PRÁCTICA 1: PROGRAMACIÓN PARALELA CON OPENMP

ARQUITECTURA DE COMPUTADORES

Curso 2015-2016

Grado en Ingeniería Informática

Universidad Carlos III de Madrid

JAVIER PRIETO CEPEDA: **100 307 011**

SANDRA JUÁREZ PUERTA: **100 303 528**

ÍNDICE

TABLAS DE RESULTADOS

ÍNDICE DE GRÁFICAS

1 Introducción

En este documento se procede a la explicación de la práctica 1, *"Programación paralela con OpenMP"* de la asignatura Arquitectura de Computadores del grado en Ingeniería Informática de la Universidad Carlos III de Madrid. La realización de la misma la han llevado a cabo los alumnos Javier Prieto Cepeda y Sandra Juárez Puerta.

En primer lugar, se describirá el código, tanto en la versión secuencial proporcionada inicialmente como en la versión paralelizada, indicando las dos versiones de código realizadas. Posteriormente, se detallarán las decisiones de diseño pertinentes para la correcta implementación de la práctica y obtención de una mejora significativa en la optimización del código.

Una vez descrito el código, se realizará una evaluación de las versiones secuencial y paralela. En ambos casos, se realizará un doble análisis, observando en primer lugar el comportamiento con la variación del número de cuerpos y en segundo lugar el comportamiento con la variación del número de iteraciones. Cabe destacar, que de la versión paralela además, se detallará en cada uno de los dos comportamientos, la evaluación variando el número de hilos y viendo el impacto que éstos tienen sobre el comportamiento de la ejecución.

Posteriormente, se realizará un análisis del impacto que tiene la planificación de la paralelización, con los modelos *static, dynamic y guided,* analizando como en el apartado anterior, el comportamiento en primer lugar variando el número de cuerpos, y en segundo lugar variando el número de iteraciones Además, para cada uno de los comportamientos, se evaluará el comportamiento variando el número de threads y viendo el impacto que éstos tienen sobre el comportamiento de la planificación.

Por último, se realizarán una serie de valoraciones y conclusiones sobre la realización de la práctica.

2 Referencias

[1] Vector class:

http://www.cplusplus.com/reference/vector/vector/

[2] "What every computer scientist should know about floating-pointarithmetic". David Goldberg:
http://dl.acm.org/citation.cfm?id=103163

[3] High_resolution_clock class:
http://en.cppreference.com/w/cpp/chrono/high_resolution_clock

[4] Intel Core i5 3570:

http://ark.intel.com/es-es/products/65702/Intel-Core-i5-3570-Processor-6M-Cache-up-to-3_80-GHz

3 Descripción del código

En este apartado, se va a describir el código de la práctica, describiendo en primer lugar la primera versión realizada, y posteriormente la versión final.

En esta práctica, se realiza una implementación del comportamiento en el espacio de los n-cuerpos. Éstos, contarán con una serie de propiedades, como la posición en cada instante en los ejes X e Y, la velocidad en los ejes X e Y, y la masa que posee cada cuerpo. Estos cuerpos, interactuarán entre sí en cada iteración, viendo modificadas sus posiciones y velocidades.

El código inicial, cuenta con dos versiones. Una primera llamada ***AOS*** (Array of structures), en la que cada cuerpo se almacena en una posición de un vector[1] (tipo de estructura de datos de la clase vector de la biblioteca estándar de c++) y una segunda llamada ***SOA*** (Structure of Arrays), en la cual cada propiedad de los cuerpos queda almacenada en su vector correspondiente, es decir, existirán un vector para cada atributo de los cuerpos, y las propiedades de cada cuerpo quedará almacenada en la posición iésima de cada array, correspondiente los atributos del cuerpo n con la posición n de cada vector.

El objetivo de la práctica, es obtener un speed-up que mejore el tiempo de ejecución del código en su versión secuencial. Para ello, en primer lugar, se ha realizado un análisis para detectar cuáles son las partes del código más pesadas computacionalmente hablando y que podrían ser paralelizadas para obtener una mejora de rendimiento.

3.1 Optimización inicial

Una vez analizado el código, se ha detectado una sección del código en la cual se puede obtener un speed-up considerable. Esta sección del código, se encontrará en los ficheros **"bodiesaos.cpp"** y **"bodiessoa.cpp"**, incluidos en el directorio ***"src"*** del código proporcionado.

En un primer momento, se ha realizado una optimización en la ejecución de la función **"compute_forces"** de ambos ficheros.

3.1.1 AOS

La optimización realizada sobre la versión de array de estructuras, ha sido mediante la adición de un ***#pragma omp parallel for*** en el código que se encarga de la paralelización del bucle interno, que se encarga de iterar con la variable ***"j"***. Al realizar la paralelización del bucle interno, no es necesario poner ninguna variable como privada, puesto que aquellas que se desean mantener como copias privadas del hilo, quedan declaradas dentro del scope del hilo, por lo que ya son privadas. De esta forma, cada hilo compartirá la variable de iteración ***"i"*** correspondiente al bucle superior, lo cual conlleva a que las escrituras realizadas en los elementos iésimos de los vectores en cada hilo, han de estar protegidas, para que no se produzcan condiciones de carrera, por lo que será necesario insertar en una sección critica las escrituras en las posiciones iésimas de los vectores. Esta sección crítica se declarará mediante la sentencia de openMP ***#pragma omp critical***. Concretamente, ha de protegerse la escritura en la posición iésima del vector **"forces"**. A continuación, se indica la escritura que debe quedar introducida en la sección crítica:

```
#pragma omp critical
forces[i]     += deltaf;
```

3.1.2 SOA

La optimización realizada sobre la versión de estructura de arrays, ha sido mediante la adición de un ***#pragma omp parallel for*** en el código que se encarga de la paralelización del bucle interno, que se encarga de iterar con la variable ***"j"***. Al realizar la paralelización del bucle interno, no es necesario poner ninguna variable como privada, puesto que aquellas que se desean mantener como copias privadas del hilo, quedan declaradas dentro del scope del hilo, por lo que ya son privadas. De esta forma, cada hilo compartirá la variable de iteración ***"i"*** correspondiente al bucle superior, lo cual conlleva a que las escrituras realizadas en los elementos iésimos de cada uno de los vectores en cada hilo, ha de protegerse para que no se produzcan condiciones de carrera, por lo que será necesario insertar en una sección critica las escritura en la posiciones iésimas de los vectores. Esta sección crítica se declarará mediante la sentencia de openMP ***#pragma omp critical***. Concretamente, ha de protegerse en esta sección crítica la escritura en la posición iésima de los vectores **"xforces"** e **"yforces"**.

A continuación, se indican las escrituras que deben quedar introducidas en la sección crítica:

```
#pragma omp critical
{
  xforces[i]      += xdeltaf;
  yforces[i]      += ydeltaf;
}
```

3.2 Optimización final

Tras realizar una primera versión de la paralelización del código, y analizar el comportamiento e impacto de la misma en el speed-up, se ha decidido realizar una optimización. Observando que la adición de secciones críticas en cada hilo produce una serie de stalls con sus correspondientes demoras en tiempo de ejecución, se ha estudiado la posibilidad de cómo poder eliminar las secciones críticas.

Estas secciones críticas, se encargan de proteger las escrituras en los vectores de fuerzas, pero se puede observar, que cada una de las escrituras, se encarga de ir acumulando valores de fuerzas en cada una de las iteraciones. Una opción, sería realizar una optimización de tipo **"reduction"**, de forma que cada hilo pudiera acumular los valores de forma privada, y realizando al final de la ejecución la acumulación total. A continuación, se describe la manera de lograr esta optimización en cada uno de los tipos.

3.2.1 AOS

En esta versión, para llevar a cabo la optimización de reduction, es necesario observar que la acumulación se realiza sobre el vector **"forces"**. Este vector, almacena objetos de tipo **"phys_vector"**. Este tipo de objetos, cuenta con 2 atributos de tipo double. En un primer momento, no se podría realizar la optimización de **"reduction"**, puesto que no se permite su aplicación sobre estructuras de datos como lo son los vectores o los arrays. Pero teniendo en cuenta el detalle anteriormente mencionado, si definimos dos variables auxiliares en el scope del bucle **"i"** de tipo double, que se encarguen de acumular las fuerzas en las posiciones iésimas que comparten todos los hilos, y aplicamos la reducción sobre éstas variables auxiliares (que sí nos permite openMP), logramos evitar las secciones críticas, con la mejora correspondiente. Para ello, una vez finalizan todos los hilos, fuera de la sección del bucle interno, se

realiza la escritura en la posición iésima del vector **"forces"**, de forma secuencial y evitando condiciones de carrera.

3.2.2 SOA

En esta versión, para llevar a cabo la optimización de reduction, es necesario observar que las acumulaciones se realizan sobre los vectores **"xforces"** e **"yforces"**. Estos vectores, almacenan variables de tipo double. Al igual que en la versión ***AOS*** y teniendo en cuenta el detalle anteriormente mencionado, si definimos dos variables auxiliares en el scope del bucle **"i"** de tipo double, que se encarguen de acumular las fuerzas en las posiciones iésimas que comparten todos los hilos, y aplicamos la reducción sobre éstas variables auxiliares, logramos evitar las secciones críticas, con la mejora correspondiente. Para ello, una vez finalizan todos los hilos, fuera de la sección del bucle interno, se realiza la escritura en la posición iésima de los vectore **"xforces"** e **"yforces"**, de forma secuencial y evitando condiciones de carrera.

3.3 Decisiones de diseño

En cuanto a las decisiones de diseño del código contempladas, se pueden destacar las siguientes:

- Se ha decidido la paralelización del bucle interno, debido a que gracias a ello, se puede realizar el **"reduction"** indicado en la versión final, evitando la pérdida de tiempo provocado por las secciones críticas.
- Con respecto a la optimización inicial realizada tanto sobre las versiones ***AOS*** y ***SOA***, se obtiene un mejor speed-up paralelizando el bucle externo.
- El motivo de la paralelización del bucle interno, es la simplicidad a la hora de realizar el código, además de aprovecha el *pool* de hilos que openMP crea, ya que si no se varía el número de hilos a lo largo del programa, el pool de hilos se mantiene, de forma que al no tener que crear y destruir el *pool* de hilos en cada iteración del bucle externo, se obtiene un buen speed-up.
- Se ha considerado también, la paralelización del bucle externo, aplicando un "**reduction"** para mejorar los tiempos. Esta implementación, consistiría en la utilización de vectores auxiliares en cada hilo que acumulen las fuerzas. Se ha

desechado esta opción por la complejidad que entrañaría, ya que no es el principal objetivo de la práctica.

- Se contemplado la posible paralelización de la función **"apply_forces"**, pero dicha implementación ha quedado descartada debido a su escaso impacto en la escala máxima pedida en la práctica.

4 Evaluación del rendimiento

A continuación, se procederá a evaluar el rendimiento de las versiones tanto secuencial como paralela. La ejecución de ambas versiones, ha sido realizada sobre una máquina con las siguientes características:

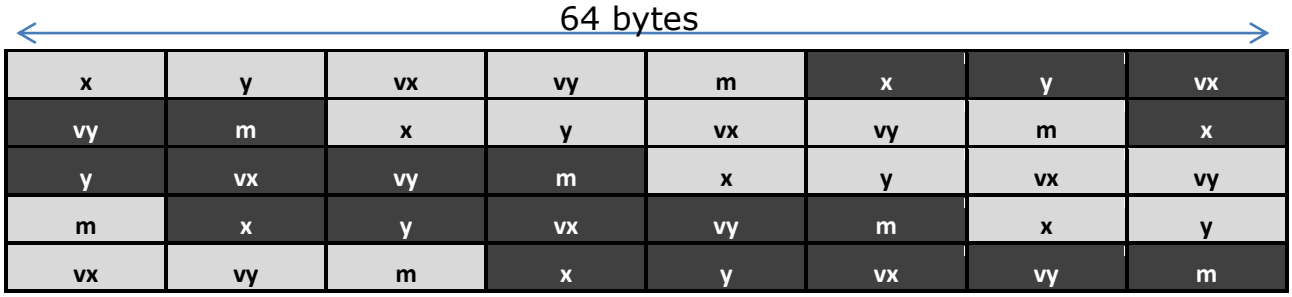
- Procesador: Intel core i5 3570 3,4 Ghz. [4]
  - 4 cores.
  - Cache L1:4 x 256 KB 8 vías asociativa.
  - Cache L2: 4 x 1024 KB 8 vías asociativa.
  - Cache L3: 6 MB 12 vías asociativa y compartida.
- Memoria Principal: 8 GB.
- Sistema operativo Ubuntu 14.04 64 bits.
- Versión compilador: gcc 4.8.

Otro dato importante y necesario a tener en cuenta, son los flags y optimizaciones indicados al compilador para la compilación del código. Éstos son los siguientes:

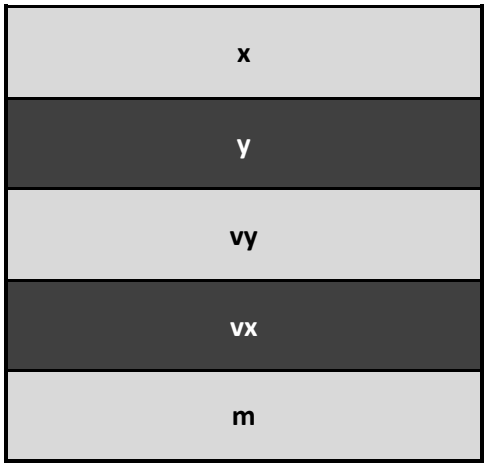
- O3: Activa las optimizaciones más costosas del compilador, tales como las funciones en línea. Además, incorpora todas las optimizaciones de los niveles inferiores como O2 y O1.

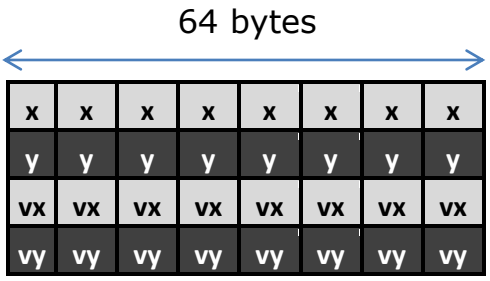
Para comprender mejor el análisis del rendimiento, es de ayuda explicar cómo se organizan en memoria las diferentes versiones.

- **AOS**: En la versión de ***AOS***, tenemos en cada posición del vector, 5 elementos de 8 bytes cada uno. Esto supone, que cada cuerpo, tiene un tamaño de 40 bytes, menor al tamaño de una línea de caché, que es de 64 bytes. Por tanto, en cada línea, seremos capaces de almacenar 8 variables del vector. La organización en caché sería la siguiente:



- **SOA**: En la versión de ***SOA***, tenemos 5 vectores de con variables de tipo doublé. Esto supone, que entre atributos de un mismo cuerpo, no exista una buena proximidad en memoria, ya que ésta dependerá del número de cuerpos que se indiquen. La organización en memoria principal y caché, con n-cuerpos, sería la siguiente:





Es necesario indicar, que las mediciones de los tiempos se han realizado desde el comienzo hasta el final de la simulación de los n-cuerpos, obteniendo todos los resultados con una precisión de nanosegundos. La medición de estos tiempos, ha sido realizada mediante la clase high_resolution_clock[3] de la librería estándar de c++.

Tanto para la versión secuencial como para la versión paralelizada, se ha seguido la misma estructura de pruebas. En primer lugar, se muestran los tiempos obtenidos en media tras la medición de la ejecución manteniendo un número de 100 iteraciones y variando el número de cuerpos en 250 unidades desde 250 cuerpos hasta 1000 cuerpos, ambos inclusive. En segundo lugar, se muestran los tiempos obtenidos tras la ejecución manteniendo un número de 50 cuerpos y

variando el número de iteraciones en 50 unidades desde 50 iteraciones hasta 200, ambas inclusive. Además, se indican para la versión paralela, los tiempos obtenidos en ambos casos, con una variación del número de hilos desde 1 hasta 16 en potencias de 2, analizando el impacto que éstos tienen en la ejecución de la simulación. Por último, se mostrarán las gráficas que permitan comparar con mayor facilidad las diferencias de tiempos y speed-up que se obtienen con las diferentes configuraciones, además de una gráfica que permita visualizar y comprar los tiempos medios obtenidos con las distintas configuraciones.

4.1 Versión secuencial

A continuación, se muestran los tiempos obtenidos en para la ejecución de forma secuencial del código. Observando los tiempos obtenidos tanto de la versión ***AOS*** como de la versión ***SOA***, ambos son prácticamente idénticos. Esto es debido, a que el tamaño máximo que el problema alcanza en memoria (la estructura de datos de los cuerpos), en el caso de un tamaño de 1000 cuerpos, no llega a producir reemplazos en memoria caché, es decir, no se llega a desbordar la caché. Debido a esto, no se puede observar la diferencia en cuanto a rendimiento entre ambas versiones.

Si se realizase un problema con un número de cuerpos lo suficientemente algo como que se produzca un desbordamiento en la caché, es decir, para que no todos los cuerpos entren en caché, y se produzcan reemplazos en memoria caché, de manera teórica, se podría decir que sería mejor la opción de ***SOA***, puesto que al encontrarse los elementos de forma consecutiva, realizando un acceso secuencial, se aprovecharía la proximidad espacial de éstos, obteniendo un número mayor de aciertos en los accesos.

4.1.1 Variación nbodies

Con lo que respecta a la variación del número de cuerpos, se puede observar, que al aumentar el tamaño de los cuerpos del problema, el tiempo aumenta asemejándose a una función exponencial. Esto se debe, a que cada cuerpo ***i***, interaccionará con los (***n-i)*** restantes, siendo ***n*** el número total de cuerpos e ***i*** la posición del cuerpo en el vector.

De esta forma, el tiempo computacional aumenta de forma exponencial con cada aumento del número de cuerpos, lo cual explica los resultados obtenidos, en donde los tiempos van desde 0,9

segundos en el experimento de menor tamaño con 250 cuerpos hasta 5,5 segundos en el experimento de mayor tamaño con 1000 cuerpos.

Resultados bodies Secuencial				
tipo/bodies	250	500	750	1000
aos	412450698	1441498317	3149149763	5567368937
soa	416349926	1417025675	3138914728	5529542613

Tabla 1: Tiempos nanosegundos bodies secuencial

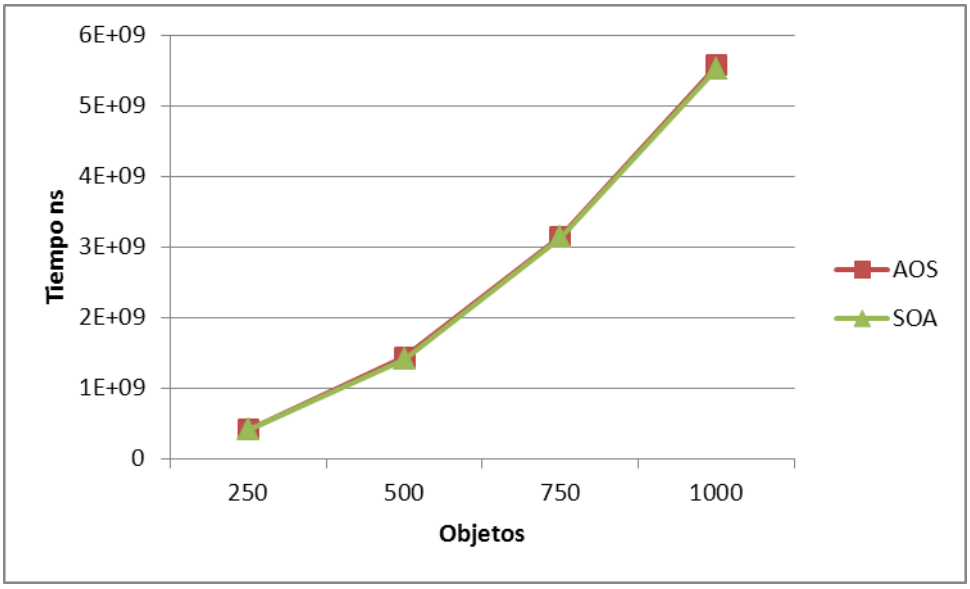


Ilustración 1: Gráfica bodies secuencial

4.1.2 Variación iteraciones

Con lo que respecta a la variación del número de iteraciones, se puede observar, que al aumentar el número de iteraciones, el comportamiento de ambas versiones va a seguir una función lineal. Esto es debido, a que el impacto del número de iteraciones no implica un aumento significativo del cómputo, como si lo es el aumento del número de cuerpos, ya que cada iteración únicamente multiplicar el problema por n, siendo n el aumento en el número de iteraciones.

Resultados iteraciones Secuencial				
tipo/bodies	50	100	150	200
aos	7591091	14702356	25777339	28250076
soa	8260950	14012900	23489301	29186112

Tabla 2: Tiempos nanosegundos iteraciones secuencial

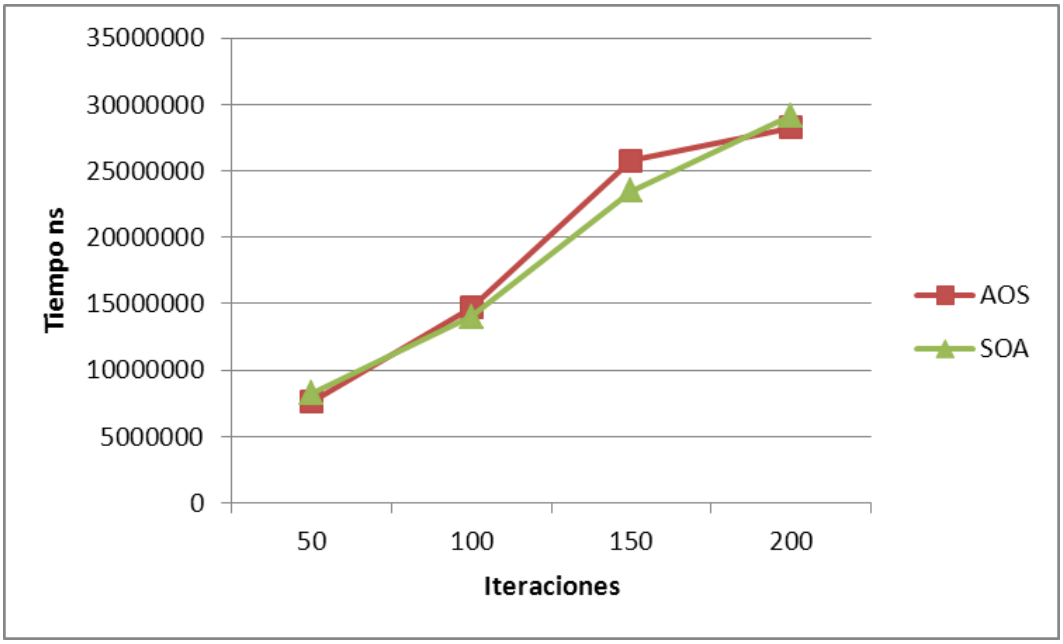


Ilustración 2: Gráfica iteraciones secuencial

4.1.3 Tiempo medio por iteración

Como podemos observar, en función del tamaño de los cuerpos, el tiempo de iteración aumenta de manera considerable, obteniendo una gráfica de tiempos que se asemeja a una función exponencial. Como se ha explicado con anterioridad, esto se debe a que el aumento computacional en cada iteración con el aumento de los cuerpos, provoca éstos aumentos.

Tiempo medio iteración Secuencial				
tipo/bodies	250	500	750	1000
aos	4124507	14414983	31491498	55673689
soa	4163499	14170257	31389147	55295426

Tabla 3: Tiempo medio nanosegundos por iteración secuencial

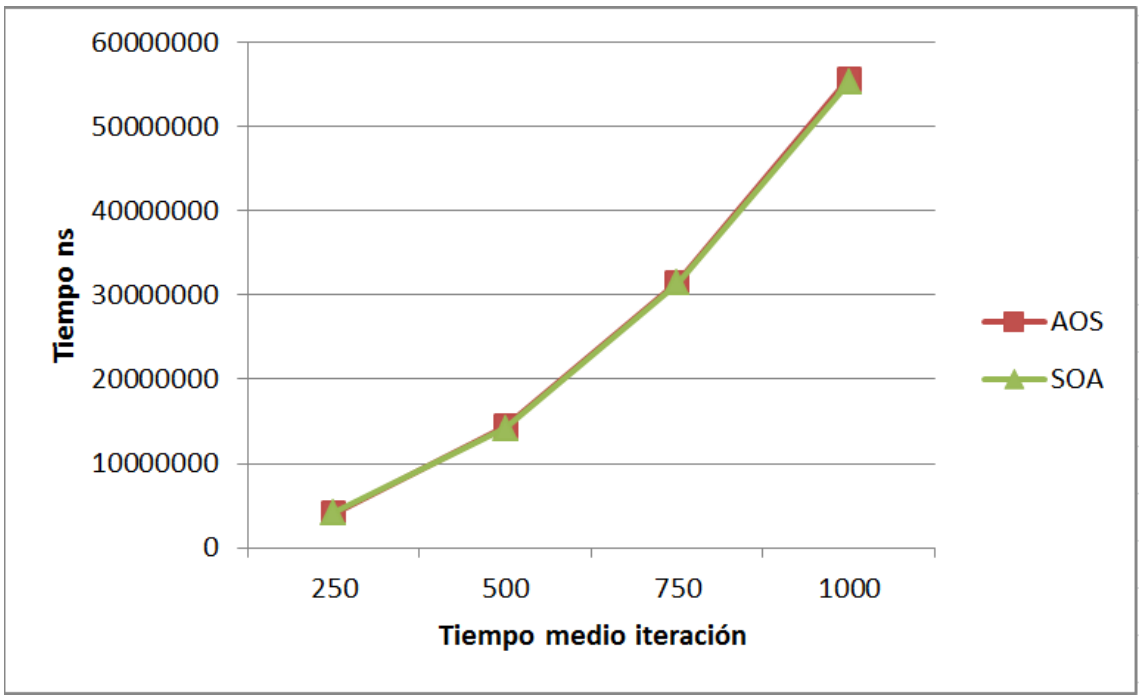


Ilustración 3: Gráfica tiempo medio por iteración secuencial

4.2 Versión paralelizada

A continuación, se muestran los tiempos obtenidos en la versión paralelizada final. Como se puede observar, al igual que en la versión secuencial, los tiempos obtenidos en ambas estructuras de datos son muy parejas, debido a que el tamaño máximo de experimento, en el caso de 1000 cuerpos, no desborda la caché de nivel 3, que es la caché compartida, no provocándose reemplazos y no observando las diferencias en tiempos que produce la organización de los datos entre ***AOS*** y ***SOA***. Por tanto, nos centraremos en explicar cómo afecta a esta implementación la utilización de diferentes números de threads.

Se debe mencionar, que debido a que openMP, cada vez que crea un *pool* de threads, mientras no se cambie el tamaño de ésta mediante la variable de contorno **OMP_NUM_THREADS**, no destruirá los hilos, de forma que no se deben de estar creando y destruyendo constantemente con el consecuente tiempo extra que conllevaría.

Es importante añadir, que si se hace un experimento lo suficientemente grande, el speed-up obtenido sería más significativo, debido a que el tiempo inicial de los hilos es relevante con el tiempo de ejecución tan pequeño debido al tamaño del experimento.

4.2.1 Variación nbodies

A continuación, se muestran los tiempos obtenidos tras la ejecución de los experimentos manteniendo un número fijo de 100 iteraciones y variando el número de cuerpos. Se puede observar, como en todos los casos, pese a aumentar el número de hilos, el comportamiento del aumento de tiempo, es el mismo, siguiendo una función exponencial, al igual que en la versión secuencial. Es por ello, que no se indicarán las gráficas correspondientes a cada tabla en este apartado.

Más adelante, se analizará la diminución del tiempo de ejecución en función del número de hilos, calculando el tiempo medio de iteración con la variación de los cuerpos, donde si podremos observar con mayor claridad que influencia tiene el número de hilos en la ejecución del programa.

Resultados bodies 1 thread				
tipo/bodies	250	500	750	1000
aos	415944575	1439315950	3178106747	5606263337
soa	402569611	1456810042	3186901314	5568525519

Tabla 4: Tiempos bodies 1 thread

Resultados bodies 2 threads				
tipo/bodies	250	500	750	1000
aos	202504300	785045008	1709693623	3061797800
soa	194110509	775311184	1653652992	2942588584

Tabla 5: Tiempos bodies 2 threads

Resultados bodies 4 threads				
tipo/bodies	250	500	750	1000
aos	123182847	448437622	939586622	1696203002
soa	123689098	427968935	916433686	1592032906

Tabla 6: Tiempos bodies 4 threads

Resultados bodies 8 threads				
tipo/bodies	250	500	750	1000
aos	444112421	1078374584	1943421500	3142464114
soa	723940414	1076192938	1940921412	3039753731

Tabla 7: Tiempos bodies 8 threads

Resultados bodies 16 threads				
tipo/bodies	250	500	750	1000
aos	785852574	1808498753	3159956669	4836304053
soa	785535520	1803915698	3140524481	4817488584

Tabla 8: Tiempos bodies 16 threads

4.2.2 Variación iteraciones

A continuación, se muestran los tiempos obtenidos variando el número de iteraciones y manteniendo un número de cuerpos fijo de 50 cuerpos. Se puede observar, al igual que en la versión secuencial, que el aumento del tiempo aumenta de manera lineal con el número de iteraciones. Es interesante ver reflejado en una gráfica como en cada caso, pese a aumentar el número de hilos, el tiempo por iteración se mantiene constante, viéndose como aumenta de manera lineal con el aumento del número de iteraciones. El motivo por el que se muestra en dos gráficas distintas los tiempos obtenidos en ***AOS*** y ***SOA***, se debe a que como se ha explicado anteriormente, al no producirse desbordamiento en la caché compartida de nivel 3, el comportamiento de ***AOS*** y ***SOA*** es prácticamente el mismo.

Se observa como el mayor tiempo lo registran las ejecuciones con 16 hilos, y esto es debido a que al ejecutar sobre una máquina de 4 cores que permiten threads, el rendimiento óptimo se logra con 4 threads. Al asignar un número de threads mayor que 4, los tiempos aumentan de manera considerable debido a que el tiempo de cómputo es menor al de la creación de los hilos y únicamente 4 hilos podrán estar en ejecución simultánea, por lo que no se obtiene un mejor rendimiento al tener que controlar el cambio de hilos en los cores.

Resultados iteraciones 1 thread				
tipo/bodies	**50**	**100**	**150**	**200**
aos	8501022	21088365	30529721	42148986
soa	8381787	18915484	30240440	37950978

Tabla 9: Tiempos iteraciones 1 thread

Resultados iteraciones 2 threads				
tipo/bodies	**50**	**100**	**150**	**200**
aos	6242457	11398623	17114507	22330128
soa	6171041	11399117	16711584	21877143

Tabla 10: Tiempos iteraciones 2 threads

Resultados iteraciones 4 threads				
tipo/bodies	50	100	150	200
aos	5427674	15813496	14884743	19573927
soa	5490465	10068929	19623466	25469189

Tabla 11: Tiempos iteraciones 4 threads

Resultados iteraciones 8 threads				
tipo/bodies	50	100	150	200
aos	39440235	77807986	113233196	151680392
soa	38988087	86528140	113669041	147929314

Tabla 12: Tiempos iteraciones 8 threads

Resultados iteraciones 16 threads				
tipo/bodies	50	100	150	200
aos	72806246	143147511	212837068	282554144
soa	71250627	141543105	211764994	281296458

Tabla 13: Tiempos iteraciones 16 threads

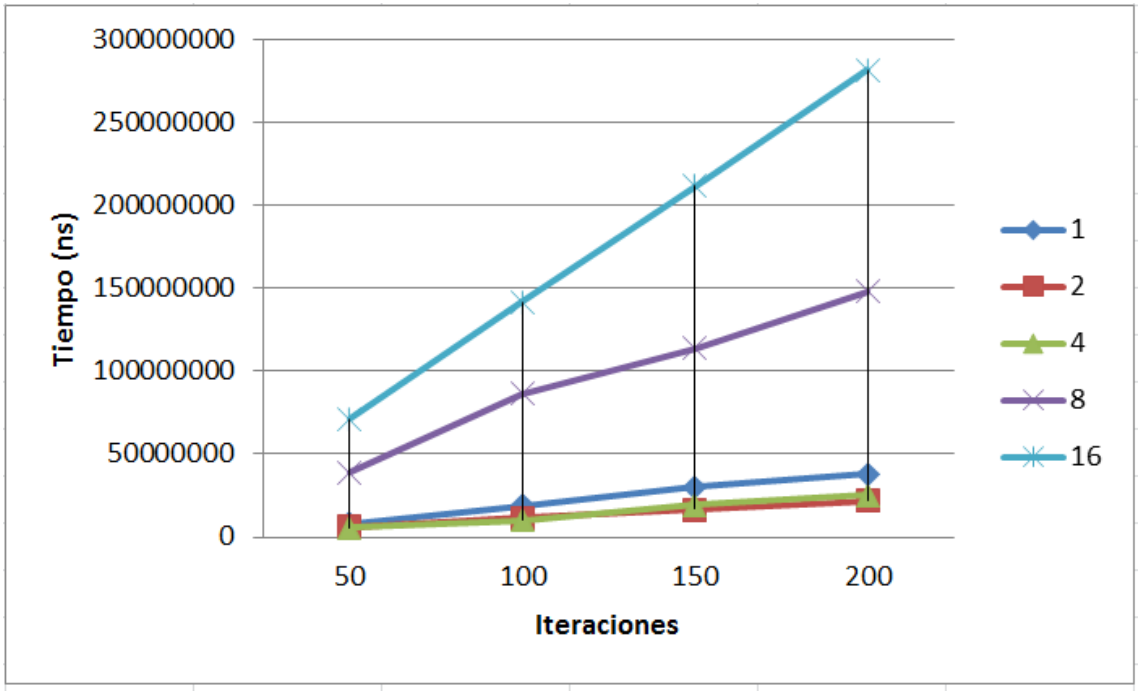


Ilustración 4: Gráfica iteraciones AOS

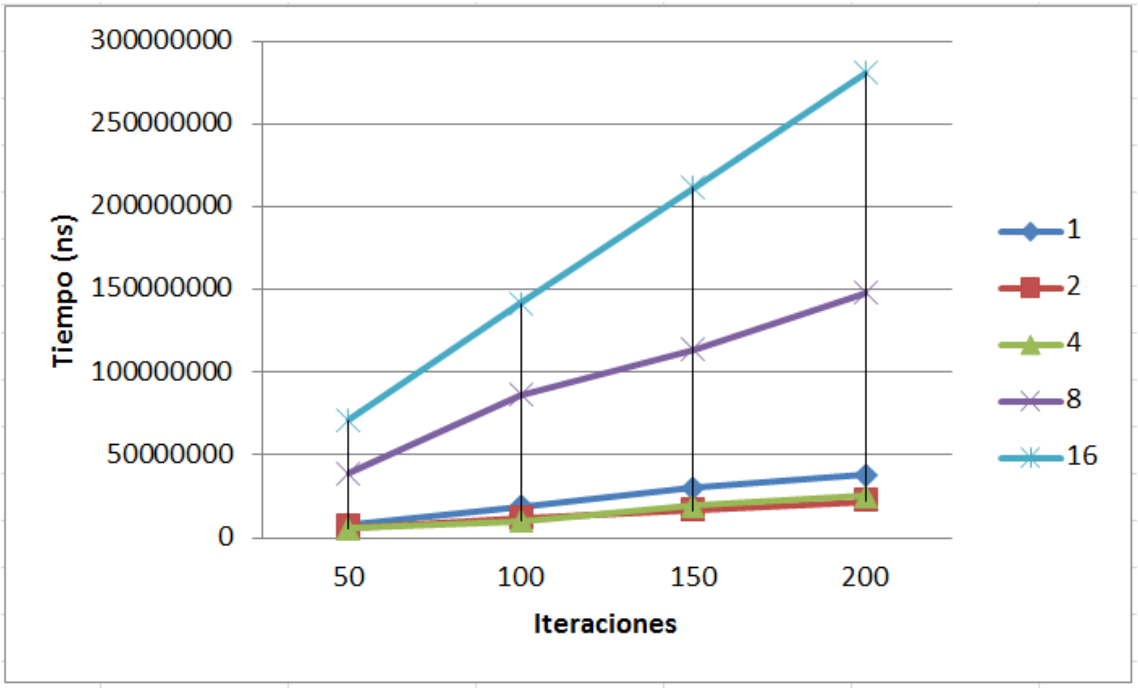


Ilustración 5: Gráfica iteraciones SOA

4.2.3 Tiempo medio por iteración

Una vez obtenido el tiempo medio por iteración, que es (tiempo_ejecución/número_iteraciones) siendo el número de iteraciones igual a 100, podemos analizar el impacto que tiene el número de hilos en la ejecución de la simulación variando el tamaño del problema (el número de cuerpos). Se observa como el peor caso, es con un hilo, debido a que no se produce ningún tipo de paralelismo, pero con un número pequeño de cuerpos, obtiene un tiempo mejor que todas las ejecuciones con un número de hilos mayor que 4. Esto se debe a que al realizar los experimentos en una máquina de 4 cores, la forma óptima de paralelización, se logra con 4 threads, ya que no soporta *hyperthreading*. Por tanto, con una ejecución de cuerpos pequeños, con un thread se obtiene un tiempo menor que con 8 y 16 ya que el tiempo de creación del *pool* de hilos es menor. A medida que se aumenta el tamaño del problema, se obtendrán mejores tiempos con 8 y 16 hilos, debido a que existe paralelismo, pero observando el caso de 16 hilos, no mejora en gran medida al secuencial debido al coste de cambio de hilo en cada núcleo.

Por tanto, podemos concluir que con 2 threads, se mejora en gran medida el tiempo de ejecución, pero la mejor forma de realizar la

paralelización se logra con un número de hilos igual al máximo soportado por el procesador.

Tiempo medio iteración bodies 1 thread				
tipo/bodies	250	500	750	1000
aos	4159446	14393160	31781067	56062633
soa	4159446	14393160	31781067	56062633

Tabla 14: Tiempos bodies 1 thread

Tiempo medio iteración bodies 2 threads				
tipo/bodies	250	500	750	1000
aos	2025043	7850450	17096936	30617978
soa	2025043	7850450	17096936	30617978

Tabla 15: Tiempos bodies 2 threads

Tiempo medio iteración 4 threads				
tipo/bodies	250	500	750	1000
aos	1231828	4484376	9395866	16962030
soa	1231828	4484376	9395866	16962030

Tabla 16: Tiempos bodies 4 threads

Tiempo medio iteración 8 threads				
tipo/bodies	250	500	750	1000
aos	4441124	10783746	19434215	31424641
soa	4441124	10783746	19434215	31424641

Tabla 17: Tiempos bodies 8 threads

Tiempo medio iteración 16 threads				
tipo/bodies	250	500	750	1000
aos	7858526	18084988	31599567	48363041
soa	7858526	18084988	31599567	48363041

Tabla 18: Tiempos bodies 16 threads

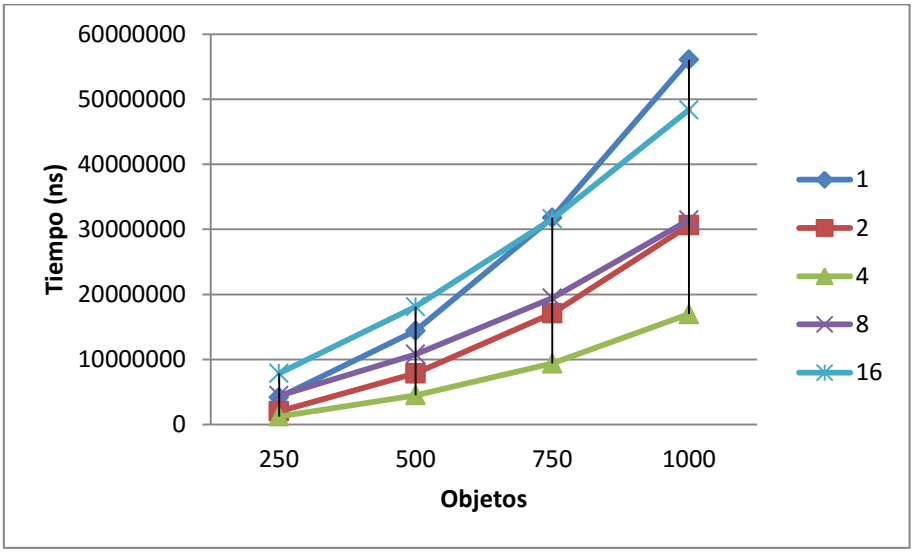


Ilustración 6: Gráfica tiempo medio threads AOS

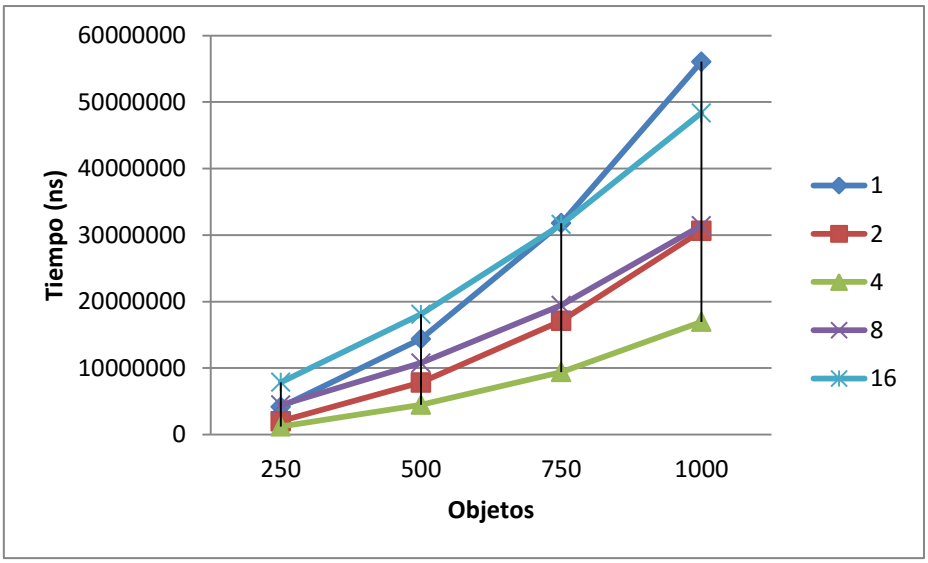


Ilustración 7: Gráfica tiempo medio threads SOA

4.3 Speed-up obtenido

A continuación, se analiza el speed-up obtenido en función de los hilos del tiempo medio de iteración con respecto a la versión secuencial. Se mostrarán los speed-up de las versiones AOS y SOA por separado, debido a que sus comportamientos son muy similares, y es más representativo mostrarlo por separado.

Se puede observar como en ambos casos, el comportamiento es idéntico. Esto se ha mencionado y explicado con anterioridad, y es debido al tamaño del problema y no causar desbordamiento en la caché compartida.

El mejor speed se obtiene con la ejecución en 4 threads. Esto se debe a que es el número de cores que tiene la máquina y el máximo permitido ejecutando simultáneamente en el procesador. Este speed-up es muy cercano a 3.5. Con la ley de amdahl, se sabe que el máximo speed-up en un región de código es igual al tiempo en secuencial entre el número de cores, siendo un speed-up ideal de 4 en nuestro caso. Por tanto, se puede concluir, que pese a no estar muy próximo a 4, se aproxima bastante, y la paralelización es bastante buena.

Con un número de threads igual a 2, se mejora bastante, obteniendo un speed-up de 2. Con 8 threads, se mejora el speed-up conforme el número de cuerpos aumenta, y esto se debe a que si el número de cueropos es pequeño, el coste de lanzar 8 threads no queda compensado con un cómputo muy pequeño.

Con 16 threads, se puede ver claramente como el coste mencionado anteriormente de la creación de los threads, no queda compensado con un número de cuerpos pequeño. El speed-up con 16 hilos por tanto, irá mejorando conforme aumente el tamaño del problema, siendo en el experimento más grande, un poco mayor que uno, lo cual demuestra que no es una buena opción.

En definitiva, la mejor opción a la hora de la ejecución, es elegir un número de threads igual al máximo soportado simultáneamente en ejecución por el procesador, en nuestro caso, 4 threads.

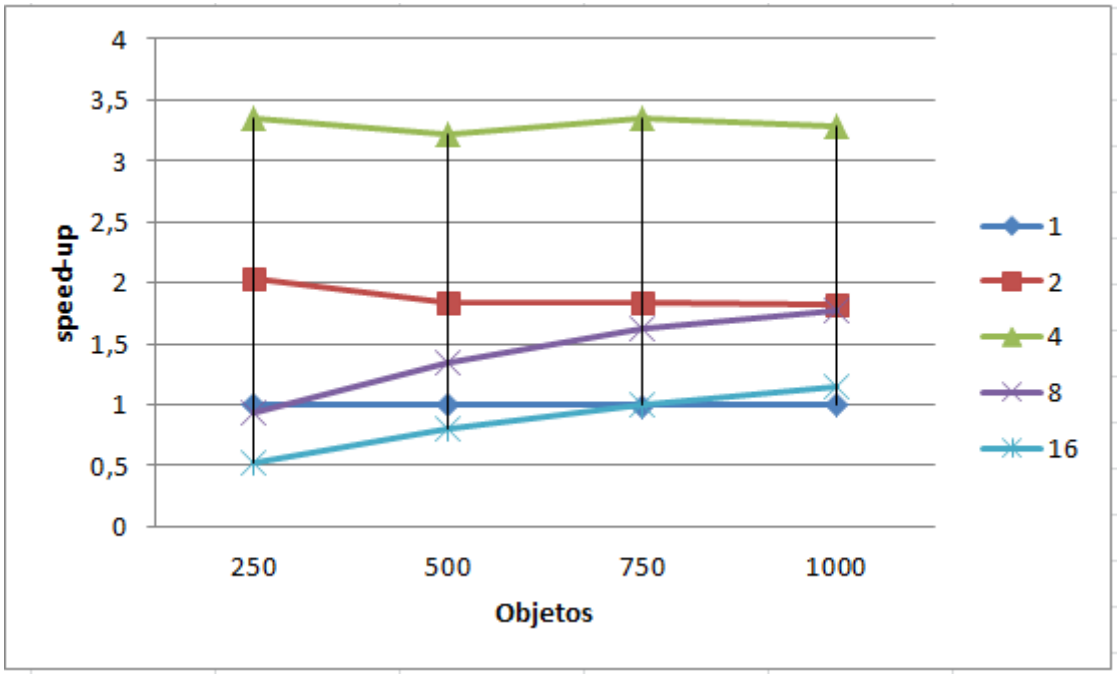


Ilustración 8: Gráfica speed-up AOS

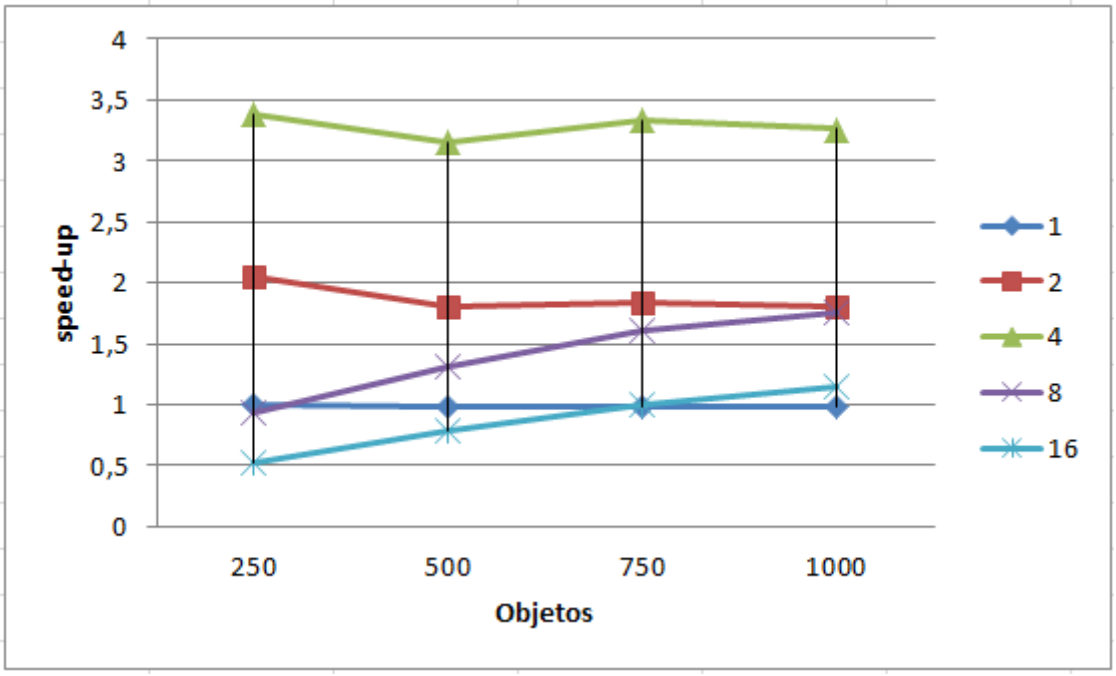


Ilustración 9: Gráfica speed-up SOA

5 Impacto de la planificación

A continuación, vamos a proceder a observar el impacto que produce cada uno de los 3 planificadores requeridos en la ejecución del código. Estos planificadores son static, dynamic y guided. El comportamiento de cada uno, quedará explicado en su correspondiente apartado. Se han realizado las simulaciones con un número de threads igual a 4 y 16. Lo óptimo, sería ver el tamaño de *chunk* a asignar al planificador en cada caso, el cual debe de ser un número entre el número de iteraciones a realizar en el bucle y el número de threads del *pool*.

En este caso, únicamente analizaremos el impacto que tiene el planificador en el tiempo medio de iteración variando el número de cuerpos. Todos los tiempos, están indicados en nanosegundos.

5.1 Planificadores

5.1.1 Static

Este planificador, como su propio nombre indica, reailza una planificación estática de cómo se realizará el reparto de iteraciones a cada hilo. En tamaño del chunk, será asignado por defecto dividiendo en número de iteraciones totales entre el número de hilos. En nuestro caso, se irán asignando chunks de mayor tamaño a menor tamaño, puesto que al paralelizar el bucle interno, el primer chunk dado será de número de cuerpos entre número de hilos. A medida que la ejecución avance y se aumente el iterador del bucle externo, el tamaño del chunk y por tanto, iteraciones asignadas a cada thread, irá disminuyendo. Esta planificación se realiza de manera estática antes de la ejecución del código, es decir, antes de ejecutar el código, ya se sabe que chunk del bucle le toca ejecutar a cada thread.

Tiempo medio iteración static 4 threads				
tipo/bodies	250	500	750	1000
aos	1273195	4389480	9410696	16053278
soa	1217060	4227289	9294552	16282389

Tabla 19: Tiempo medio nanosegundos static bodies 4 threads

Tiempo medio iteración static 16 threads				
tipo/bodies	250	500	750	1000
aos	7825487	18068480	31417608	48288554
soa	7818149	18018445	31334253	48215287

Tabla 20: Tiempo medio nanosegundos static bodies 16 threads

5.1.2 Dynamic

Este planificador realiza una planificación dinámica del reparto de chunks del bucle a cada thread, es decir, no asignará como el static la asignación de cada chunk a cada thread antes de la ejecución, sino que cada thread tomará el chunk disponible una vez acabe el que se encuentra ejecutando. De esta forma, se espera que no existan esperas para la ejecución, de forma que aumente el rendimiento y aprovechamiento de los threads.

Tiempo medio iteración dynamic 4 threads				
tipo/bodies	250	500	750	1000
aos	5654533	21756225	47991179	93025728
soa	5623024	22284893	49167812	85451200

Tabla 17: Tiempo medio nanosegundos dynamic bodies 4 threads

Tiempo medio iteración dynamic 16 threads				
tipo/bodies	250	500	750	1000
aos	11769227	31958919	70272754	115897407
soa	11019773	34046317	70520431	101287187

Tabla 18: Tiempo medio nanosegundos dynamic bodies 16 threads

5.1.3 guided

Este planificador, realizada una planificación dinámica del reparto de chunks, pero con una mejora con respecto al dynamic. Esta mejora tiene que ver con que el tamaño del chunk, se irá modificandoa lo largo de la ejecución para lograr que exista un equilibrio en el reparto de carga de trabajo de cada thread.

Por tanto, se espera que los chunks al comienzo, tengan un tamaño grande, pero que conforme se vaya aproximando al final de las iteraciones, el chunk se vaya haciendo más pequeño, de forma que se aproveche de mejor manera el número de threads.

Tiempo medio iteración guided 4 threads				
tipo/bodies	250	500	750	1000
aos	5654533	21756225	47991179	93025728
soa	5623024	22284893	49167812	85451200

Tabla 21: Tiempo medio nanosegundos guided bodies 4 threads

Tiempo medio iteración guided 16 threads				
tipo/bodies	250	500	750	1000
aos	7858526	18084988	31599567	48363041
soa	11019773	34046317	70520431	101287187

Tabla 22: Tiempo medio nanosegundos guided bodies 16 threads

5.2 Conclusión de la planificación

Una vez realizados los experimentos con los distintos planificadores, vamos a proceder a explicar las distintas conclusiones que se pueden obtener de los resultados. En primer lugar, vamos a proceder a analizar los tiempos de ejecución de cada planificador. Una vez analizados los tiempos obtenidos por cada planificador tanto en ***AOS*** como en ***SOA*** con los distintos números de threads, vamos a analizar el speed-up obtenido planificando el bucle con respecto a la versión secuencial.

5.2.1 Tiempos de planificadores

Podemos observar, como los mejores tiempos obtenidos se dan en la planificación estática con 4 hilos. En primer lugar, hay que destacar que cualquiera de las planificaciones con 16 hilos, darán tiempos altos al igual que sin utilizar el planificador, debido a que la máquina únicamente soporta 4 threads simultáneamente ejecutando en el procesador.

Uno de los motivos por los que los tiempos salen tan altos, se debe a que al tener un tamaño de experimentos tan pequeño, el coste de

planificar el bucle es mayor a la mejora obtenida. Si las simulaciones fueran de un tamaño mayor, podría analizarse mejor el rendimiento y mejora obtenido por los planificadores. Por tanto, puede decirse que la planificación estática es la mejor en este caso puesto que al no realizarse una replanificación en tiempo de ejecución, para nuestro tamaño de problema, es el mejor. Puede observarse como los tiempos obtenidos por dynamic y guided, son muy similares.

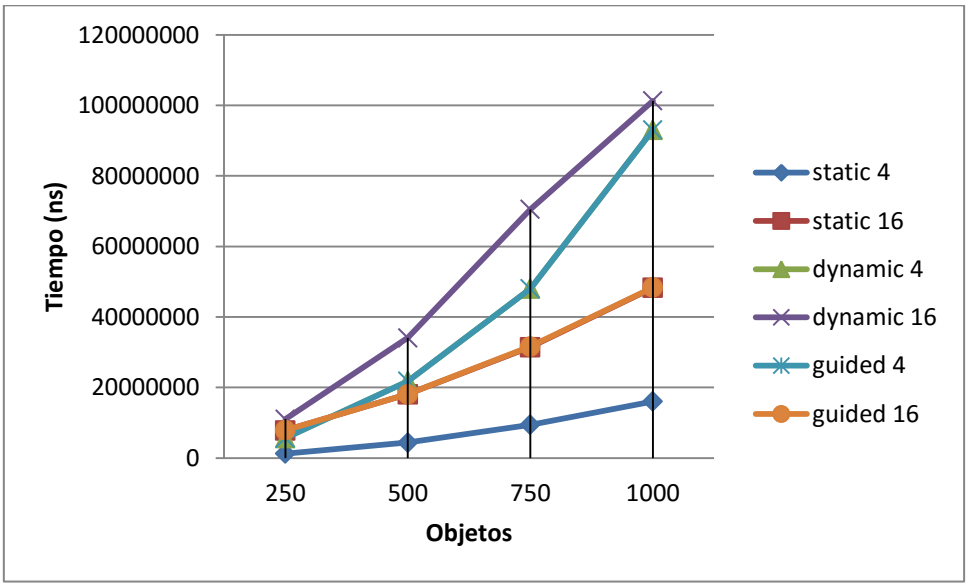


Ilustración 10: Gráfica comparación planificadores AOS

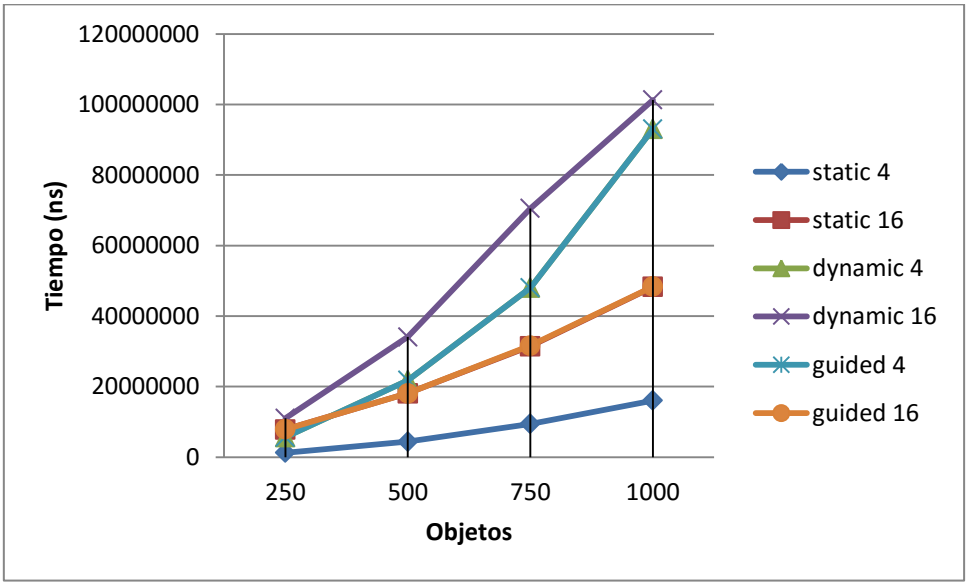


Ilustración 11: Gráfica comparación planificadores SOA

5.2.2 Speed-up de los planificadores

Al analizar el speed-up obtenido con los planificadores, obtenemos varias conclusiones. En primer lugar, podemos observar que no todos los planificadores tienen el mismo impacto en ***AOS*** y ***SOA***. El planificador estático con 4 threads, nos proporciona en ambos casos un speed-up bastante bueno, muy próximo a 3,5. Esto puede deberse a que al realizarse la planificación antes de la ejecución, no se tienen pérdidas de planificación, las cuales en un problema como el nuestro, con un tamaño de cuerpos tan pequeño, tiene un gran impacto.

Ningún planificador en ***AOS*** salvo el static con 4 hilos y el guided con 16, mejoran la versión secuencial. Es destacable que el planificador guided con 16 hilos, sea mejor que el resto de planificadores, lo cual puede deberse al reajuste del chunk, ya que su speed-up va mejorando al aumentar el tamaño del problema. En cambio en ***SOA***, el speed-up de este planificador es bastante malo.

Hay que decir que además de las pérdidas por la planificación dinámica, hay que tener en cuenta el impacto de los fallos en caché causados por el tipo de planificación.

Por tanto, podemos concluir que para nuestra implementación paralelizada del código, la mejor opción sería añadir un planificador static con 4 threads.

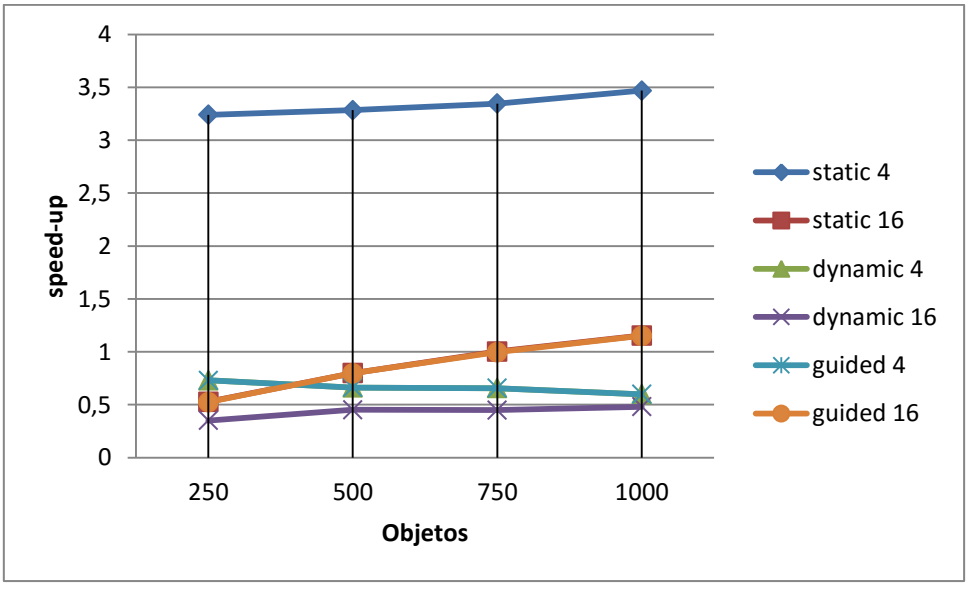


Ilustración 12: Gráfica speed-up planificadores AOS

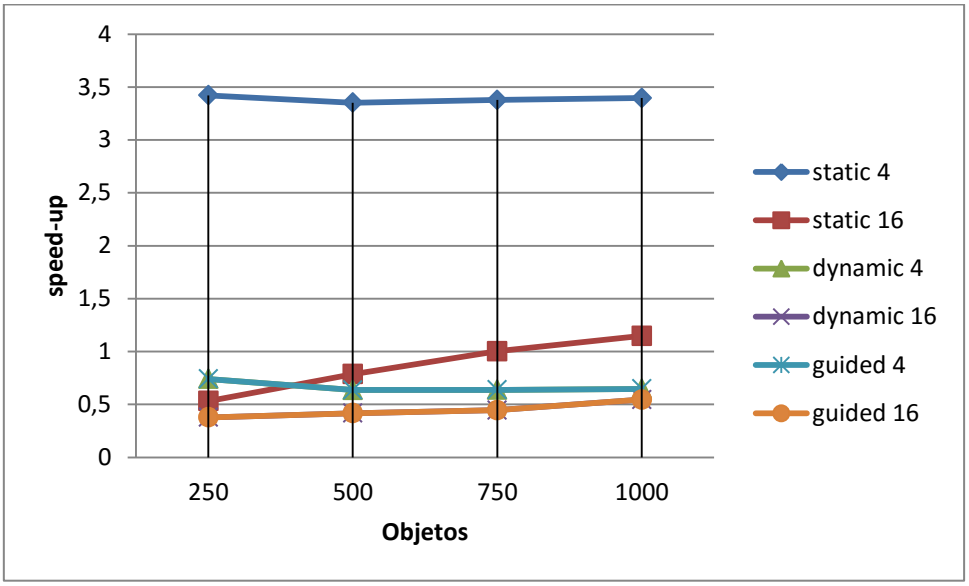


Ilustración 13: Gráfica speed-up planificadores SOA

6 Conclusiones

Una vez finalizada la práctica, han de considerase una serie de aspectos importantes a destacar. Entre estos aspectos, se encuentra la complejidad de analizar el correcto funcionamiento de la implementación, debido a que cada resultado obtenido tras ejecuciones con los mismos parámetros de entrada, difiere con respecto al resto. Esto se debe, a la precisión de las operaciones aritmético lógicas con variables de tipo doublé. La explicación a dicho problema, queda escrita en el artículo especificado en las referencias [2].

Otro aspecto importante a tener en cuenta, es el tamaño máximo de los experimentos. Debido a que la ejecución de las pruebas ha sido en una máquina con un tamaño de memoria caché de 6 MB, y que las pruebas realizadas son de 1000 objetos con 5 variables de tipo double cada uno, en ningún momento se realiza un desbordamiento en la caché compartida que produzca reemplazos en esta caché. Por tanto, realizar un análisis entre AOS y SOA, resulta imposible con este tipo de experimentos. Si se realizasen pruebas de mayor tamaño, se verían como afecta el prefetching que realiza la memoria caché a la hora de acceder a los datos cada vez que se produce un fallo, de forma que ahí si se podría observar el rendimiento de cada estructura de datos. Pese a que el enunciado no lo pide y tras realizar pruebas, hemos podido observar que el tipo SOA, pese a tener los elementos separados entre cuerpos, una vez se deben de realizar

reemplazos, produce una menor tasa de fallos en cada acceso, causa posiblemente ocasionada por el prefetching.

También se debe de mencionar, que no se puede realizar una comparación de speed-up entre AOS y SOA de manera real, además de por el tamaño de los experimentos, porque el código empleado para una estructura de datos y otra es diferente, lo cual no realiza una comparativa real de los experimientos.

Por tanto, sería interesante poder observar el comportamiento de los dos tipos de estructuras de datos con unos tamaños de cuerpos considerablemente grandes, y pudiendo analizar los casos para los que uno y otro tipo es mejor, y analizar las causas que hacen que cada tipo de estructura obtenga mejor rendimiento en cada caso.

En definitiva, la práctica sirve para comparar el speed-up y rendimiento en función del número de hilos y el uso de openMP, pero no se puede obtener una idea real de la diferencia de uso entre AOS y SOA, lo cual es el principal objetivo de misma.