Arquitectura de Computadores

Práctica de programación orientada al rendimiento

Proyecto de programación paralela

Grupo 82, equipo 5

Marco Antonio Merola 100413665, Adrian Perez Ruiz 100429044,

Victor Herranz Sanchez 100429052, Alejandro Gonzalez Nuñez 100429135

Índice

1. Decisiones de diseño:

En este apartado de la práctica, vamos a explicar las partes más importantes de nuestro código y las diferentes decisiones que hemos tomado a la hora de paralelizar diferentes secciones del código.

1.1. Creación de hilos:

En nuestro programa decidimos crear hilos en función a una cantidad fija en el código, en la versión entregada son 16, cada uno con su respectiva identificación. Sin embargo cabe destacar que este número de hilos se podría cambiar pero siempre debe ser un número potencia de 2. La creación de dichos hilos se hace al inicio de la paralización del código y no se crean más en ningún momento. Esto lo hemos decidido así ya que es ineficiente estar creando hilos dentro de la región paralela (se crearían muchos hilos). Ya que estaríamos creando una gran cantidad de hilos inútiles, y podemos utilizar los mismos en todo momento.

Cada hilo se va a encargar de calcular las diferentes fuerzas, velocidades, aceleraciones y posiciones de un número determinado de objetos. De está manera lo que hacemos es dividir los cálculos (la parte más costosa de ejecutar de nuestro programa) en los diferentes hilos. Más adelante veremos que esto provoca mejoras sustanciales en los tiempos de ejecución del programa.

1.2. Creación de vectores:

Con el propósito de no cambiar los valores respecto a la versión original, se agregaron una serie de vectores con dos dimensiones, una para identificar el thread y otra para el objeto. Dichos vectores auxiliares tienen el principal propósito de ir almacenando los cálculos que van realizando de forma paralela cada uno de los hilos para luego ser actualizados en el objeto correspondiente. En total se tuvieron que crear 12 vectores, estos 12 vectores están relacionados con los diferentes atributos de los objetos (posiciones, fuerzas, velocidades y aceleraciones de cada eje). Estos vectores son inicializados al principio de la paralelización y son utilizados a lo largo de toda la simulación.

1.3. Paralelización del cálculo de fuerzas:

Para hacer el cálculo de las fuerzas , utilizamos hilos creados al principio de la simulación. Estos hilos se repartirán las fuerzas a calcular entre los diferentes objetos (para nuestra simulación dejamos que OpenMP las distribuya las diferentes iteraciones de forma predeterminada), es decir, se utilizaran x hilos y cada hilo calculará los objetos que se les asignen. Se deben guardar las fuerzas calculadas a cada objeto de su respectivo hilo, para después sumarle el valor a los diferentes objetos

1.4. Sincronización de los resultados de las fuerzas:

Con la intención de no variar los valores de la ejecución original y de que los resultados no varíen respecto a otra ejecución, utilizamos los vectores mencionados anteriormente. Esta sincronización, se hace de manera secuencial, de esta manera, el objeto 1 siempre tendrá los mismos valores da igual cuantas veces se ejecute el mismo programa (con los mismos parámetros de entrada). Además, como se hace de forma secuencial, nos dará los mismos resultados que en la simulación original sin paralelizar. Estos bucles se podrían paralelizar sin mucho problema, pero al ser operaciones en coma flotante, estos valores pueden variar además de definir diferentes mutex o zonas críticas que generan paradas y "ralentizan" el código. Por lo que decidimos dejar este bucle anidado de forma secuencial.

1.5. Paralelización del cálculo de aceleraciones, velocidades y posiciones:

Esta fase es muy parecida a la fase de cálculo de fuerzas. Aquí se reparten los hilos los diferentes objetos que tenemos y se hacen los cálculos de dichos objetos. El resultado de los cálculos se guardan en los vectores. Posteriormente, se utilizarán para actualizar los valores de los diferentes objetos. Cabe destacar que el orden en que se ejecutan cada una de las operaciones es la más óptima para obtener un menor número de fallos en caché, pero que al hacerlo de forma paralela es posible que haya una cantidad diferente de fallos cada vez que se ejecute, lo que puede variar los tiempos de ejecución de las diferentes pruebas.

1.6. Sincronización de los resultados de las aceleraciones, velocidades y posiciones:

La sincronización de los resultados es en esencia el mismo procedimiento que la sincronización de las fuerzas, es decir, utilizaremos los vectores auxiliares en donde se han guardado los resultados de los cálculos y de forma secuencial se irán almacenando en los correspondientes objetos para así obtener los mismos resultados que en la versión anterior. Además, en esta fase de sincronización, se realizan los diferentes rebotes que tienen los objetos cuando sus posiciones son negativas o mayores al tamaño de recinto, cambiando también el sentido de la velocidad cuando ocurre algún rebote (multiplicar por -1).

1.7. Paralelización de las colisiones:

Es de suma importancia tener en cuenta que en ciertos procedimientos no conviene hacerlos de forma paralela. Principalmente por cómo este afecta al rendimiento. En nuestro caso no hemos paralelizado el proceso de comprobación de coaliciones entre los objetos debido a que es ineficiente y se generan diferencias entre los cálculos originales en la versión no paralelizada.

2. Implementación:

2.1. Implementación de PAOS y PSOA:

No hay variaciones en cuanto a la implementación de la paralización de ambos programas, las diferencias son las discutidas y analizadas en el proyecto 1 entre SOA y AOS, es por ello que la implementación de la librería de OpenMP en ambas es prácticamente la misma.

Nuestro código mantiene casi toda la estructura y se realizan los mismos cálculos que en el proyecto 1, es decir, principalmente hemos hecho modificaciones solamente para poder hacer que el código se pueda paralelizar. No hay ningún cambio que hayamos hecho con respecto a la versión anterior que destaque mencionar que no esté relacionado con el proceso de paralelización con OpenMP.

Para lograr de forma correcta la implementación de los 2 programas de forma paralela se tuvieron en cuenta los siguientes puntos:

Las fuerzas no son acumulativas entre iteraciones, es decir tuvimos que implementar que las fuerzas se acumulan durante una iteracion pero cuando se pasa a la siguiente no son acumulativas por lo que se resetean a 0. La velocidad y posición son acumulativas entre iteración, esto se debe a la propia fórmula, por ende no aplicamos “+=” [sin reseteo por iteración] sino que se usa “=” por otra parte la aceleración no es acumulativa entre iteraciones tampoco por lo que se emplea “=” pero tenemos que calcularlas nuevamente por cada iteración

Además hay que tener en cuenta el uso de “=” en los vectores de posición velocidad y aceleración entre hilos, ya que en diferentes iteraciones pueden asignarse distintos objetos al mismo hilo, y debido a esto usar “+=” podría generar incongruencias en los resultados esperados.

Existen varios bucles que no pueden ser paralelizados, debido a que se está trabajando con operaciones en coma flotante y los resultados podrían verse afectado por ello, sobre todo cuando sincronizamos los cálculos obtenidos nos vemos forzados a hacerlo de forma secuencial para ir guardando los resultados en el mismo orden de operación que se hacían en la versión secuencial.

Decidimos que el reparto entre hilos se hiciera de forma automática por parte de OpenMP y no de forma manual, esto tiene distintas implicaciones pero la principal diferencia es que se pudiera realizar de forma eficiente teniendo en cuenta que algunos hilos pueden acabar antes que otros.

Se realizó un Intento de paralelizar colisiones, el principal problema de esto es que es una comprobación de cada objeto con otro y al nosotros estar realizado la paralelización desde las interacciones afectaba al rendimiento y valores no eran exactos, después de realizar una prueba de región crítica con mutex para evitar concurrencia nos dimos cuenta que no aportaba un beneficio de rendimiento considerable sino que más bien hacia que los

datos fueran ligeramente distintos a la versión secuencial. Es por esto que no se implementó dicha paralelización

3. Evaluación de rendimiento:

Para evaluar el rendimiento de las versiones paralelas de nuestro programa, hemos utilizado como parámetros el tamaño de recinto, la semilla, número de iteraciones, número de objetos y tiempo. Para las diferentes ejecuciones, variamos las iteraciones y el número de objetos. Pero para el resto se mantienen constantes. Estos valores son:

- Time step: 0.1
- Tamaño del recinto:1.000.000
- Semilla: 4.

Características del ordenador: modelo de procesador: Ryzen 7 3700U, número de cores físicos 4, Número de subprocesos 8, tamaño de memoria principal 8gb, jerarquía de la memoria caché: L1d 128 KiB, L1i 256 KiB, L2 2MiB, L3 4MiB , software del sistema Ubuntu 20.04 LTS

Link de caracterizticas del CPU, https://www.amd.com/es/products/apu/amd-ryzen-7-3700u

3.1. Resultados AOS:

Resultados:

Población	Iteraciones	Media (s)
4000	250	30,0851
4000	500	60,1355
8000	250	121,5343
8000	500	250,1492

3.2. Resultados SOA:

Resultados:

Población	Iteraciones	Media (s)
4000	250	32,53918182
4000	500	64,53088889
8000	250	131,31
8000	500	263,3647

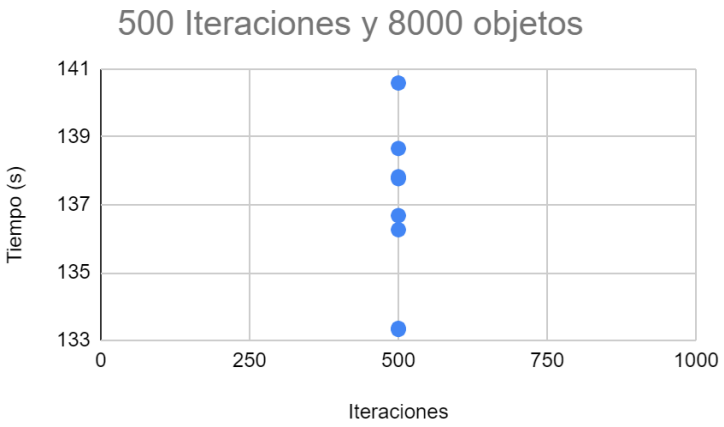
3.3. Resultados PAOS:

1 Hilo:

2 Hilos:

4 Hilos

8 Hilos:



Resultados:

Población	Iteraciones	Hilos	Media(s)	Speedup respecto a AOS	Speedup respecto a SOA
4000	250	1	32,2932	0,9316233758	1,00761714
4000	500	1	63,6289	0,945097275	1,014175774
8000	250	1	129,1129	0,9413025344	1,01701689
8000	500	1	266,2501	0,9395271589	0,989162821
4000	250	2	23,6774	1,270625153	1,374271745
4000	500	2	43,6783	1,376782063	1,477413015
8000	250	2	93,9177	1,294051068	1,398139009
8000	500	2	195,0349	1,28258686	1,350346528
4000	250	4	17,4969	1,71945316	1,859711253
4000	500	4	33,5395	1,792975447	1,924026562
8000	250	4	61,3748	1,980198713	2,13947744
8000	500	4	134,6192	1,858198533	1,956368037
4000	250	8	17,177	1,751475811	1,894346034
4000	500	8	32,4028	1,855873566	1,991521995
8000	250	8	58,0165	2,094823024	2,263321641
8000	500	8	135,5931	1,844851987	1,942316386
4000	250	16	14,859	2,024705566	2,189863505
4000	500	16	29,7529	2,021164323	2,168894087
8000	250	16	55,9571	2,171919202	2,346619106
8000	500	16	136,2477	1,835988424	1,932984557

3.4. Resultados PSOA:

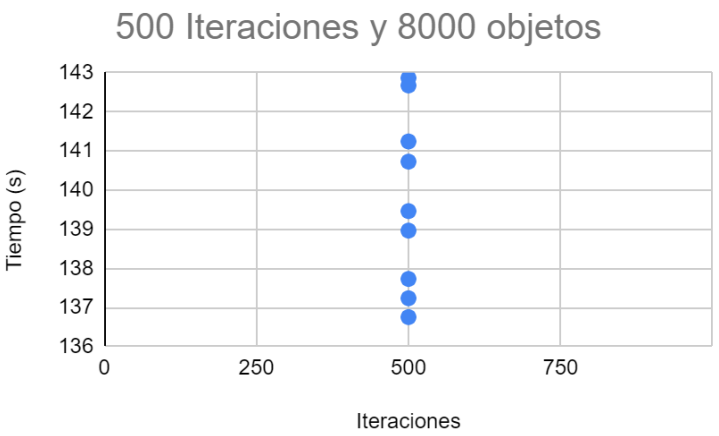
1 Hilo:

2 Hilos:

4 Hilos:

8 Hilos:

16 Hilos:

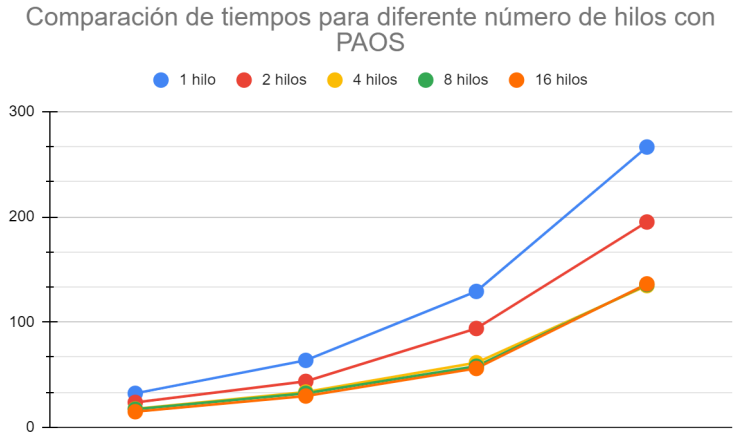


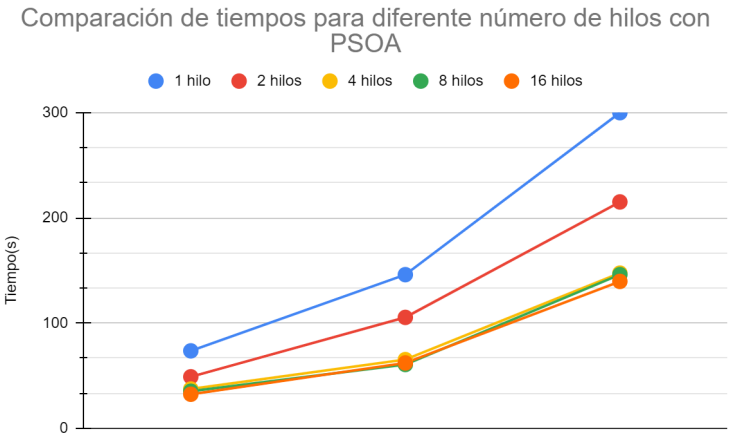
Resultados:

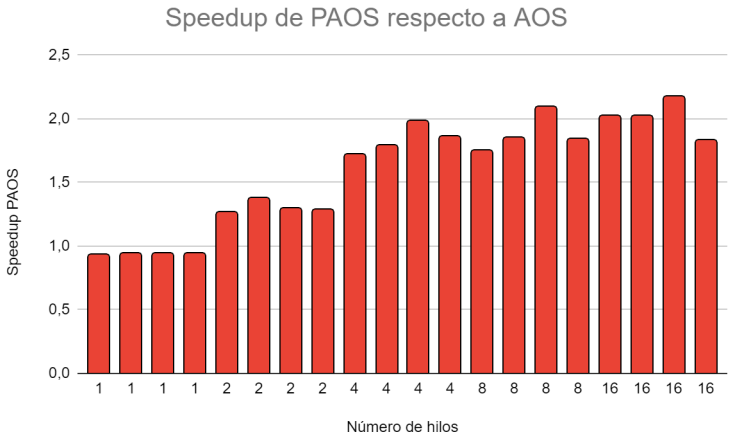
Población	Iteraciones	Hilos	Media (s)	Speedup respecto a AOS	Speedup respecto a SOA
4000	250	1	36,203	0,8310112422	0,8987979399
4000	500	1	73,6044	0,8170095809	0,8767259687
8000	250	1	145,9789	0,8325470325	0,8995135598
8000	500	1	299,8261	0,8343142909	0,8783915076
4000	250	2	25,1147	1,197907998	1,295622955
4000	500	2	48,9746	1,227891601	1,317639938
8000	250	2	105,3983	1,153095448	1,245845521
8000	500	2	215,0505	1,163211432	1,224664439
4000	250	4	17,9246	1,678425181	1,815336566
4000	500	4	37,3914	1,608270886	1,725821683
8000	250	4	65,3019	1,861114301	2,010814387
8000	500	4	147,4154	1,696900053	1,786548081
4000	250	8	17,6205	1,707391958	1,846666202
4000	500	8	35,2516	1,705894201	1,830580424
8000	250	8	60,6639	2,003404001	2,164549262
8000	500	8	146	1,713350685	1,803867808
4000	250	16	16,323	1,843110948	1,993455971
4000	500	16	32,4099	1,855467002	1,991085714
8000	250	16	62,1023	1,957001593	2,114414442
8000	500	16	139,5823	1,792126939	1,886805849

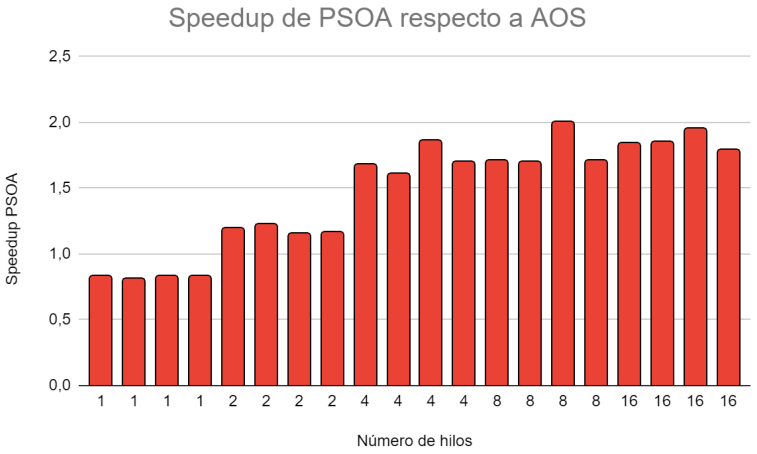
3.5. Resultados globales:

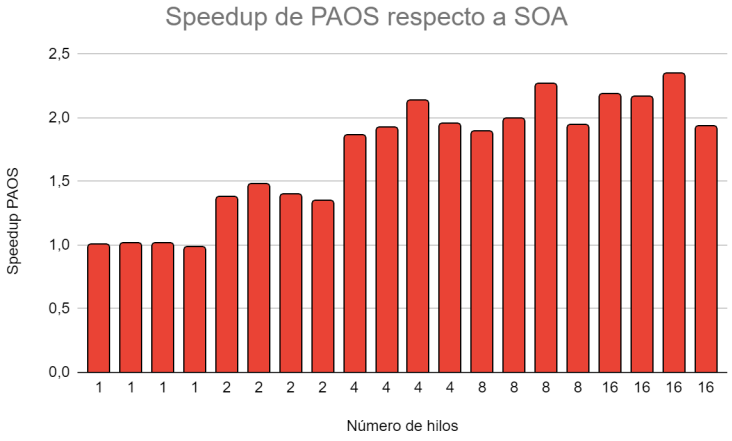
En las gráficas, cada punto se refiere a una de las ejecuciones. Es decir, el primer punto se refiere a 4000 objetos y 250 iteraciones, el segundo 4000 y 500 iteraciones, el tercero a 8000 y 250 iteraciones y el último punto son 8000 y 500 iteraciones. Exactamente lo mismo para el diagrama de barras.

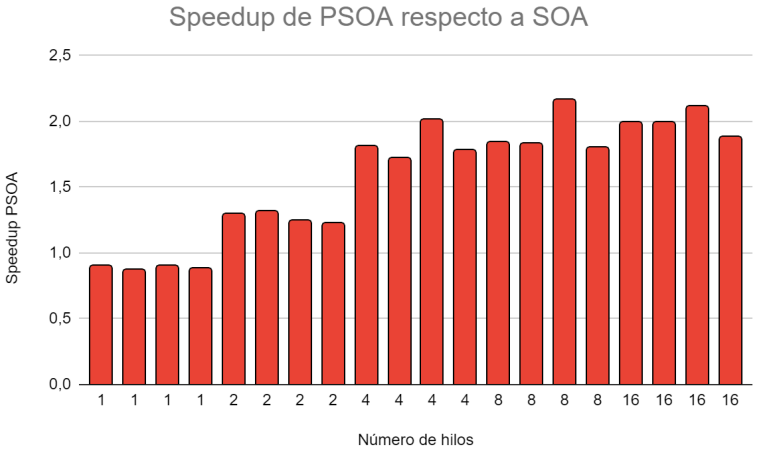


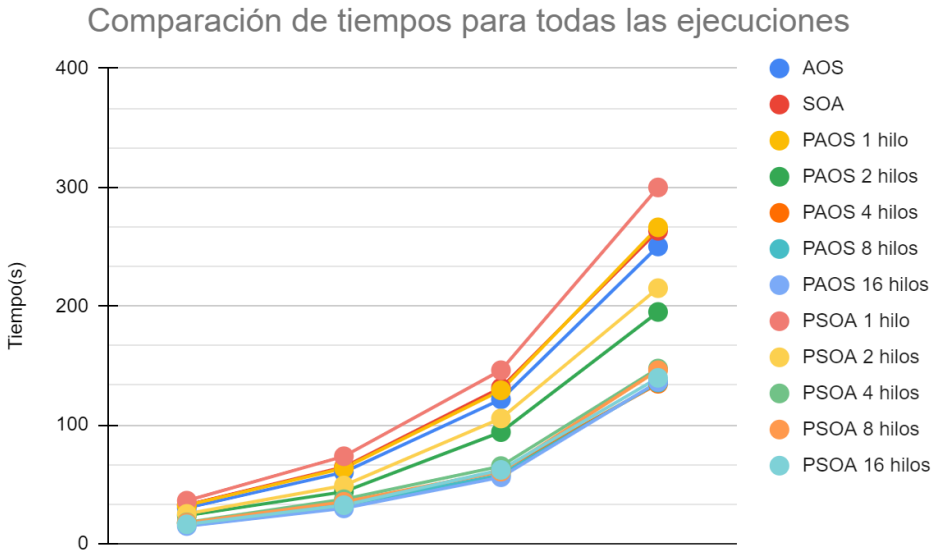












3.6. Análisis de resultados:

Se puede observar claramente que PAOS es la versión que mejores resultados obtiene. Así también debido a que hemos paralelizado de forma correcta, mientras más hilos empleamos mejores son los resultados. También, queremos destacar como disminuye el tiempo cuantos más hilos, pero llega un momento en el que ya la diferencia es mínima. Para nuestro caso, el umbral esta en 4 hilos, es decir, a partir de ese número de hilos, la mejora es sustancialmente menor. Esto está relacionado con el número de hilos que es capaz de soportar nuestra máquina. Si tuvieras un mayor número de procesadores, el tiempo seguramente se hubiera reducida en mayor medida incluso.

Cuando el número de objetos e interacciones son pequeños los speedups son menores, debido a que la paralización es mucho más efectiva cuando se deben realizar muchas operaciones. Logramos obtener speedups de hasta 2.34 con PAOS 16 Hilos respecto a SOA.

En relación a la comparación de PSOA vs PAOS observamos que los resultados siguen el patrón de SOA vs AOS en el proyecto 1, es decir vemos que no hay una diferencia muy grande entre ellos sin embargo PAOS es ligeramente más rápido respecto a PSOA, esto se debe a que PSOA tiene un mayor número de fallos en caché que genera más lecturas de memoria principal, lo que provoca menor rendimiento.

Para finalizar, queremos destacar como la paralelización carece de sentido cuando solo hay un hilo. Esto ocurre debido a que en nuestro código, tenemos un par de bucles más, que generan mayor carga de trabajo, y por lo tanto, peor rendimiento. Pero, no tiene sentido desarrollar una versión paralela, si solo tenemos un hilo, ya que es como si ejecutaras en .

4. Conclusión:

Para culminar este proyecto hemos llegado a una serie de conclusiones basándonos tanto en las investigaciones realizadas como en el análisis de los resultados obtenidos.

Habiendo aplicado una serie de implementaciones podemos concluir que la paralelización de un programa es una tarea complicada pero que sin embargo tiene la capacidad de aumentar considerablemente el rendimiento de un programa. Es importante tener en cuenta cual es la meta de un programa para poder saber cómo paralelizar, por lo que aunque conlleve más tiempo y sea más abstracto, a la hora de paralelizar es importante ver dónde está el “bottleneck” en un programa, es decir, donde pasa la mayor cantidad de tiempo nuestro programa, lo que en la mayoría de los casos serán los bucles, así también usar herramientas que te permitan evaluar tanto la memoria caché como el tiempo de ejecución durante la evaluación del rendimiento.

OpenMP es una librería muy completa y bien estructurada que nos permite paralelizar es una forma sencilla y eficaz. Ha sido indispensable para este proyecto utilizar diversas funciones de OpenMP, sin embargo también se deberá ajustar el programa con vectores auxiliares y funciones para sincronizar los datos dependiendo del caso.

A la hora de comparar PAOS vs PSOA, como se ha visto en la sección 3 de esta memoria podemos observar que no hay grandes diferencias en el rendimiento ya que hemos aplicado muchas optimizaciones para que en ambos sea lo más eficiente posible. Sin embargo comparando estas 2 con SOA y AOS vemos un speedup importante para todos los casos que hemos evaluado.

Como conclusión final, podemos decir que la optimización de un código no es tarea para nada sencilla y conlleva su tiempo. En esta asignatura hemos visto cómo llevarlas a cabo, ya sea paralelizando el código (esta parte del proyecto) o generando menor tasa de fallos en el código (primera parte del proyecto). Por lo que decidir una buena estructura de datos y cómo acceder a ellos es importante, y también dividir el programa en diferentes hilos para que se puedan ejecutar a la vez sin variar cambios y en menor tiempo.