hilos; el último tuvo un tiempo de ejecución de 1,2 minutos, lo cual se puede evidenciar en la aceleración y la eficacia obtenidas en la ejecución.

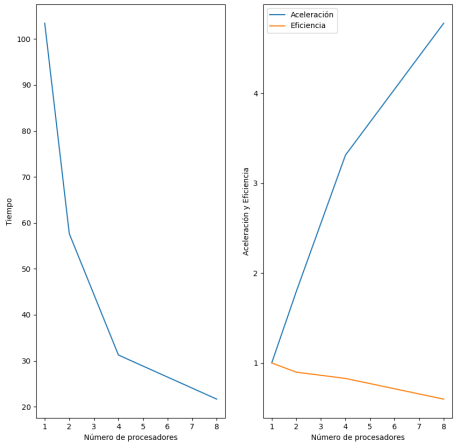

Figura 5. Gráficas de aceleración y aceleración vs eficiencia Multiprocessing

Al observar las gráficas de comparación entre tiempo de ejecución, aceleración y eficacia, podemos observar que para un hilo la ejecución tomaba más tiempo en reproducir la solución. A medida que se incrementaban los hilos, el tiempo de ejecución mejoraba. Sin embargo, a mayor cantidad de hilos, por el overhead, los tiempos van en aumento, la aceleración disminuye y la eficacia se pierde.

```
Tiempo de ejecución mpi con 1 procesos: 365.0365560054779
Aceleración con 1 procesos: 1.0
Eficiencia con 1 procesos: 1.0
Tiempo de ejecución mpi con 2 procesos: 190.51074814796448
Aceleración con 2 procesos: 1.91609428630223
Eficiencia con 2 procesos: 0.958047143151115
Tiempo de ejecución mpi con 3 procesos: 135.4464213848114
Aceleración con 3 procesos: 2.69506238904893
Eficiencia con 3 procesos: 0.8983541296829767
Tiempo de ejecución mpi con 4 procesos: 104.55816102027893
Aceleración con 4 procesos: 3.4912296892318095
Eficiencia con 4 procesos: 0.8728074223079524
```

Figura 6. Tiempos para la escalabilidad de MPI

En la imagen anterior vemos los valores que nos arroja la ejecución del algoritmo con diferentes números de núcleos y el tiempo.

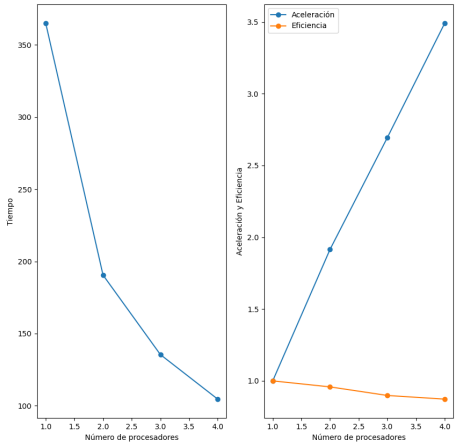

Figura 7. Gráficas de aceleración y aceleración vs eficiencia MPI

En la gráfica vemos que hasta cierto punto, a medida que aumentamos el número de núcleos, la ejecución del algoritmo se hace más rápida. Pero entre los 3 y 4 núcleos, la aceleración disminuye y su eficiencia también.

```
Tiempo de ejecución PyCUDA: 2.5200414657592773
Aceleración con 10000 longitud de secuencia: 1.0
Eficiencia con 10000 longitud de secuencia: 0.5
Tiempo de ejecución PyCUDA: 2.357171058654785
Aceleración con 15000 longitud de secuencia: 1.0690957096670959
Eficiencia con 15000 longitud de secuencia: 0.5345478548335479
Tiempo de ejecución PyCUDA: 2.3207497596740723
Aceleración con 20000 longitud de secuencia: 1.0858738454046823
Eficiencia con 20000 longitud de secuencia: 0.36195794846822743
Tiempo de ejecución PyCUDA: 2.401687145233154 3
Aceleración con 25000 longitud de secuencia: 1.0492796577443617
Eficiencia con 25000 longitud de secuencia: 0.26231991443609043
Tiempo de ejecución PyCUDA: 2.285701274871826
Aceleración con 30000 longitud de secuencia: 1.1025244171072146
Eficiencia con 30000 longitud de secuencia: 0.2205048834214292 92
```

Figura 8. Tiempos para la escalabilidad de pycuda

En la imagen anterior vemos los valores que nos arroja la ejecución del algoritmo con diferentes números de núcleos y el tiempo.

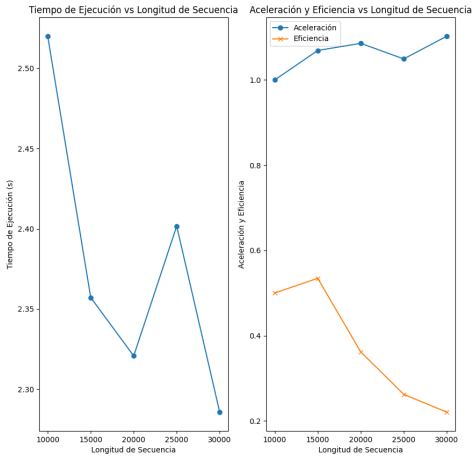

Figura 9. Gráficas de tiempo de ejecución vs longitud de secuencia, aceleración y eficiencia vs longitud de secuencia

En la primera gráfica, ”Tiempo de Ejecución vs Longitud de Secuencia”, observamos que el tiempo de ejecución varía en función de la longitud de la secuencia. Para una secuencia de longitud 10,000, el tiempo de ejecución inicia arriba de 2.5 segundos. Al aumentar la longitud de la secuencia a 15,000, el tiempo disminuye a aproximadamente 2.35 segundos. Cuando la secuencia tiene una longitud de 20,000, el tiempo de ejecución desciende un poco más, se acerca a 2.3 segundos. Cuando la longitud es de 25000 el tiempo aumenta a 2.4 segundos y luego para una longitud de 30000 el tiempo desciende a menos de 2.3 segundos. De esto se concluye que, a medida que la longitud de la secuencia aumenta, el tiempo de ejecución varía de manera no lineal.

En la segunda gráfica, .ªceleración y Eficiencia vs Longitud de Secuencia”, se muestra cómo la aceleración y la eficiencia cambian con la longitud de la secuencia. La aceleración, representada en azul, disminuye constantemente a medida que aumenta la longitud de la secuencia. Por otro lado, la eficiencia, mostrada en naranja, también tiende a disminuir conforme se incrementa la longitud de la secuencia. Aunque hay fluctuaciones en la aceleración, la tendencia general es una disminución a medida que la longitud de la secuencia crece. Esto indica que, con secuencias más largas, tanto la aceleración como la eficiencia tienden a reducirse.

REFERENCIAS

[1] J. S. Piña, S. Orozco-Arias, N. Tobón-Orozco, M. S. Candamil-Cortés, R. Tabares-Soto y R. Guyot, .ªlineamiento gráfico de secuencias a través de programación paralela: un enfoque desde la era postgenómica”, Revista Ingeniería Biomédica, vol. 13, n.º 26, pp. 37–45, 2019.

[2] G. Zaccone, Python Parallel Programming Cookbook. Packt Publishing, Limited, 2015.

[3] ”Función de convolución—ArcMap —Documentación”. [Online]. Available: https://desktop.arcgis.com/es/arcmap/latest/manage-data/raster-and-images/convolution-function.htm