DESARROLLO Y OPTIMIZACIÓN DE UN ALGORITMO DE HAPLOTIPADO GENÓMICO.

Peréz, L.O - Navarro, P.

Introducción

Contexto y Relevancia del problema

El haptotipado genómico, también conocido como "puesta en fase" (phasing), es un proceso bioinformático fundamental para la interpretación de datos genéticos. Su objetivo es asignar los alelos de múltiples variantes genéticas a sus cromosomas de origen, ya sea materno o paterno, para reconstruir secuencias cromosómicas coherentes (Tewhey et al., 2011). Esta información es crucial en la genómica médica, ya que permite, por ejemplo, el diagnóstico de enfermedades complejas y la identificación de heterocigotos compuestos, donde dos alelos recesivos diferentes en un mismo gen causan una enfermedad sólo si se encuentran en cromosomas distintos (Roach et al., 2010; Tewhey et al., 2011).

Para formalizar los conceptos clave, definimos los siguientes términos:

- **Haplotipo:** Es la combinación específica de alelos o polimorfismos que se encuentran en una misma región de un cromosoma. Debido a su proximidad física, estos alelos tienden a heredarse juntos (McVean & Cardin, 2005). Como consecuencia, a nivel poblacional se observan asociaciones o agrupamientos no aleatorios de variantes, que son la base para la inferencia por consenso (Li & Stephens, 2003).

- **Puesta en fase:** Es el procedimiento computacional mediante el cual se determina qué alelos pertenecen a cada uno de los dos cromosomas homólogos de un individuo.

El desafío computacional central del haplotipado radica en que para un conjunto de n posiciones heterocigotas, existen 2^n combinaciones de haplotipos teóricamente posibles. Este crecimiento exponencial del espacio de búsqueda convierte la determinación de la fase por fuerza bruta en un problema de alto coste computacional, especialmente al analizar grandes cohortes poblacionales o regiones genómicas extensas (Browning & Browning, 2007; Scheet & Stephens, 2006).

Objetivos

Este informe documenta el desarrollo y la optimización de un algoritmo de haplotipado. Los objetivos específicos del trabajo son los siguientes:

- Implementar en Python un algoritmo de haplotipado basado en la inferencia por consenso poblacional, que utiliza la frecuencia de los haplotipos en una muestra para inferir la fase en cada individuo.
- Identificar y analizar los cuellos de botella computacionales inherentes al diseño secuencial del algoritmo.
- Aplicar técnicas de paralelismo, específicamente la biblioteca ***multiprocessing*** de Python, para optimizar el rendimiento y reducir significativamente los tiempos de ejecución.
- Realizar un análisis de rendimiento exhaustivo para evaluar la escalabilidad del algoritmo optimizado, variando sistemáticamente la carga de trabajo (número de individuos y posiciones genómicas) y el número de procesos computacionales.

A continuación, se describe en detalle la metodología empleada para alcanzar estos objetivos.

Metodología

Esta sección detalla el enfoque estratégico adoptado para el desarrollo y la evaluación del rendimiento del algoritmo. Se describe el flujo de trabajo del algoritmo base, identificando sus puntos de mayor coste computacional. Posteriormente, se explica la estrategia de paralelización implementada para mitigar estos cuellos de botella. Finalmente, se presenta

el entorno experimental, el diseño de las pruebas y las métricas utilizadas para cuantificar la eficacia de la optimización. El conjunto de datos y el código fuente necesarios para la reproducción de las pruebas se encuentran alojados en el siguiente repositorio: https://github.com/pablo1n7/computacion_paralela_2025.

Algoritmo Base y Cuellos de Botella

El algoritmo secuencial fue implementado en Python y su flujo de trabajo se divide en tres etapas principales:

1. **Carga y Preparación de Datos:** En esta fase inicial, se cargan los datos genotípicos utilizando las bibliotecas `pandas` y `numpy`. Los datos provienen del Proyecto 1000 Genomas [1], específicamente de una región del cromosoma 6 (posiciones 32604240-32612402, built g37), previamente filtrada y procesada con la herramienta Plink para garantizar la calidad de la muestra. Además se procesó el archivo *.ped* original de Plink a un formato más amigable (separando los dos alelos de todos los individuos en archivos distintos) para abrir en Python con la librería Pandas.

2. **Construcción del Consenso Poblacional (de haplotipos):** Esta etapa constituye el primer cuello de botella computacional. El proceso consiste en iterar sobre cada individuo de la muestra y, para cada uno, analizar todas las 2^n `combinaciones` de haplotipos posibles. Se contabiliza la frecuencia absoluta de cada haplotipo observado en la población total, creando un diccionario de consenso que servirá como referencia para la fase de asignación.

El siguiente código muestra la implementación del algoritmo secuencial para construir el consenso poblacional:

[1] https://www.cog-genomics.org/plink/2.0/resources

```
IND, POS = 5,3 # para 5 individuos con 3 posiciones (2^3 = 8 comb)
print(combinaciones)#[(0, 0, 0),  (0, 0, 1), (0, 1, 0),  (0, 1, 1),
                    #(1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
alelos = np.concatenate([
alelo_uno.reshape(IND, POS, 1),
alelo_dos.reshape(IND, POS, 1)], axis=2) #Unimos los dos alelos
alelos.shape # (IND, POS, 2) → (5, 3, 2)
consenso = {}
for x in range(IND): # por cada individuo
   for i in range(len(combinaciones)):# por cada combinación 2^pos
      val = ''.join(alelos[x, np.arange(POS), combinaciones[i]])
      if val not in consenso: # por cada individuo
         consenso[val] = 0
      consenso[val] += 1
```

3. **Asignación de Haplotipos (Phasing):** Este proceso representa el segundo cuello de botella computacional. Para cada individuo, el algoritmo vuelve a iterar sobre todas las 2^n combinaciones posibles. Para cada combinación (haplotipo 1), calcula su haplotipo complementario (haplotipo 2) y consulta sus respectivas frecuencias en el consenso poblacional construido en la etapa anterior. Finalmente, se selecciona el par de haplotipos (1, 2) que maximiza la suma de las frecuencias poblacionales de ambos haplotipos del par, asumiendo que el par más probable es aquel compuesto por los haplotipos más frecuentes en la población.

El siguiente código muestra el algoritmo secuencial para resolver la asignación de haplotipos:

```
for x in range(IND):# por cada individuo
   combinaciones_con_prob = []
   cantidad_total # len(combinaciones)*IND
   for i in range(len(combinaciones)):# por cada combinación 2^pos
       alelo1_ind = combinaciones[i] # comb i
       alelo2_ind = np.abs(np.array(alelo1_ind) - 1)# opuesta comb(i)
       alelo1 = ''.join(alelos[x, np.arange(POS), alelo1_ind])
       alelo2 = ''.join(alelos[x, np.arange(POS), alelo2_ind])
       prob_1 = consenso[alelo1]/cantidad_total
       prob_2 = consenso[alelo2]/cantidad_total
       combinaciones_con_prob.append([alelo1, prob_1, alelo2, prob_2])

   print("Seleccionado:", funcion_seleccion(combinaciones_con_prob))
```

La ***"funcion_seleccion"*** es el algoritmo encargado de seleccionar los haplotipos mediante el criterio de obtener el par de haplotipos maximizando la suma de probabilidades en ambos alelos:

```
def funcion_seleccion(combinaciones_con_prob):
   """ algoritmo de selección """
   combinaciones_con_prob = np.array(combinaciones_con_prob)
   indice = np.argmax(np.array(combinaciones_con_prob[:, 1], dtype=float) +
np.array(combinaciones_con_prob[:, 3], dtype=float))
   return combinaciones_con_prob[indice]
```

Estrategia de Optimización y Paralelismo

La estrategia de optimización se centró en la paralelización del código mediante la biblioteca *multiprocessing* de Python, que permite explotar el paralelismo de datos a nivel de individuo. Esta estrategia se aplicó directamente a los dos cuellos de botella identificados:

• Paralelización del Consenso y la Asignación: La carga de trabajo en ambas fases es inherentemente paralela por individuo. Se distribuyó el bucle principal que itera sobre los individuos, de modo que cada proceso de un Pool de *multiprocessing* se encarga de analizar un subconjunto de la población.

Para la implementación se utilizaron herramientas específicas que facilitaron la paralelización:

• ***collections.Counter***: En la fase de construcción de consenso, esta estructura de datos permite agregar los resultados parciales de cada proceso de manera muy eficiente, sumando las frecuencias de haplotipos de forma simple y robusta.

• ***functools.partial***: Se empleó para fijar argumentos estáticos mediante una clausura de función (functools.partial) para reducir la sobrecarga de la comunicación entre procesos. Esto permitió pasar eficientemente a las funciones trabajadoras (***_worker_task*** y ***_worker_calcular_hoplotipo***) los datos que son comunes a todos los procesos (como los datos de alelos y las combinaciones precalculadas).

El código para calcular el consenso utilizando paralelización de *multiprocessing* es el siguiente:

```
import multiprocessing
from functools import partial
from collections import Counter

def _worker_task(x, alelos, num_posiciones, combinaciones):
   """
   Función que procesa un solo individuo (x)
   y devuelve un Counter con sus frecuencias.
   """

   res_local = Counter()
     for i in range(len(combinaciones)):
       val = ''.join(alelos[x, np.arange(num_posiciones), combinaciones[i]])
       res_local[val] += 1

   return res_local

def calcular_frecuencias(alelos, num_individuos, num_posiciones, combinaciones, procesos):
   """
   Calcular frecuencias de cada combinación en paralelo
   usando el número especificado de 'procesos'.
   """
   tarea_parcial = partial(_worker_task,
                           alelos=alelos,
                           num_posiciones=num_posiciones,
                           combinaciones=combinaciones)

   indices_individuos = range(num_individuos)

   with multiprocessing.Pool(processes=procesos) as pool:

       lista_de_counters = pool.map(tarea_parcial, indices_individuos)

   res_final = Counter()
   for res_local in lista_de_counters:
       res_final.update(res_local) # .update() suma las cuentas

   return dict(res_final)
```

Se modularizó el cálculo de frecuencia para un solo individuo dentro de la función _worker_task. Además se creó la función calcular_frecuencias para encargarse de crear el pool de tareas para cada individuo. Su funcionamiento se basa en la función pool.map, la cual divide la lista *indices_individuos* en bloques (chunks). Por defecto, los tamaños de los

bloques se calculan basándose en la longitud de la lista y el número de CPUs. Dentro de la estructura *with,* se envía un bloque entero de índices (por ejemplo, los índices del 0 al 100) a un trabajador (o *Worker*). El mismo recibe el bloque y ejecuta _worker_task secuencialmente para los índices que recibió (desde 0, luego el 1, hasta el 100). Al momento que todos los Workers finalizan su trabajo *lista_de_counters* contiene todos los resultados parciales.

Para la función de selección de haplotipos se procedió de manera similar:

```
def _worker_calcular_hoplotipo(x, posiciones, alelos, combinaciones, res, cantidad_total):

    combinaciones_con_prob = []
    for i in range(len(combinaciones)):
        aleloA_ind = combinaciones[i]
        aleloB_ind = np.abs(np.array(aleloA_ind) - 1)
        aleloA = ''.join(alelos[x, np.arange(posiciones), aleloA_ind])
        aleloB = ''.join(alelos[x, np.arange(posiciones), aleloB_ind])

        prob_A = res.get(aleloA, 0) / cantidad_total
        prob_B = res.get(aleloB, 0) / cantidad_total
        combinaciones_con_prob.append([aleloA, prob_A, aleloB, prob_B])

    return funcion_seleccion(combinaciones_con_prob)

def calcular_hoplotipo(individuos, posiciones, alelos, combinaciones, res, cantidad_total,
procesos):
    """
    Calcular haplotipo más frecuente para cada individuo (en paralelo).
    """
    tarea_parcial = partial(_worker_calcular_hoplotipo,
                            posiciones=posiciones,
                            alelos=alelos,
                            combinaciones=combinaciones,
                            res=res,
                            cantidad_total=cantidad_total)


    indices_individuos = range(individuos)

    with multiprocessing.Pool(processes=procesos) as pool:
        results = pool.map(tarea_parcial, indices_individuos)

    return results
```

Entorno Experimental, Diseño de Pruebas y Métricas

Todas las pruebas se ejecutaron en un entorno controlado con las siguientes especificaciones:

Hardware	● CPU: Intel Core i9-14900K @ 6.00GHz (32 cores) ● RAM: 125 GB ● Discos HDD: ○ sda: 7.3 TB (ST8000VE000-3BZ1)
Software	● SO: Ubuntu 24.04.1 LTS ● CPython: 3.12.3
Datos	Proyecto 1000 Genomas (chr6: 32604240-32612402, built g37) Subset de 3000 individuos con 585 posiciones.

El diseño experimental se estructuró para evaluar la escalabilidad del algoritmo bajo diferentes condiciones. Los parámetros variables fueron:

- Número de Procesos: [1, 2, 3, 4, 6, 8, 16 32]
- Número de Individuos (Tamaño del Problema): [500, 1000, 1500, 3000]
- Número de Posiciones (Complejidad del Problema): [4, 6, 8, 10, 12]

Para evaluar cuantitativamente el rendimiento de la implementación paralela, se utilizaron dos métricas estándar en computación de alto rendimiento:

- **Speedup:** Mide la ganancia de velocidad obtenida al utilizar N procesos en comparación con la ejecución secuencial (un solo proceso). Se calcula con la fórmula: $Speedup = T\_base / T\_N$ Donde T_base es el tiempo de ejecución con 1 proceso y T_N es el tiempo de ejecución con N procesos.
- **Eficiencia:** Es una medida de cuán eficazmente se utilizan los recursos de procesamiento en paralelo. Una eficiencia del 100% indica una escalabilidad lineal perfecta. Se calcula como: $Eficiencia = Speedup / N$

Los resultados obtenidos a partir de estas métricas se presentan y analizan en la siguiente sección.

Análisis de resultados

En esta sección se presentan los resultados empíricos del benchmark de rendimiento. El análisis se enfoca en cuantificar la escalabilidad del algoritmo en dos dimensiones ortogonales: el tamaño del problema (carga de datos, $N\_individuos$) y la complejidad computacional (espacio de búsqueda, 2^{n_pos}).

Análisis de Rendimiento por Variación de Carga (Número de Individuos)

Para evaluar el impacto del tamaño del problema, se mantuvo fijo el número de posiciones (pos=12) y se varió el número de individuos. Los resultados se presentan en la Tabla 1.

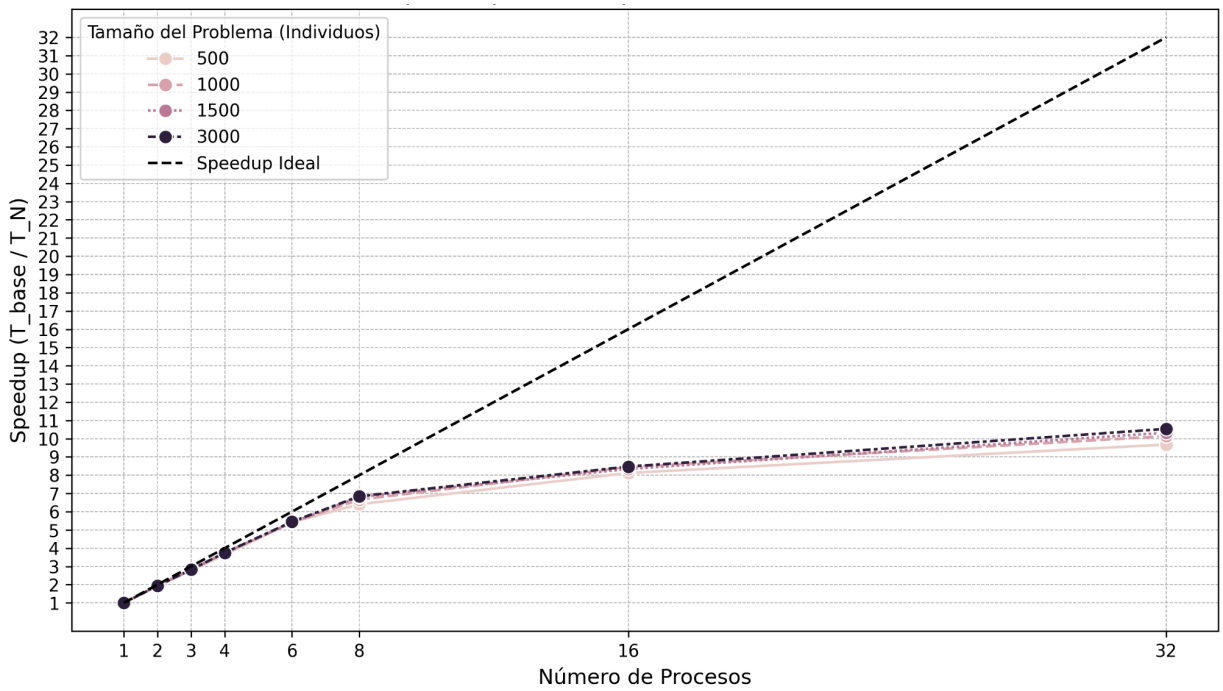
Tabla 1: Tiempo total de cómputo (promedio de 10 repeticiones) variando la carga. Se comparan los resultados obtenidos con 12 marcadores genéticos.

Tamaño del Problema	Cant. Procesos	Tiempo prom (Seg)	Speedup	Eficiencia
500	1	13,25	1,00	100,00
	4	3,65	3,63	90,81
	6	2,45	5,42	90,31
	8	2,07	6,41	80,09
	16	1,63	8,12	50,74
	32	1,37	9,68	30,25
1000	1	26,64	1,00	100,00
	4	7,15	3,73	93,15
	6	4,93	5,40	90,02
	8	4,00	6,66	83,23
	16	3,15	8,45	52,78
	32	2,63	10,14	31,68
1500	1	41,06	1,00	100,00
	4	10,98	3,74	93,52
	6	7,60	5,40	90,03
	8	6,03	6,81	85,10

	16	4,92	8,34	52,13
	32	3,97	10,33	32,28
3000	1	81,66	1,00	100,00
	4	21,79	3,75	93,70
	6	14,99	5,45	90,77
	8	11,93	6,84	85,55
	16	9,64	8,47	52,95
	32	7,75	10,54	32,93

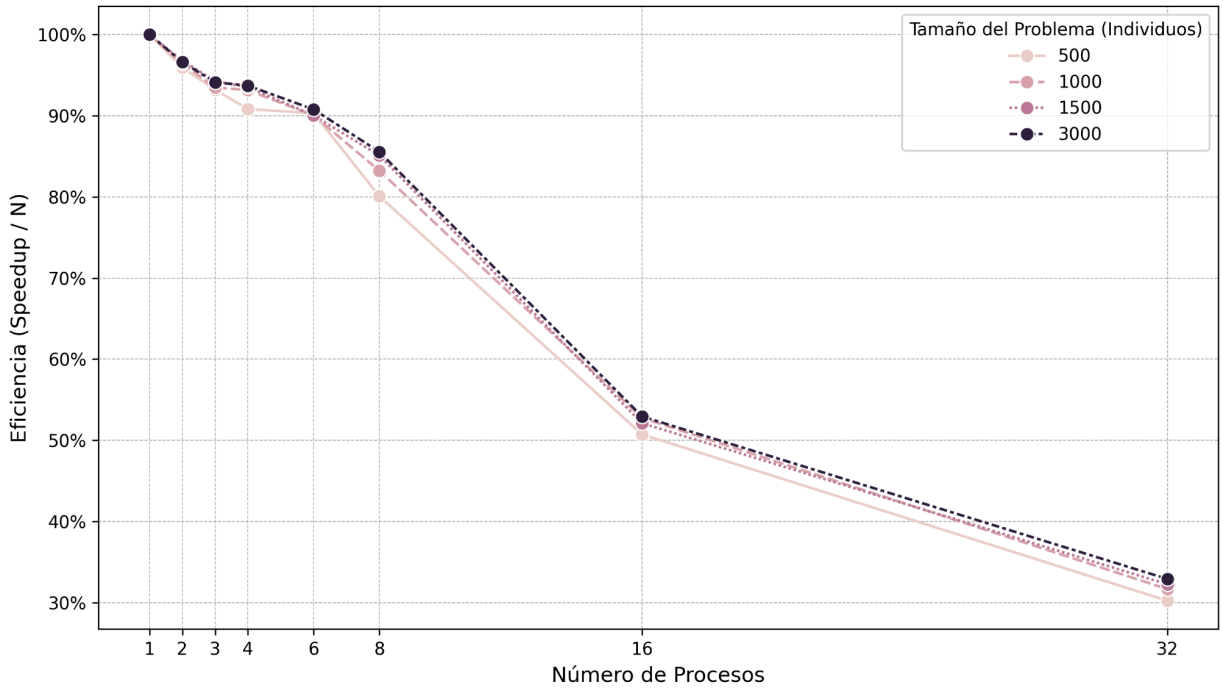
Se logró una escalabilidad robusta aunque sub-lineal, con un speedup que aumenta consistentemente con el número de procesos para todas las cargas de trabajo. El hallazgo clave es que la ganancia de velocidad se aproxima más al speedup ideal a medida que aumenta el número de individuos. Por ejemplo, con 12 posiciones y 8 procesos, la carga de 3000 individuos alcanza un speedup de 6.84, lo que representa el 85.5% del speedup ideal (Ver Figura 1).

Figura 1: Gráfico de *speedup* del algoritmo completo procesando datos de 12 posiciones genómicas. Las curvas de colores representan diferentes tamaños del problema (N° de individuos). La línea punteada negra indica el *speedup* teórico ideal ($s_p = p$), donde el rendimiento aumenta linealmente con el número de procesos.



Como es esperado, la eficiencia disminuye al añadir procesos debido al *overhead* inherente a la orquestación del *Pool* de procesos (creación, comunicación y sincronización) . Sin embargo, para un número fijo de procesos, la eficiencia es sistemáticamente mayor en problemas con más individuos (Ver Figura 2). Esto se debe a que un mayor número de individuos aumenta la granularidad computacional del trabajo asignado a cada proceso. Con tareas de mayor granularidad, el coste fijo del *overhead* representa una fracción menor del tiempo total de ejecución, mejorando la eficiencia global.

Figura 2: Gráfico de eficiencia para distintos tamaños de población, utilizando 12 posiciones genómicas.



Exceptuando las configuraciones de alto paralelismo (16 y 32 procesos), los datos de la Tabla 1 evidencian que el incremento en la carga de trabajo amortigua la pérdida de eficiencia asociada al incremento del paralelismo. Al escalar de 4 a 8 procesos con un tamaño de problema pequeño (500 individuos), la eficiencia sufre una caída notable de 10,7 puntos porcentuales (descendiendo del 90,81% al 80,09%). En contraste, al elevar la carga a 3000 individuos, esta disminución se atenúa a 8,15

puntos (pasando del 93,70% al 85,55%), lo que confirma que el algoritmo aprovecha mejor los recursos de cómputo en escenarios de mayor carga.

Análisis de Rendimiento por Variación de Complejidad (Número de Posiciones)

Por otro lado, para evaluar el impacto de la complejidad computacional, se mantuvo fijo el número de individuos (3000) y se varió el número de posiciones genómicas.

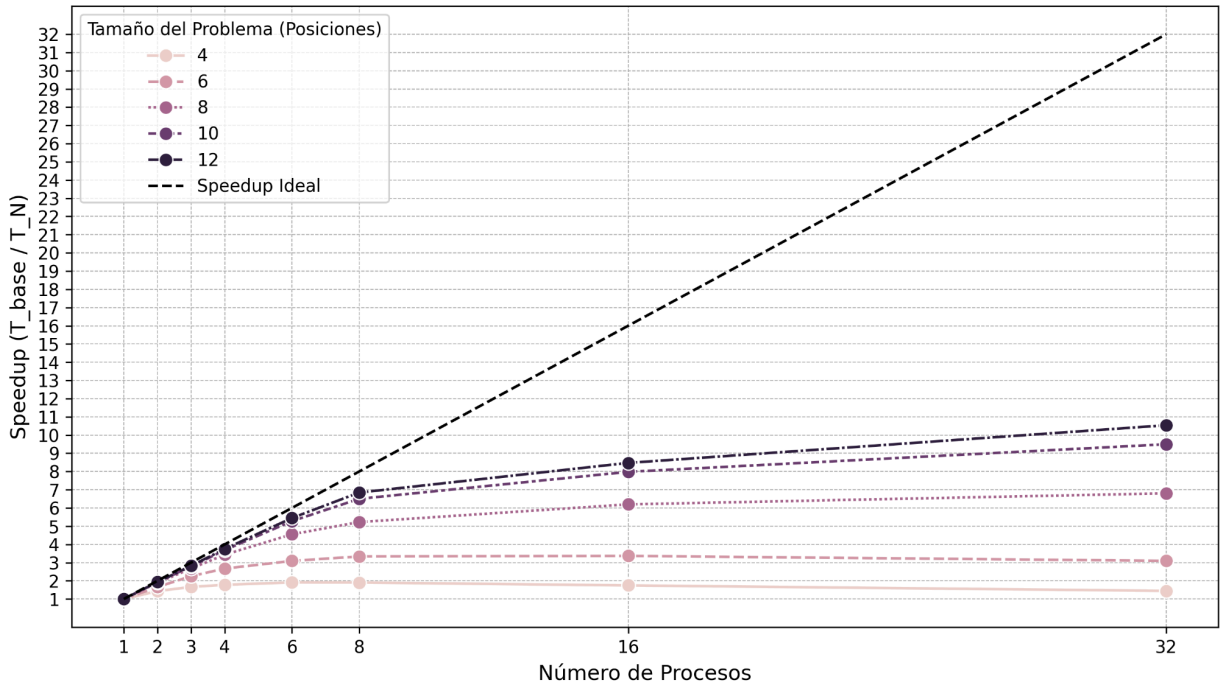
Tabla 2: Tiempo total de cómputo (promedio de 10 repeticiones) variando la complejidad. Se comparan los resultados obtenidos con 3000 individuos.

Número de Posiciones	Cant. Procesos	Tiempo prom (Seg)	Speedup	Eficiencia
4	1	0,45	1,00	100,00
4	2	0,32	1,44	71,76
4	3	0,27	1,66	55,48
4	4	0,26	1,78	44,53
4	6	0,24	1,91	31,87
4	8	0,24	1,91	23,82
4	16	0,26	1,75	10,92
4	32	0,31	1,45	4,52
6	1	1,33	1,00	100,00
6	2	0,80	1,66	82,82
6	3	0,59	2,26	75,19
6	4	0,50	2,67	66,75
6	6	0,43	3,09	51,55
6	8	0,40	3,34	41,70
6	16	0,40	3,37	21,04
6	32	0,43	3,10	9,68
8	1	4,99	1,00	100,00
8	2	2,63	1,90	94,79
8	3	1,87	2,67	89,05
8	4	1,45	3,43	85,86
8	6	1,10	4,55	75,87
8	8	0,96	5,22	65,29

8	16	0,81	6,19	38,71
8	32	0,73	6,80	21,25
10	1	19,81	1,00	100,00
10	2	10,27	1,93	96,38
10	3	7,11	2,79	92,90
10	4	5,38	3,68	92,05
10	6	3,75	5,28	87,92
10	8	3,05	6,50	81,25
10	16	2,48	7,98	49,88
10	32	2,09	9,50	29,67
12	1	81,66	1,00	100,00
12	2	42,26	1,93	96,61
12	3	28,92	2,82	94,11
12	4	21,79	3,75	93,70
12	6	14,99	5,45	90,77
12	8	11,93	6,84	85,55
12	16	9,64	8,47	52,95
12	32	7,75	10,54	32,93

Los datos demuestran que la paralelización es drásticamente más efectiva cuando el tiempo de cómputo por tarea, que crece exponencialmente con el número de posiciones (n), domina sobre el overhead de la gestión de procesos. Con 3000 individuos y 8 procesos, el speedup para 4 posiciones es de aproximadamente 1.9, una ganancia modesta. Sin embargo, para 12 posiciones, el speedup se dispara a 6.9. Esto indica que a mayor granularidad computacional (tareas más complejas por individuo), mejor es la escalabilidad (Ver Figura 3).

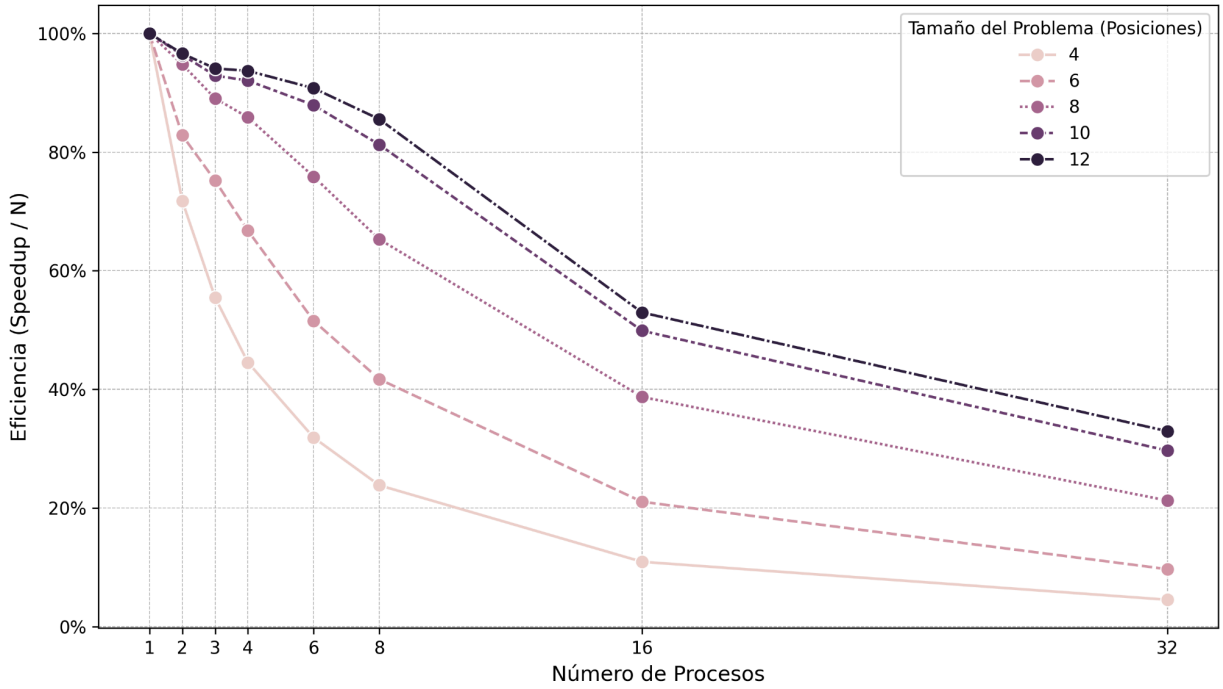
Figura 3: Gráfico de *speedup* del algoritmo completo procesando datos de 3000 individuos. La línea punteada negra indica el *speedup* teórico ideal ($s_p = p$), donde el rendimiento aumenta linealmente con el número de procesos.



La eficiencia muestra una tendencia complementaria. A medida que aumenta la complejidad computacional, el overhead de la paralelización se vuelve proporcionalmente menos significativo, resultando en una mayor eficiencia global.

Para 3000 individuos, la eficiencia con 8 procesos para un problema de 4 posiciones es de apenas un 24%, mientras que para un problema de 12 posiciones, la eficiencia se mantiene en un robusto 85.55%. La curva de eficiencia para 12 posiciones muestra una caída mucho más suave y contenida en comparación con la drástica caída observada para 4 posiciones (Ver Figura 4).

Figura 4: Gráfico de eficiencia para distintos tamaños de población, utilizando 3000 individuos.



Análisis de los cuellos de Botella

Finalmente, se analizó de forma desagregada el comportamiento de las dos funciones críticas del algoritmo: *calcular_frecuencias* (construcción del consenso) y *calcular_hoplotipo* (asignación de haplotipos).

Aunque ambas funciones exhiben patrones de escalabilidad análogos (Ver Figura 5 y 6), un análisis detallado revela diferencias sutiles. Se observa que la función de asignación de haplotipos (*calcular_hoplotipo*) mantiene una eficiencia marginalmente superior en comparación con la de construcción de consenso (*calcular_frecuencias*).

Esto podría deberse a que, si bien ambas iteran sobre 2^n combinaciones, la segunda función realiza búsquedas en el diccionario de consenso y cálculos de probabilidad, lo que podría implicar una carga computacional ligeramente más densa por individuo en comparación con la simple agregación de contadores en *calcular_frecuencias*. En cualquier caso, ambas escalan de forma robusta, validando que la estrategia de paralelización fue efectiva para los dos cuellos de botella identificados.

Figura 5: Gráfico de speedup para las funciones desagregadas, utilizando 3000 individuos y 12 posiciones.

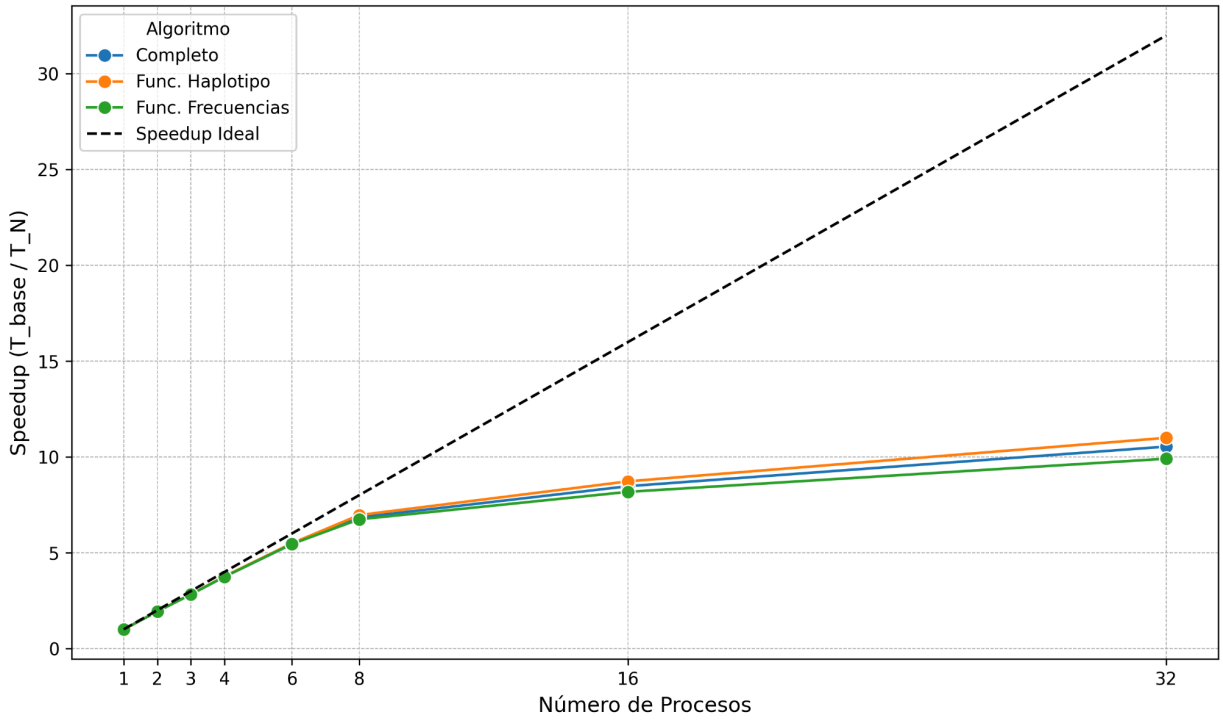
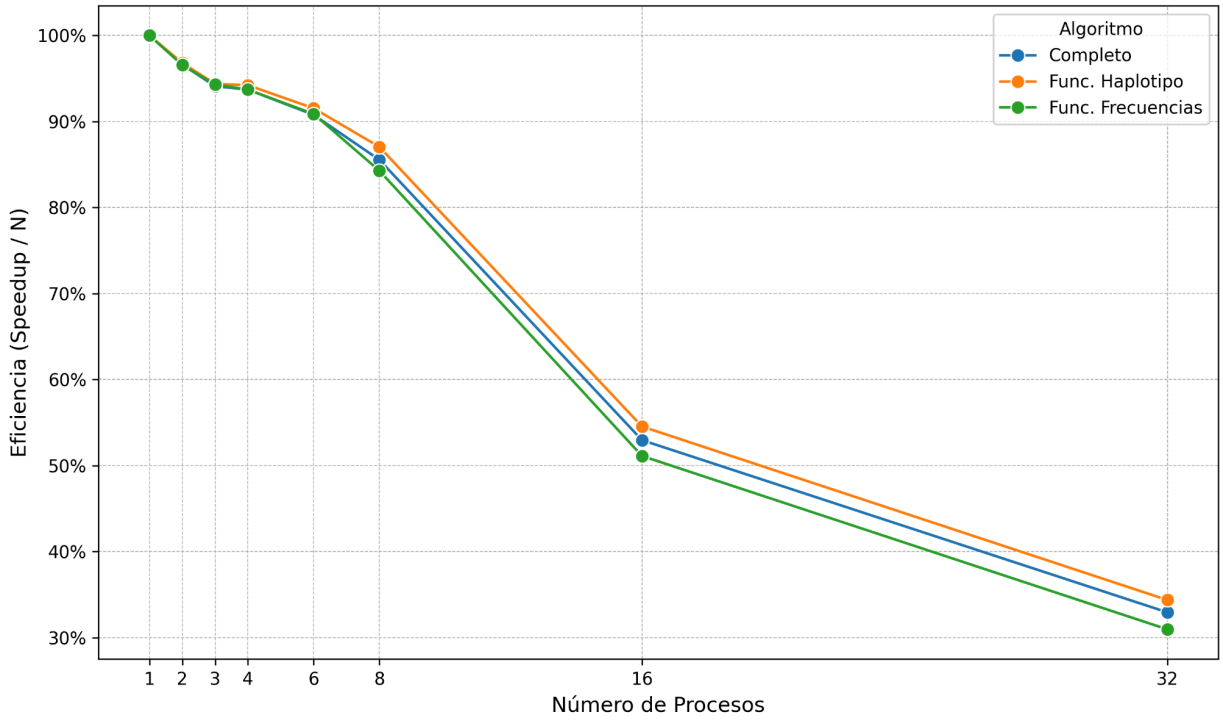


Figura 6: Gráfico de eficiencia para las funciones desagregadas, utilizando 3000 individuos y 12 posiciones.



Conclusiones

Los resultados de este trabajo permiten obtener varias conclusiones, la paralelización del algoritmo de haplotipado mediante la biblioteca multiprocessing de Python demostró ser una estrategia altamente exitosa. Se lograron reducciones significativas en los tiempos de ejecución, validando el enfoque para abordar un problema computacionalmente intensivo.

También se ha demostrado empíricamente que el rendimiento de la paralelización, medido tanto en speedup como en eficiencia, mejora sustancialmente con el aumento del tamaño del problema (más individuos) y de su complejidad computacional (más posiciones). Esto se debe a que el overhead asociado a la gestión de procesos se amortiza de manera más efectiva en cargas de trabajo de mayor granularidad.

La elección del número de procesos dependerá de qué objetivo queremos apuntar de acuerdo a la performance, tiempo o eficiencia del uso de procesadores. Sin embargo, podemos concluir que a partir de los resultados expuesto se puede apreciar que el speedup se incrementa linealmente hasta llegar a los 8 procesadores, y más arriba el incremento se reduce drásticamente. Coincidentemente este punto es similar a la reducción de la eficiencia, aunque de una manera menos abrupta. Con esto en mente creemos que paralelizar en 8 procesadores es un buen balance entre ambas métricas

Cabe destacar que si bien la paralelización fue efectiva, el speedup obtenido fue sub-lineal, lo que indica la presencia de un overhead inevitable. Este comportamiento es un ejemplo empírico de la Ley de Amdahl, que postula que la porción secuencial de un programa (en este caso, carga de datos, creación del Pool de procesos y agregación final de resultados) impone un límite superior teórico al speedup alcanzable.

Limitaciones y Trabajo Futuro

La implementación actual, aunque optimizada, presenta limitaciones inherentes a su diseño algorítmico, el enfoque de fuerza bruta para generar y evaluar todas las combinaciones posibles se vuelve computacionalmente intratable para un gran número de

posiciones (p. ej., 32 o más), ya que el espacio de búsqueda crece exponencialmente (Qin et al., 2002).

A nivel biológico, a medida que aumenta el número de marcadores analizados, también lo hace la distancia física entre ellos en el cromosoma. Esto puede debilitar la correlación genética (desequilibrio de ligamiento), complicando la inferencia precisa de haplotipos largos (Li & Stephens, 2003; McVean & Cardin, 2005).

Se propone la investigación e implementación de algoritmos más avanzados que no dependan de un análisis exhaustivo de todas las combinaciones. Una solución prometedora es trabajar con bloques haplotípicos: regiones de fuerte correlación genética que pueden ser inferidas eficientemente (Browning & Browning, 2007; Scheet & Stephens, 2006). Posteriormente, estos bloques pueden ser ensamblados para reconstruir haplotipos de regiones cromosómicas más grandes, permitiendo un análisis escalable y biológicamente más robusto.

Bibliografía

• Browning, S. R., & Browning, B. L. (2007). Rapid and accurate haplotype phasing and missing-data inference for whole-genome association studies by use of localized haplotype clustering. The American Journal of Human Genetics, 81(5), 1084–1097.

• Li, N., & Stephens, M. (2003). Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. Genetics, 165(4), 2213–2233.

• McVean, G. A., & Cardin, N. J. (2005). Approximating the coalescent with recombination. Philosophical Transactions of the Royal Society B: Biological Sciences, 360(1459), 1387–1393.

• Qin, Z. S., Niu, T., & Liu, J. S. (2002). Partition-ligation-expectation-maximization algorithm for haplotype inference with single-nucleotide polymorphisms. *The American Journal of Human Genetics*, *71*(5), 1242–1247.

• Roach, J. C., Glusman, G., Smit, A. F., Huff, C. D., Hubley, R., Shannon, P. T., ... & Galas, D. J. (2010). Analysis of genetic inheritance in a family quartet by whole-genome sequencing. Science, 328(5978), 636–639.

• Scheet, P., & Stephens, M. (2006). A fast and flexible statistical model for large-scale population genotype data: Applications to inferring missing genotypes and haplotypic phase. The American Journal of Human Genetics, 78(4), 629–644.

• Tewhey, R., Bansal, V., Torkamani, A., Topol, E. J., & Schork, N. J. (2011). The importance of phase information for human genomics. Nature Reviews Genetics, 12(3), 215–223.