Análisis de secuencias de ADN: Programación secuencial vs Programación Paralela (Multiprocessing - MPI4py)

Juan D. Osorio, Anderson J. López, y Edwar A. Marín
Estudiantes de Ingeniería de sistemas y computación

Resumen - En este informe, se presenta un análisis de rendimiento de tres enfoques diferentes para el cálculo del dotplot: secuencial, utilizando multiprocessing y utilizando mpi4py. El dotplot es una técnica comúnmente utilizada en bioinformática para comparar secuencias de ADN o proteínas. El objetivo de este estudio es evaluar el rendimiento y la eficiencia de cada enfoque en términos de tiempos de ejecución, aceleración, eficiencia y escalabilidad. Los resultados mostraron que las implementaciones paralelas lograron reducir significativamente los tiempos de ejecución en comparación con la implementación secuencial. Las implementaciones en paralelo mostraron mejoras en el rendimiento al aumentar el número de procesos o hilos utilizados, pero se observó un punto de saturación. Estos hallazgos resaltan la importancia de la paralelización en el cálculo del dotplot y brindan información valiosa para la selección del enfoque más adecuado según el entorno de ejecución y los recursos disponibles.

Palabras clave – Multiprocessing, Programación paralela, MPI4py, Programación secuencial, aceleración, eficiencia, rendimiento, dotplot

I. INTRODUCCIÓN

La bioinformática es un área multidisciplinaria que apoya el descubrimiento de información biológica utilizando enfoques computacionales. Este campo de investigación se encuentra en la intersección de varias ciencias como la biología, la informática y las matemáticas, con el objetivo de analizar y clasificar datos biológicos.

Una de las tareas más comunes en bioinformática, relevante en metagenómica y análisis filogenético, es la alineación de secuencias, la cual consiste en comparar 2 o más secuencias de nucleótidos o proteínas contra una referencia.

Uno de los tipos de alineación más utilizados es la alineación gráfica de secuencias, que proporciona una visualización de reorganizaciones, inserciones, deleciones y otras estructuras encontradas en secuencias de ADN o proteínas. Las alineaciones gráficas se representan comúnmente en una matriz de puntos llamada diagrama de puntos.

Existen varios paquetes de software publicados capaces de generar diagramas de puntos. Dotter es uno de los alineadores gráficos de secuencia más populares, que utiliza programación dinámica y es adecuado para pequeñas secuencias de ADN o proteínas (algunos miles de nucleótidos).

A pesar del progreso significativo en el uso de grandes conjuntos de datos de secuencias, la calidad y los tiempos de ejecución de los diagramas de puntos siguen representando un desafío.

Los avances en computación de alto rendimiento, supercomputación y programación paralela han mejorado el tiempo de ejecución en varias áreas, debido a que la programación paralela puede lanzar procesos en arquitecturas heterogéneas como CPU, GPU o CPU + GPU utilizando bibliotecas para programar de manera rápida y flexible, incluso con memoria compartida o no compartida.

En bioinformática, estas técnicas permiten acelerar el análisis de la información genética. Diferentes aplicaciones de bioinformática utilizan enfoques de programación paralela. Sin embargo, los alineadores gráficos que utilizan estrategias paralelas no están disponibles.

En este informe, presentaremos el análisis de rendimiento de tres formas de calcular el dotplot: secuencialmente, utilizando multiprocessing y utilizando mpi4py. Se describirá el marco teórico, el procedimiento utilizado, se presentará el código implementado, se analizarán los resultados obtenidos, se discutirán las implicaciones y se brindarán conclusiones sobre el rendimiento y la eficiencia de cada enfoque.

II. MARCO TEÓRICO

Documento presentado el día 07/06/2023 para el proyecto de la asignatura ‘Programación concurrente y distribuida’ en el semestre 2023-01 impartida por el docente e investigador Reinel Tabares Soto. Universidad de Caldas.
Contacto de los autores:
Juan David Osorio (juan.1701714291@ucaldas.edu.co)
Anderson Julián López (andersson.1701522324@ucaldas.edu.co)

A. *DOTPLOT*

En un dotplot, las secuencias se colocan a lo largo de los ejes horizontal y vertical de la matriz. Cada celda de la matriz representa una comparación entre un par de caracteres de las dos secuencias. Si los caracteres en la posición correspondiente de las secuencias son iguales o similares, se coloca un punto en la celda. La intensidad del punto (o su color) puede variar según el grado de similitud entre los caracteres.

A través de la visualización en dotplot, los científicos pueden detectar fácilmente regiones conservadas, repeticiones, inversiones, inserciones y deleciones en las secuencias. Las regiones conservadas aparecerán como líneas diagonales ininterrumpidas, mientras que las repeticiones se presentarán como líneas diagonales paralelas. Las inversiones se mostrarán como líneas diagonales en dirección opuesta a la diagonal principal, y las inserciones y deleciones aparecerán como interrupciones en las líneas diagonales.

El dotplot es especialmente útil en el análisis de secuencias largas y complejas, donde las técnicas de alineamiento tradicionales pueden resultar ineficientes o difíciles de interpretar. Además, el dotplot permite una comparación visual rápida y es una herramienta valiosa en áreas como la genómica, la metagenómica, la filogenia y la biología molecular para estudiar la evolución, las relaciones funcionales y estructurales entre secuencias y la identificación de secuencias homólogas.

B. *PARALELIZACIÓN*

La paralelización es una técnica fundamental en la computación de alto rendimiento (HPC, por sus siglas en inglés) que busca optimizar la ejecución de tareas computacionalmente intensivas mediante la división del trabajo en partes más pequeñas y su procesamiento simultáneo utilizando múltiples procesadores, núcleos de computación o unidades de procesamiento gráfico (GPU). Esta estrategia permite aprovechar al máximo los recursos computacionales disponibles y acelerar significativamente el tiempo de ejecución de las tareas.

Paralelismo en memoria compartida: Se utiliza en sistemas donde todos los procesadores o núcleos tienen acceso a una memoria común. Los hilos de ejecución pueden compartir información a través de esta memoria. Ejemplos de modelos de programación para este enfoque incluyen OpenMP y Pthreads.

Paralelismo en memoria distribuida: Se aplica en sistemas donde cada procesador o núcleo tiene su propia memoria local. La comunicación entre procesadores se realiza mediante el intercambio de mensajes o el uso de mecanismos de sincronización. Ejemplos de modelos de programación para este enfoque incluyen MPI (Message Passing Interface) y PGAS (Partitioned Global Address Space).

III. ESTRUCTURA DE LA APLICACIÓN

Este programa de línea de comandos en Python permite calcular el Dotplot de dos secuencias en formato FASTA. Se utilizan las siguientes bibliotecas:

- BioPython: para leer secuencias desde archivos FASTA.
- NumPy: para crear y manipular matrices que representan el Dotplot.
- Matplotlib: para generar la representación gráfica del Dotplot.
- multiprocessing: para realizar cálculos en paralelo utilizando múltiples procesadores.
- mpi4py: para realizar cálculos en paralelo utilizando el Message Passing Interface (MPI).
- convolve2d: es una función de la biblioteca SciPy que realiza la convolución bidimensional de dos matrices dadas.
- Argparse: facilita la creación de programas de línea de commandos

A. *LECTURA DE SECUENCIAS*

La función *merge_sequences_from_fasta* lee las secuencias desde el archivo FASTA utilizando la biblioteca BioPython y las concatena para que sean sólo una variable str.

B. *CÁLCULO SECUENCIAL*

La función *calculate_dotplot_sequential* realiza el cálculo del Dotplot de manera secuencial utilizando un algoritmo que compara todas las posiciones de ambas secuencias. Se crea una matriz vacía llamada dotplot, luego se recorren las secuencias y se llena la matriz con 1s donde las posiciones correspondientes coinciden y 0s donde no coinciden.

C. *CÁLCULO PARALELO (MULTIPROCESSING)*

La función *calculate_dotplot_parallel* utiliza la biblioteca multiprocessing para realizar el cálculo del Dotplot en paralelo. La tarea se divide en segmentos y se asigna cada segmento a un proceso diferente, utilizando la clase Pool de multiprocessing. Se crea una lista de segmentos, donde cada uno es un subconjunto de la secuencia 1, y se invoca la función dotplot_chunk en paralelo utilizando Pool.starmap. Luego, los resultados se combinan en una matriz única llamada dotplot.

D. CÁLCULO PARALELO (MPI4PY)

La función *calculate_dotplot_mpi* utiliza la biblioteca mpi4py para realizar el cálculo del Dotplot en paralelo utilizando MPI. Se divide la secuencia 1 entre los procesos utilizando las funciones comm.scatter o comm.scatterv y se recopilan los resultados utilizando comm.gather. El proceso principal (rank 0) combina los resultados en una matriz única y aplica el filtro filter_dotplot.

E. FILTRO DEL DOTPLOT

La función *filter_dotplot* aplica un filtro de ventana deslizante a la matriz del Dotplot para suavizar las áreas de alta coincidencia. Utiliza la función convolve2d de la biblioteca SciPy para aplicar una ventana de convolución en la matriz del Dotplot y luego aplica un umbral para generar una matriz binaria final.

Fig1. Imagen sin filtrar

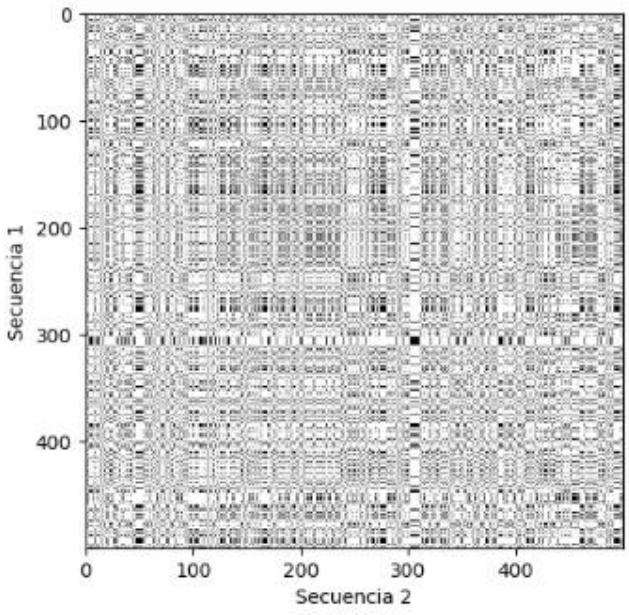
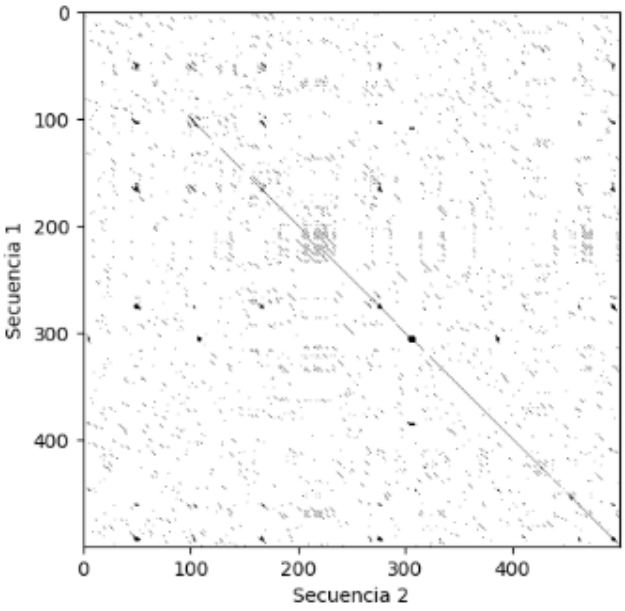


Fig2. Imagen filtrada



F. GENERACIÓN Y GUARDADO DE IMAGEN

La función *save_dotplot* guarda la representación gráfica del Dotplot en un archivo de imagen utilizando la biblioteca matplotlib. La función imshow se utiliza para visualizar la matriz del Dotplot y el resultado se guarda en un archivo con el nombre proporcionado.

G. LINEA DE COMANDOS

El programa admite argumentos de línea de comandos para especificar los archivos de entrada y salida, los núcleos, así como las opciones para realizar el cálculo secuencial, en paralelo utilizando multiprocessing, o en paralelo utilizando mpi4py y el nombre de la salida, la cual es una imagen. La función *parse_arguments* se encarga de analizar estos argumentos para compararlos con la entrada desde la terminal.

H. EJECUCIÓN

La función *run_dotplot_analysis* se encarga de leer las secuencias desde los archivos FASTA, calcular el Dotplot utilizando la opción de cálculo especificada y guardar la representación gráfica del Dotplot en un archivo de imagen.

IV. ANÁLISIS DE MÉTRICAS

A. TIEMPOS DE EJECUCIÓN TOTALES

El tiempo total de ejecución lleva la contabilización desde el instante en el que se ejecutó el programa hasta que finalizó, está incluido en él la carga de datos, la creación y asignación de las estructuras de datos, la comparación de las secuencias (ya sea paralela o secuencal), el cargado de imágenes y demás.

Para los tres tipos de ejecución se escogieron cómo pruebas de entrada 350 líneas de la secuencia de Escherichia coli (aproximadamente 23920 proteínas) contra 350 líneas de Salmonella.

La ventana de convolución para el filtrado está configurada para que sea de 5x5 con un threshold de 0,8.

A.1. SECUENCIAL

Se hizo una única ejecución, en la cuál se demoró en procesar 118.51 segundos.

```
El código en secuencial se ejecutó en: 118.5133147239685 segundos
```

A.2. MULTIPROCESSING

Para la primera ejecución, la cual se ejecutó utilizando un solo núcleo, tuvo un tiempo de 359.45 segundos

```
La matriz de resultado tiene tamaño:  (23920, 23920)
Dotplot con  1  procesadores, tiempo:  359.4583613872528  segundos
```

Para la segunda ejecución, la cual se ejecutó utilizando dos núcleos, tuvo un tiempo de 222.29 segundos

```
La matriz de resultado tiene tamaño:  (23920, 23920)
Dotplot con  2  procesadores, tiempo:  222.29658389091492  segundos
```

Para la tercera ejecución, la cual se ejecutó utilizando tres núcleos, tuvo un tiempo de 196.11 segundos

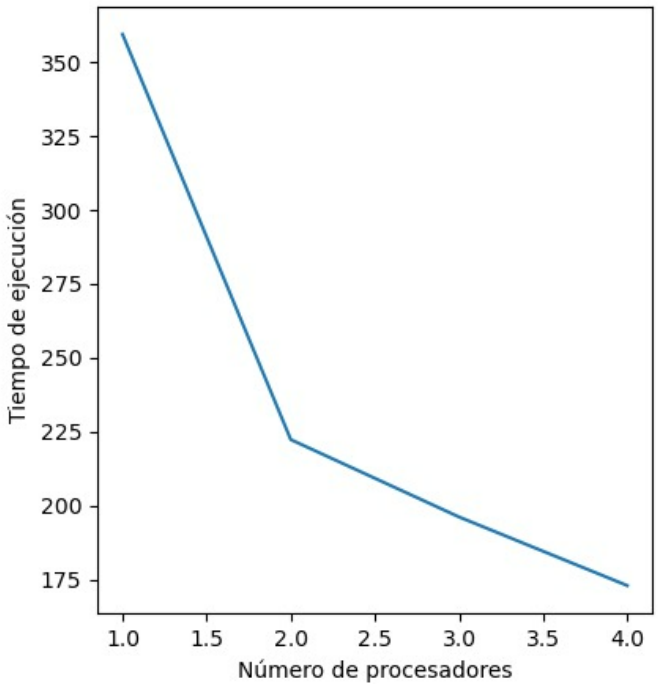
```
La matriz de resultado tiene tamaño:  (23920, 23920)
Dotplot con  3  procesadores, tiempo:  196.1170892715454  segundos
```

Para la cuarta ejecución, la cual se ejecutó utilizando cuatro núcleos, tuvo un tiempo de 172.89 segundos

```
La matriz de resultado tiene tamaño:  (23920, 23920)
Dotplot con  4  procesadores, tiempo:  172.89522004127502  segundos
```

Podemos resumir estas ejecuciones en la siguiente gráfica, la cuál nos indica que el tiempo de ejecución fue descendiendo a medida que el número de procesadores aumentaba de una manera constante desde el segundo procesador, lo que nos indica que la velocidad es proporcional.

Fig3. Tiempo de ejecución vs. Procesadores (multiprocessing)



A.3. MPI4PY

Para la primera ejecución, la cual se ejecutó utilizando un solo núcleo, tuvo un tiempo de 352.81 segundos

```
yecto/AnalisisSecuenciasADN_Paralelizacion$ mpiexec -n 1 python comandos5.py --i
nput1 Salmonella6.fna --input2 E_coli6.fna --output dotplot15.png --mpi
```

```
Tiempo de ejecución total en MPI: 352.8144178390503 segundos
```

Para la segunda ejecución, la cual se ejecutó utilizando dos núcleos, tuvo un tiempo de 225.89 segundos

```
yecto/AnalisisSecuenciasADN_Paralelizacion$ mpiexec -n 2 python comandos5.py --i
nput1 Salmonella6.fna --input2 E_coli6.fna --output dotplot15.png --mpi
Tiempo de ejecución total en MPI: 225.89974403381348 segundos
```

Para la tercera ejecución, la cual se ejecutó utilizando tres núcleos, tuvo un tiempo de 187.84 segundos

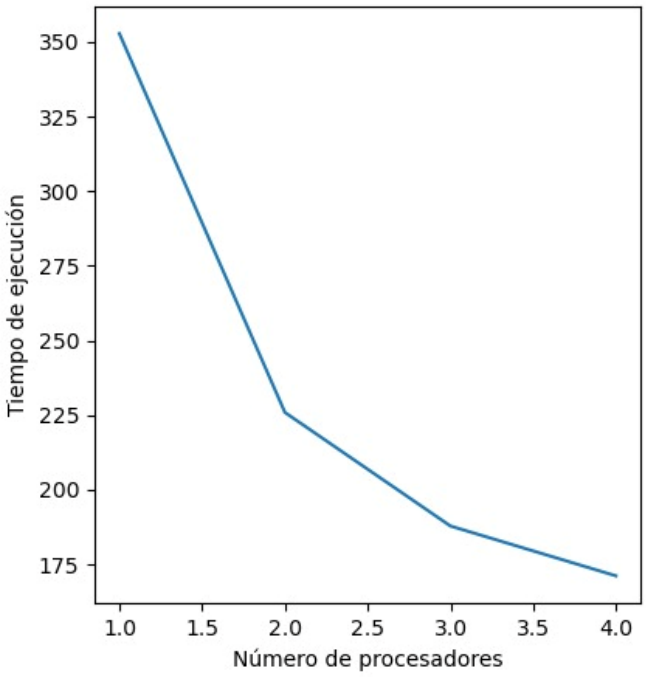
```
yecto/AnalisisSecuenciasADN_Paralelizacion$ mpiexec -n 3 python comandos5.py --i
nput1 Salmonella6.fna --input2 E_coli6.fna --output dotplot15.png --mpi
Tiempo de ejecución total en MPI: 187.84728908538818 segundos
```

Para la cuarta ejecución, la cual se ejecutó utilizando cuatro núcleos, tuvo un tiempo de 171.14 segundos

```
yecto/AnalisisSecuenciasADN_Paralelizacion$ mpiexec -n 4 python comandos5.py --i
nput1 Salmonella6.fna --input2 E_coli6.fna --output dotplot15.png --mpi
Tiempo de ejecución total en MPI: 171.14893579483032 segundos
```

Podemos resumir estas ejecuciones en la siguiente gráfica, la cuál nos indica que el tiempo de ejecución fue descendiendo a medida que el número de procesadores aumentaba de una manera no tan constante), pues cada que se agregaba un procesador, la curva cambiaba su rumbo (no es tan recta), lo que nos indica que la velocidad cambia más que con multiprocessing al aumentar procesadores.

Fig4. Tiempo de ejecución vs. Procesadores (mpi4py)



B. TIEMPOS DE EJECUCIÓN PARCIALES (PORCIÓN PARALELA)

B.1. MULTIPROCESSING

Se calculó sólo la porción en dónde el Pool ejecutaba la función de comparar la secuencia con el chuck correspondiente a cada 1 de los procesos que se realizaba en paralelo. Para 4 nucleos se demoró 119.45 segundos

```
El código en multiprocessing(porción paralela) se ejecutó en: 119.45642280578613 segundos
```

B.2. MPI4PY

Se calculó sólo la porción en dónde se ejecutaba en paralelo la comparación de la secuencia con el chuck correspondiente a cada 1 de los procesos que se realizaba en paralelo y hasta que se hacia el gather para unir los resultados. Para 4 nucleos se demoró 119.10 segundos

```
Tiempo de ejecución en MPI(porción paralela): 119.10199522972107 segundos
```

C. TIEMPO DE CARGA DE DATOS

C.1. SECUENCIAL

La carga de los archivos fasta sólo tardó 0.034 segundos

```
El tiempo de carga de datos es 0.0034062862396240234 segundos
```

C.2. MULTIPROCESSING

La carga de los archivos fasta sólo tardó 0.064 segundos

```
El tiempo de carga de datos es 0.06445765495300293 segundos
```

C.3. MPI4PY

La carga de los archivos fasta sólo tardó 0.0039 segundos

```
El tiempo de carga de datos es 0.00396418571472168 segundos
```

D. TIEMPO DE GENERACIÓN DE IMAGEN

D.1. SECUENCIAL

El crear y guardar la imagen sólo tardó 1.22 segundos

```
El tiempo de carga de imagenes es 1.2260479927062988 segundos
```

D.2. MULTIPROCESSING

El crear y guardar la imagen sólo tardó 0.95 segundos

```
El tiempo de carga de imagenes es 0.9580440521240234 segundos
```

D.3. MPI4PY

El crear y guardar la imagen sólo tardó 0.91 segundos

```
El tiempo de carga de imagenes es 0.914488540008545 segundos
```

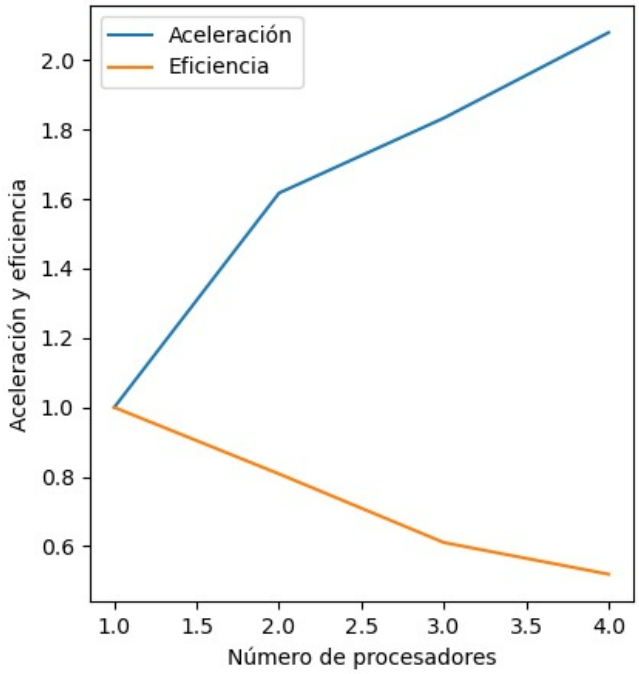
E. ACELERACIÓN Y EFICIENCIA

E.1. MULTIPROCESSING

En la aceleración, como pudimos notar en la Fig3. El tiempo de ejecución tuvo un cambio bastante alto cuando se cambió de 1 núcleo a 2, lo que nos da a entender que es allí en dónde más hubo aceleración., Al ir aumentando los núcleos este tiempo de ejecución siguió disminuyendo, aunque ya no en una relación tan alta como la primera.

Y en cuanto a la eficiencia, se puede notar que a medida que aumentábamos los nucleos esta se iba reduciendo. Esto es gracias al overhead, que impide que se pueda lograr una aceleración lineal perfecta y por tanto una eficiencia ideal de 1.

Fig5. aceleracion/eficiencia vs procesadores (multiprocessing)

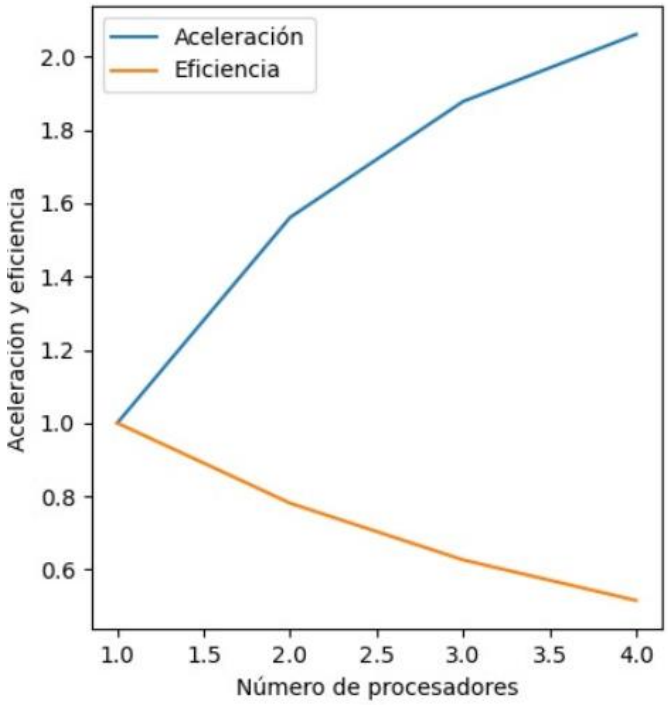


E.2. MPI4PY

En la aceleración, como pudimos notar en la Fig4. El tiempo de ejecución tuvo un cambio bastante alto cuando se cambió de 1 núcleo a 2, lo que nos da a entender que es allí en dónde más hubo aceleración., Al ir aumentando los núcleos 1 en 1, vemos que la línea de aceleración es más inclinada que la de multiprocessing, lo que nos indica que mpi4py tuvo una mejor aceleración.

Y en cuanto a la eficiencia, se puede notar que a medida que aumentábamos los nucleos esta se iba reduciendo. Esto es gracias al overhead, que impide que se pueda lograr una aceleración lineal perfecta y por tanto una eficiencia ideal de 1.

Fig6. aceleracion/eficiencia vs procesadores (mpi4py)



F. ESCALABILIDAD

F.2. MULTIPROCESSING

Fig7. Escalamiento fuerte (multiprocessing)

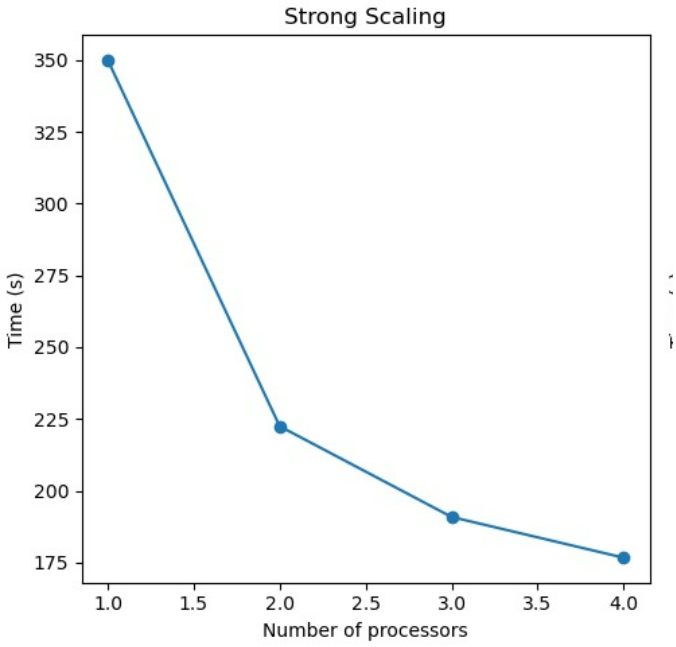


Fig8. Escalamiento débil (multiprocessing)