uc3m | Universidad Carlos III de Madrid

Ingeniería Informática (2019/2020) Grupo 81

Arquitectura de computadores
Grupo ARCOS
Departamento de informática

Práctica de programación paralela con OpenMP

Alejandro Parrado Arribas. NIA: 100383453
Andrés Vinagre Blanco. NIA: 100383414
Laura Sánchez Cerro. NIA: 100383419
Alicia Romera Gómez. NIA: 100383563

ÍNDICE

1. Introducción

En esta práctica realizamos el estudio de dos versiones de código, una secuencia y otra paralela que irá variando en las pruebas para evaluar la optimización de resultados.

Ambas versiones están realizadas en C++ y se denominan nasteroid-seq y nasteroid-par. Para las dos hemos creado una estructura llamada "struct object" que hace referencia a los asteroides y a los planetas conjuntamente. Esta estructura tiene como atributos las posiciones, las velocidades y las fuerzas para ambos ejes y la masa. Los asteroides tendrán todos los atributos rellenos, pero los planetas tendrán los atributos de las velocidades y las fuerzas a 0. Esta estructura se invoca desde las clases principales nasteroid-seq y nasteroid-par.

A lo largo de la memoria, se explica el funcionamiento del código implementado y las pruebas realizadas para su comparación cambiando el número de objetos, iteraciones, hilos y planificación.

2. Versión secuencial.

2.1 Implementación y optimización del código

Para empezar, hemos incluido las librerías necesarias para nuestro código, la estructura object y las constantes requeridas para el funcionamiento de la práctica.

Dentro del main, validamos que el número de argumentos sean los necesarios (4). En caso contrario, se mostrará un mensaje de error. Pasamos los argumentos introducidos de string a float y validamos que sean enteros positivos preguntando si el argumento menos la parte entera del argumento es igual a cero. Si es así, hacemos un casting para convertirlos a enteros. Estos argumentos los hemos denominado num_asteroids, iterations, num_planets y seed.

Para guardar los objetos, creamos un único vector de tipo object con tamaño igual a la suma de asteroides y planetas. Las primeras posiciones del vector corresponden a los asteroides y las siguientes a los planetas.

Generamos las distribuciones para obtener las coordenadas aleatorias y la masa para los objetos y creamos el fichero "init_conf" para guardar los argumentos y los valores iniciales.

Recorremos con un for el vector de objects para inicializar los atributos de los asteroides y los planetas. Por un lado, los asteroides deben tener una posición y masa aleatoria que generamos con el código proporcionado en el enunciado poniendo su velocidad y fuerza a cero.

Por otro lado, los planetas sólo tienen la posición y la masa. Cada vez que se crea un planeta, se coloca en un eje empezando por la izquierda (eje x=0, y=aleatoria). Para avanzar al siguiente eje, se hace con el movimiento de las agujas del reloj. En el eje y=0, la posición x será aleatoria. El siguiente eje es el que tiene "x" igual a la anchura del grid y la posición "y" aleatoria. El último eje tendrá la posición "x" aleatoria y la posición "y" será igual a la altura del grid.

Cuando coloca los cuatro ejes, vuelve a empezar. Para ello, utilizamos el módulo 4 y vamos incrementando la variable "j". Así, cada planeta se irá colocando en el siguiente eje.

La masa de los planetas será igual que la de los asteroides, pero multiplicada por 10.

Por último, fijamos la precisión de todos estos valores a 3 decimales haciendo uso de "fixed" y setprecision(3) y guardamos los valores de los objetos creados en el fichero "file_init". Después de colocar todos los asteroides y planetas con los bucles correspondientes, cerramos el fichero.

Creamos un bucle for para las iteraciones introducidas como argumento. Dentro de este bucle, recorremos otros dos bucles anidados para calcular la interacción de un asteroide con un objeto (es decir, con otro asteroide o un planeta). Por ello, el primer bucle (j) recorrerá los asteroides y el segundo (k) el objeto con el que interacciona. Como este último no puede ser el mismo asteroide, el bucle empieza en k=j+1. Los cálculos pedidos son la distancia y, cuando esta sea mayor de 5, la pendiente, el ángulo y las fuerzas (componente fx y

fy). Si la fuerza es mayor de 100 se trunca a este valor. Antes de hacer la acumulación de las fuerzas, la guardamos en una variable porque debemos guardar esa fuerza que se produce con el otro asteroide con el que interacciona, pero con signo negativo. Tenemos en cuenta que sólo se acumulará esta fuerza negativa si el otro objeto es un asteroide y, por ello, preguntamos que el índice k sea menor que la variable num_asteroids ya que en el mismo vector se guardan tanto asteroides como planetas.

Después, se procede a calcular la aceleración, la velocidad y la posición en los dos ejes únicamente de los asteroides y se reinicializan sus fuerzas a 0 para dejarlo preparado para la siguiente iteración.

Lo siguiente que hacemos es calcular las nuevas posiciones de los asteroides si rebotan con los bordes y le cambiamos el signo a la velocidad.

Para ello, calculamos el rebote entre los asteroides; se nos pedía en el enunciado que podían rebotar más de 2 asteroides a la vez y que al rebotar varios se debían intercambiar las velocidades con el siguiente asteroide según el orden del vector. Lo primero que hacemos es guardar en unas variables auxiliares las velocidades del **primer** asteroide que está a una distancia menor de 5 unidades a otro asteroide, después de haber guardado sus antiguas velocidades ya las podemos reemplazar por las nuevas velocidades del asteroide cercano y también guardamos en una variable auxiliar un identificador del **último** asteroide que habría que modificar por estar a una distancia menor de 5. Tras el reemplazo, pasaríamos al siguiente asteroide del vector de modo que, si existiera algún asteroide a una distancia menor que 5, habría que cambiar las velocidades de un vector y actualizar la variable auxiliar de "último asteroide". Por último; cuando ya hemos comprobado todas las distancias entre los asteroides, debemos otorgar al **último** asteroide las velocidades del **primer** gracias a las variables auxiliares.

Por último, guardamos las nuevas posiciones, velocidades y masa de los asteroides con precisión fijada a 3 decimales en el fichero denominado “out.txt”.

Para el cálculo de tiempo hacemos uso de “chrono” para medir los tiempos desde el inicio del código hasta el final. Ponemos un timer al principio y otro al final y calculamos la diferencia que se imprime como salida del programa.

2.2 Justificación del diseño

Los asteroides y los planetas los guardamos en el mismo vector debido a que muchos de los cálculos que debemos hacer para la interacción son los mismos. Por ello, para no repetir código, hemos decidido guardarlos en el mismo vector e ir diferenciándolos con las variables num_asteroids y num_planets.

Por lo tanto, como hemos dicho en la introducción, creamos una única estructura global para los asteroides y los planetas que llamamos “struct object” debido a que comparten muchos de los atributos y, aquellos que los planetas no tienen, se ponen a 0.

En un principio dividimos el código en funciones para el cálculo de distancia, pendiente, ángulo, fuerzas, rebotes, etc. Sin embargo, al ser funciones con muy poco contenido de código, decidimos eliminarlas y escribir directamente en el main para evitar que se ralentizara demasiado el programa teniendo en cuentas el tiempo de llamada y retorno de una función.

3. Versión paralela.

3.1 Implementación y optimización del código.

Para la versión paralela de la práctica hemos implementado en el código de la parte secuencial los pragmas de la librería Open MP. Como el diseño que hemos empleado para la parte paralela es el mismo que la secuencial, en este apartado solo nos centraremos en los pragmas de Open MP que hemos introducido en el código y luego, tras realizar la evaluación y las pruebas para distintos números de hilos y planificación, elegiremos la mejor versión paralela.

El primer pragma utilizado ha sido un “#pragma omp parallel for”, en el que el segmento de código que paralelizamos es el cálculo de las fuerzas. De forma que las iteraciones de este bucle serán divididas entre los hilos. El segundo pragma utilizado para paralelizar es “pragma omp parallel for” y el bloque de código que

hemos paralelizado es el del rebote de asteroides contra el grid. No obstante, el movimiento de asteroides lo intentamos paralelizar, pero no nos daba los mismos resultados ni empleando un "for reduction". Por lo que la decisión final ha sido paralelizar las fuerzas y el rebote de los asteroides contra el grid.

A lo largo de los siguientes apartados se muestra el cálculo del tiempo medio con diferente número de hilos y planificaciones, el tiempo medio por iteración y el speedup. Con esto concluimos cuál es la mejor versión paralela de entre todas las probadas y realizadas.

3.2 Justificación del diseño.

Como hemos comentado anteriormente, hemos decidido mantener el mismo diseño de la versión secuencial para la versión paralela. Por lo que la diferencia solo está en la introducción de bloques paralelos.
Se probó a hacer una versión con matrices de fuerzas para evitar que en el bucle de iteraciones hubiera una variable acumulativa de fuerzas y pudiera haber algún problema. Sin embargo, aunque los resultados de la versión secuencial y paralela eran los mismos, el tiempo era mayor por lo que optamos por paralelizar de la manera explicada.

4. Evaluación del rendimiento.

A continuación, procederemos a analizar los resultados. Hemos evaluado con 250, 500 y 1000 objetos (la mitad asteroides y la otra mitad planetas) y con 50, 100 y 200 iteraciones. Hemos repetido las pruebas 10 veces para estas combinaciones y calculado el tiempo medio total. Por otra parte, hemos calculado el tiempo medio por iteración: *Tiempo medio por iteración* = $\frac{Tiempo\ medio\ total}{número\ de\ iteraciones}$

Para el siguiente análisis, mostramos en el gráfico de la izquierda el tiempo medio total para cada prueba en función del número de objetos e iteraciones. En el gráfico de la derecha, se muestra el tiempo medio por iteración obtenido.

En la tabla de datos aparece también el speedup calculado respecto a la ejecución secuencial:

$$Speedup = \frac{Tiempo\ medio\ total\ SECUENCIAL}{Tiempo\ medio\ total\ PARALELIZACIÓN}$$

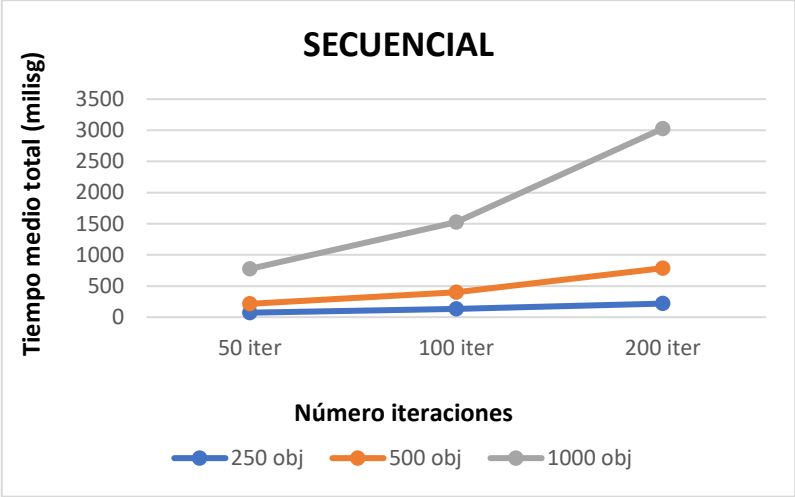
Por último, mostramos una comparativa entre las distintas paralelizaciones y una comparativa de los distintos speedup para 250 objetos y 1000 objetos.

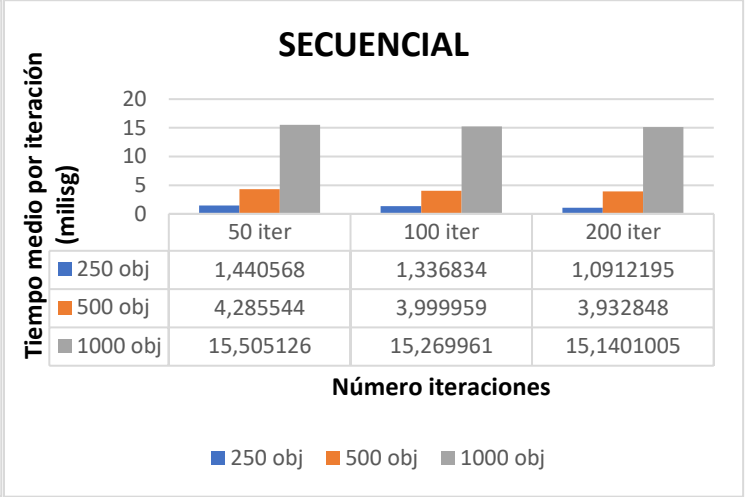
Todas estas pruebas las hemos hecho en un ordenador con las siguientes características:

- Procesador: Kabylake i5-7300HQ (4 núcleos, 4 hilos, 2.5GHz-3.5GHz, 6MB L3 Cache)
- Tecnología Hyper-Threading Intel®: No
- Memoria 8GB DDR4-2400 (1 módulo)
- Disco duro 1TB
- Controlador gráfico: GeForce® GTX 1050, 4GB GDDR5
- S.O.: Ubuntu 18.04.3 LTS (64 bits) GNOME 3.28.2
- Compiler: g++ (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0

4.1 Secuencial

Promedio del tiempo total	50 iteraciones	100 iteraciones	200 iteraciones
250	72,0284	133,6834	218,2439
500	214,2772	399,9959	786,5696
1000	775,2563	1526,9961	3028,0201





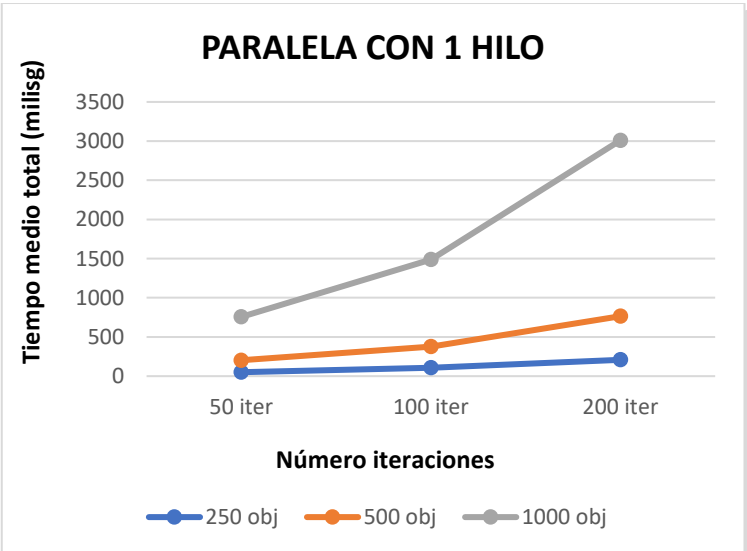
En la gráfica de la izquierda donde se muestra el tiempo total, se aprecia como el incremento del tiempo crece en función del aumento de objetos e iteraciones. Para el mismo número de objetos, si se aumentan las iteraciones, el tiempo total aumenta. Esto se debe, obviamente, a que los bucles se van a ejecutar más veces. Pero el aumento es mucho mayor cuando se incrementa el número de objetos, es decir, lo que más influye es este parámetro. De manera que la mayor pendiente se produce en la recta gris con 1000 objetos y se dispara, sobre todo, al pasar de 500 a 1000 iteraciones.

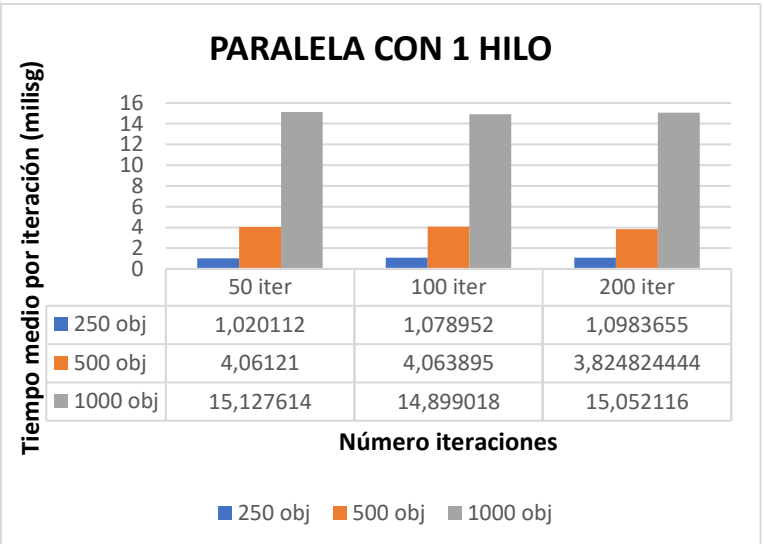
En la gráfica de la derecha se puede observar que para el mismo número de objetos el tiempo medio por iteración permanece invariable para los casos de 50, 100 o 200 iteraciones. Esto tiene sentido puesto que el tiempo no depende del número de iteraciones, sino del número de objetos que se manipulan en cada iteración. Se observa que, a medida que se incrementa el número de objetos, aumenta bastante el tiempo por iteración.

4.2.-PARALELA CON 1 HILO

En esta tabla mostramos los resultados obtenidos tras la paralelización con 1 hilo y su speedup.

Promedio del tiempo total	50 iter	100 iter	200 iter	Speedup 50 iter	Speedup 100 iter	Speedup 200 iter
250	50,2952222	107,8952	208,214333	1,432112174	1,239011559	1,048169436
500	203,0605	378,425222	764,964889	1,055238217	1,057001164	1,028242749
1000	756,3807	1489,9018	3010,4232	1,024955158	1,024897144	1,005845324

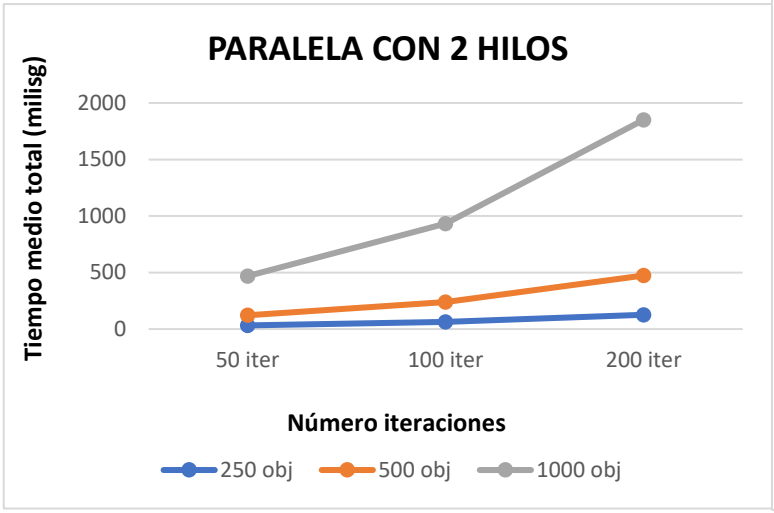


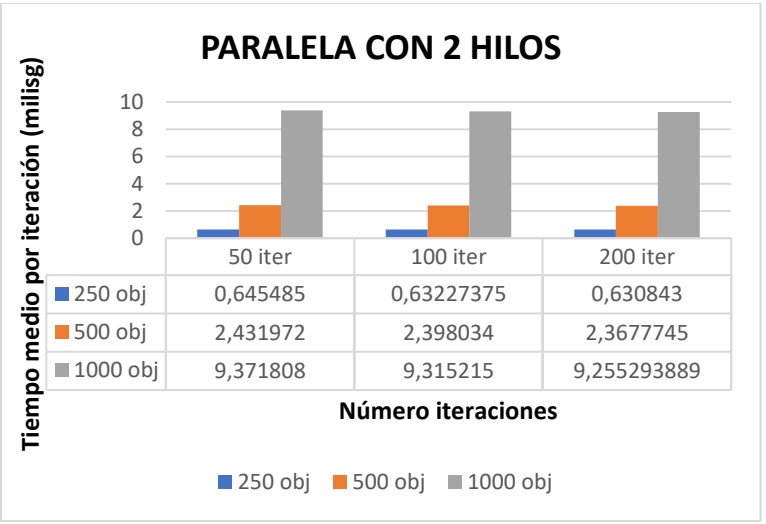


Las conclusiones respecto a estas gráficas son similares a las anteriores. Lo que cabe destacar es que se reduce el tiempo muy poco y que, por tanto, el speedup es bastante bajo ya que no se aprecia una mejoría destacable ni en cuanto al tiempo total medio ni en el tiempo medio por iteración. Esto es normal, porque la ejecución con un hilo es bastante similar a la secuencial y los resultados son los esperados.

4.3.-PARALELA CON 2 HILOS

Promedio del tiempo total	50 iter	100 iter	200 iter	Speedup 50 iter	Speedup 100 iter	Speedup 200 iter
250	32,27425	63,227375	126,1686	2,231760614	2,114327852	1,729779834
500	121,5986	239,8034	473,5549	1,762168314	1,668015966	1,660989254
1000	468,5904	931,5215	1851,05878	1,654443412	1,639249443	1,635831413

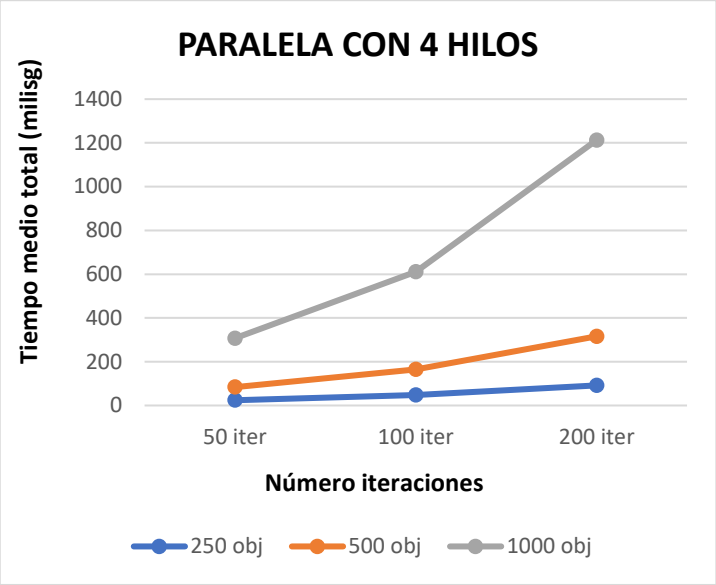


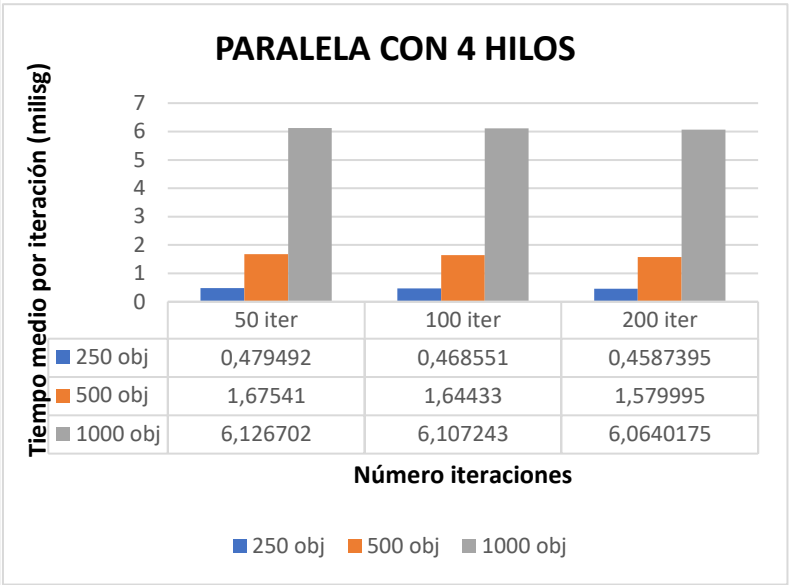


En este caso, a diferencia del anterior con un hilo, se ve que la ejecución con dos hilos mejora el rendimiento sobre todo al aumentar el número de objetos e iteraciones ya que disminuye el tiempo total y el tiempo medio por iteración se reduce significativamente: para el caso mayor (1000 obj y 200 iter) pasa de 15 a 9 milisegundos (40% de mejora). Esto se debe a que se paralelizan los cálculos respecto de la secuencial.

4.4.-PARALELA CON 4 HILOS

Promedio del tiempo total	50 iter	100 iter	200 iter	Speedup 50 iter	Speedup 100 iter	Speedup 200 iter
250	23,9415	47,13025	91,7479	3,008516593	2,836467025	2,378734554
500	83,7705	164,433	315,999	2,557907617	2,432576794	2,489152181
1000	307,479333	610,7243	1212,8035	2,521328157	2,500303492	2,496711215

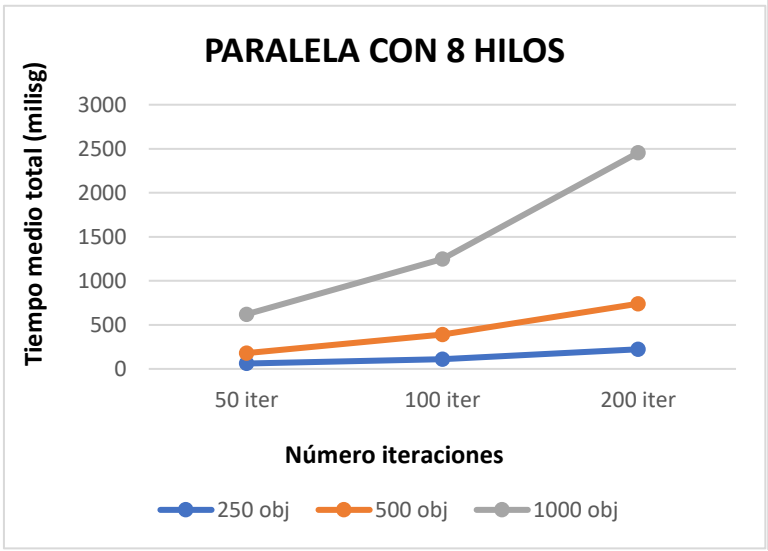


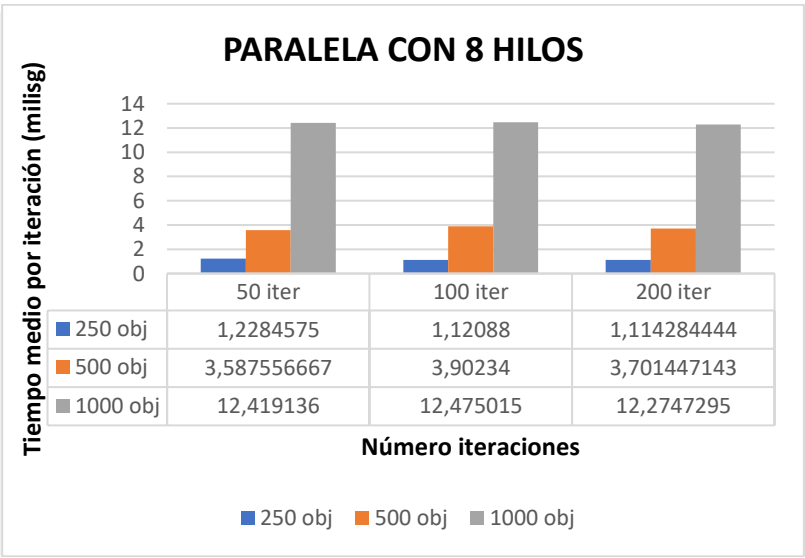


Siguiendo con la misma dinámica, el caso de 4 hilos reduce todavía más que la de 2 hilos los tiempos obteniéndose un speedup mucho mayor que en la secuencial (entre 2,5 y 3). Como en el caso anterior, para el caso más problemático, el tiempo medio por iteración se reduce de 15 a 6 milisegundos, es decir un 60% de mejora. Esto es debido a que, con 4 hilos, la CPU del equipo con el que hemos realizado las pruebas trabaja al 100%.

4.5.- PARALELA CON 8 HILOS

Promedio del tiempo total	50 iter	100 iter	200 iter	Speedup 50 iter	Speedup 100 iter	Speedup 200 iter
250	61,422875	112,088	222,856889	1,172664093	1,192664692	0,9793006668
500	179,377833	390,234	740,289429	1,194557858	1,025015504	1,06251632
1000	620,9568	1247,5015	2454,9459	1,24848669	1,224043498	1,23343659

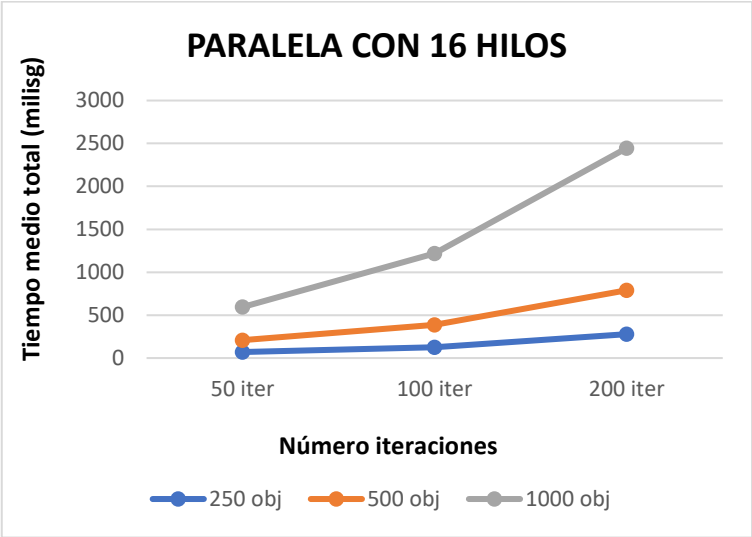


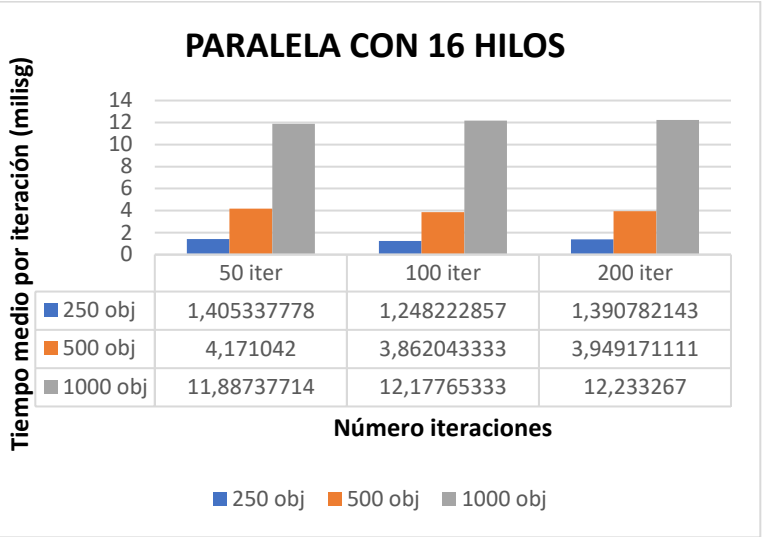


En este caso, sin embargo, no se mejora, sino que se reduce el rendimiento respecto a la de 4 hilos. Esto es debido a que, con 8 hilos, la CPU del equipo con el que hemos hecho las pruebas solo trabaja un 75% por núcleo, por lo que el resultado es peor que con la anterior.

4.6.- PARALELA CON 16 HILOS

Promedio del tiempo total	50 iter	100 iter	200 iter	Speedup 50 iter	Speedup 100 iter	Speedup 200 iter
250	70,2668889	124,822286	278,156429	1,025068864	1,070989839	0,7846085053
500	208,5521	386,204333	789,834222	1,027451654	1,035710544	0,9958666997
1000	594,368857	1217,76533	2446,6534	1,304335331	1,253932972	1,237617106



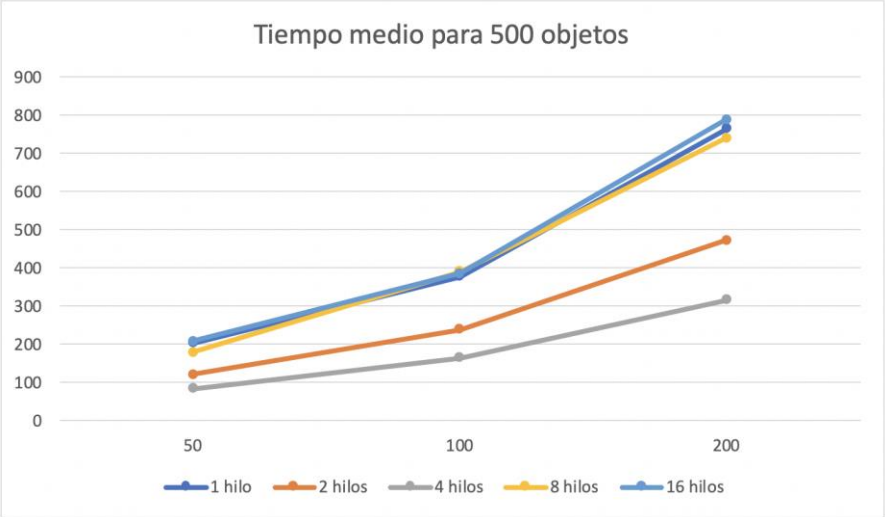


Como en el caso anterior, el rendimiento no mejora respecto a 4 hilos ni a la de 8 hilos siquiera, sino que empeora. Esto es debido a que el procesador ya no trabaja al 100%, sino al 75% por núcleo.

A continuación, hacemos una comparativa de las paralelizaciones anteriores para verlos de manera conjunta en una única gráfica. Las gráficas se han realizado según el número de objetos (250, 500 y 1000).

4.7. Tiempo medio total para los casos de 1 hilo, 2 hilos, 4 hilos, 8 hilos y 16 hilos.

En esta gráfica se representa el tiempo medio de ejecución del programa con los diferentes números de hilos para 250 objetos. Analizando esta gráfica, vemos que la versión que emplea menos tiempo, como hemos dicho, es aquella en la que se emplean 4 hilos. En las siguientes pruebas vemos cómo irá escalando a medida que aumente el número de objetos y se verá qué número de hilos es el que hace el menor tiempo de ejecución.



En esta gráfica se observa que, aumentando el número de objetos a 500, se obtiene la misma conclusión y es que la versión paralela de 4 hilos es la que mejor escala.

Con 1000 objetos, de nuevo, se obtiene que la mejor versión paralela es aquella en la que se emplean 4 hilos. Como conclusión final de estas tres pruebas hay que señalar que el caso con 1 hilo es el que consume más tiempo de ejecución con 1000 objetos ya que no tiene mucho sentido paralelizar un programa y emplear un solo hilo. Los casos de 8 y 16 hilos tardan más tiempo en ejecutarse que 1 hilo cuando se emplean 250 objetos, pero a medida que se incrementa el número de objetos escalan mejor que la versión de un hilo y con 1000 objetos el tiempo de ejecución de ambos es mejor.

En todos los casos probados aquí, el que consume menor tiempo es el de 4 hilos, como hemos dicho, ya que el ordenador tiene cuatro procesadores y con cuatro hilos cada hilo aprovecha todo el rendimiento de cada núcleo. Por ello mismo creemos que en el caso de 16 hilos, en el que se ejecutan 4 hilos en cada núcleo, tarda más en ejecutar, ya que la gestión de los 4 hilos en cada núcleo es peor que gestionar 1 hilo en cada núcleo. El caso de 2 hilos con 250, 500 y 1000 objetos es la segunda mejor paralelización (después del caso de 4 hilos), no obstante, los núcleos no se pueden aprovechar al máximo aquí ya que con dos hilos se ejecuta un hilo en cada núcleo. De forma que no se puede obtener un rendimiento máximo del procesador.

Por tanto, la mejor opción para este programa es emplear paralelización con 4 hilos. Obteniéndose un tiempo de ejecución de 1200ms.

4.8 Tiempo medio de planificación 4 y 8 hilos.

En las gráficas que se muestran a continuación se analizan los casos de 4 y 8 hilos empleando planificación ("#pragma omp parallel for schedule (tipo_planificación)" en los dos pragmas for que se han implementado en la versión paralela.

Se emplean tres tipos de planificación: *static, dynamic y guided.*

En estas pruebas empleamos 250 objetos y 1000 objetos puesto que la mejor versión paralela será aquella que escale mejor del caso inicial al caso en el que se tiene un mayor número de objetos. Sin embargo, en las tablas siguientes mostramos los datos obtenidos de tiempo medio total y speedup para todos los casos.

4 hilos STAT	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	27,6325	46,2466667	84,8298889	2,606655207	2,890660226	2,572724105
500	81,9398889	154,332	304,934	2,615053582	2,591788482	2,579474903
1000	290,9251	564,3157	1113,1701	2,664796884	2,705925247	2,720177357

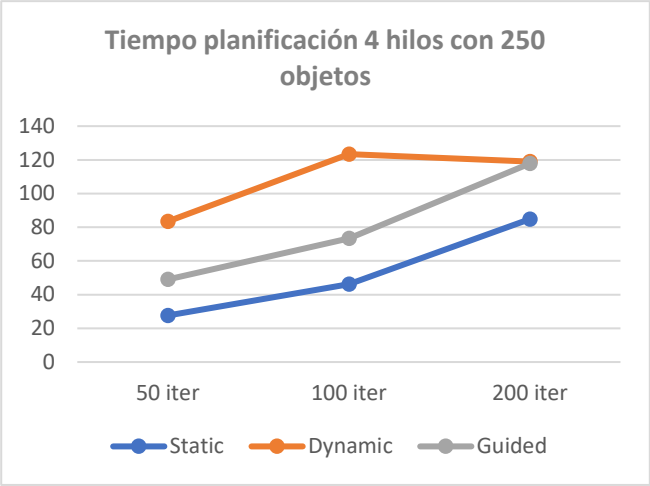
8 hilos STAT	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	27,2168	53,4239	144,3705	2,646468358	2,502314507	1,511693178
500	177,6841	375,9395	757,2789	1,205944707	1,063990084	1,03867888
1000	589,6067	1145,6558	2315,2484	1,314870235	1,332857652	1,30785971

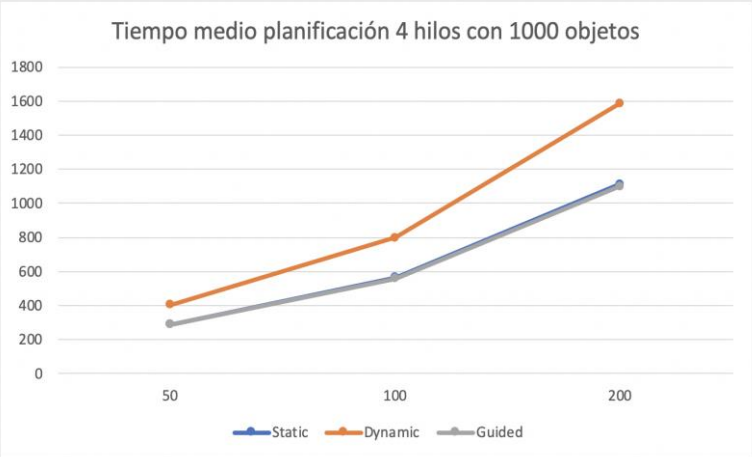
4 hilos DYN	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	83,4531	123,3833	118,8944	0,8631003522	1,083480503	1,835611265
500	112,3732	217,1762	419,5004	1,906835438	1,841803568	1,875015137
1000	405,7583	800,3076	1587,2612	1,910635716	1,908011495	1,907701203

8 DYN	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	32,3557	63,1029	143,3648	2,226142534	2,118498516	1,522297663
500	125,5385	244,7788	485,4246	1,706864428	1,634111696	1,62037441
1000	431,9948	809,5393	1603,0231	1,79459637	1,886253206	1,888943522

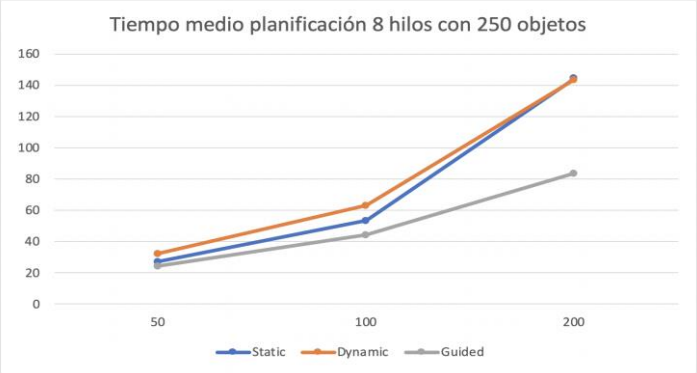
4 hilos GUIDED	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	49,0241	73,4246	117,8501	1,469244718	1,820689524	1,851877088
500	101,5637	157,4909	302,2242	2,109781349	2,539803252	2,602602968
1000	290,5748	557,8237	1100,7817	2,668009408	2,737417037	2,750790734

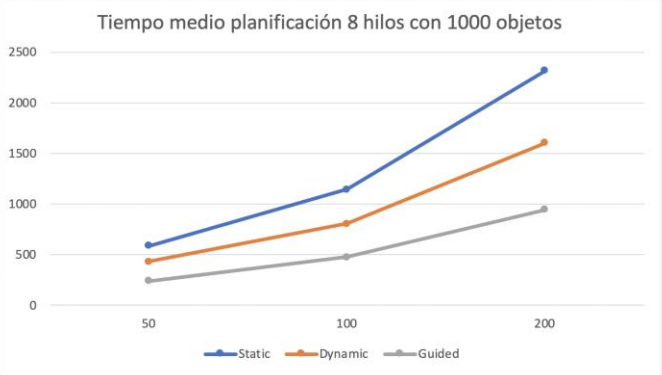
8 hilos GUIDED	**50 iter**	**100 iter**	**200 iter**	**Speedup 50 iter**	**Speedup 100 iter**	**Speedup 200 iter**
250	24,4138	44,164	83,6753	2,950314986	3,026976723	2,608223693
500	70,0591	133,5695	257,2689	3,058520592	2,99466495	3,05738315
1000	241,937	477,4566	945,6163	3,204372626	3,198188275	3,20216572





En el caso de 4 hilos, vemos que la mejor planificación es *static* puesto que observamos en las gráficas que es la que mejor escala. Al aumentar el número de objetos a 1000 *static* se comporta casi de la misma forma que *guided* (gráfica de la derecha).





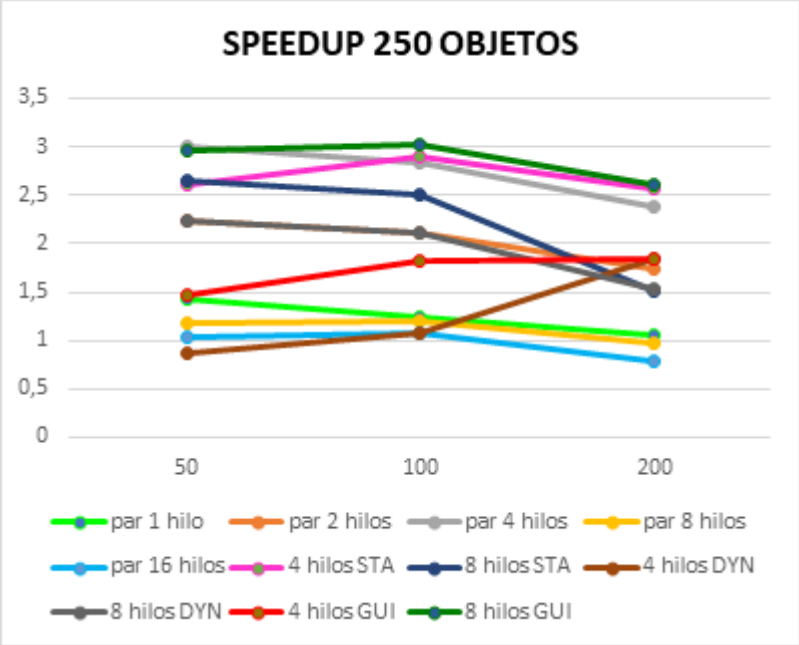
En el caso de 8 hilos, tanto para 250 objetos como para 1000 objetos vemos que la mejor planificación es *guided* ya que consigue el menor tiempo de ejecución. Además, vemos que la planificación *static* escala peor y acaba siendo la peor opción consumiendo bastante más tiempo. Con *guided* tarda unos 900 ms aproximadamente y con *static* 2400 ms, *static* tarda casi el triple de tiempo.

Como conclusión de estas pruebas, es importante señalar que para nuestro caso la mejor planificación es *guided* con 8 hilos puesto que tarda 900 ms (1000 objetos) y con 4 hilos y *guided* tarda 1100ms (1000 objetos). No obstante, hay que señalar que planificación *guided* con 250 objetos y 4 hilos se comporta peor que *guided* con 8 hilos y 250 objetos. Aun así, *guided* con 8 hilos y 250 objetos obtiene un tiempo de ejecución muy similar al de 4 hilos, 250 objetos y planificación *static*.

Por tanto, tras haber realizado estos análisis, nuestro mejor caso de la versión paralela es empleando la planificación *guided* y 8 hilos con 1000 objetos (900 ms). Aunque si no empleamos planificación el mejor caso de la versión paralela es en el que se emplean 4 hilos con 1000 objetos (1200 ms). Con 250 objetos pasa que la mejor versión paralela sigue siendo la de 8 hilos con planificación *guided*. De modo que la paralela con 8 hilos y *guided* es la que consigue menor tiempo de ejecución, tanto como para 250 objetos como para 1000, por tanto es la que mejor escala. Sin embargo, esto se verificará más adelante en las gráficas del speedup, donde la mejor versión paralela es la que tenga el speedup más alto y, por tanto, es la que se entregará.

4.9.- Comparativa de speedup's

En estas gráficas hemos comparado los speedup de todas las paralelizaciones realizadas respecto a la secuencial con objeto de analizarlas conjuntamente. Se presentan los datos de los casos extremos de 250 y 1000 objetos.



Viendo estas gráficas del speedup, la mejor versión paralela es aquella que tiene un mayor speedup. Tal y como hemos indicado anteriormente, la versión paralela de 8 hilos y planificación *guided* (GUI) es la que menor tiempo de ejecución tiene, tanto con 250 objetos como con 1000. Por ello, en estas gráficas del speed up, vemos que la que tiene un speed up mayor es la versión 8 hilos con planificación guided. De modo que, esta es la versión paralela que entregaremos. Además, como también mencionamos anteriormente la versión paralela de 4 hilos con planificación guided consigue un speed up alto.

Por otro lado, si nos fijamos en ambas gráficas en la línea de paralela 1 hilo, con 1000 objetos, vemos que el speed up es 1, por lo que paralelizar con 1 hilo es como ejecutar el código de forma secuencial. Sin embargo, hay que señalar que con la paralela un hilo y 250 objetos hay un speed up de 1,5 para 50 iteraciones, pero a medida que aumenta el número iteraciones, el speed up se reduce a 1, por lo que no hay ninguna mejora.

5. Pruebas realizadas.

Para conseguir los resultados mencionados anteriormente, se han calculado el tiempo medio total de ejecución del programa para 1,2,4,8 y 16 hilos con 250, 500 y 1000 objetos. Por otro lado, para el caso de 4 y 8 hilos se ha calculado el tiempo medio total empleando planificación *static, dynamic y guided* con 250 objetos y con 500 y 1000 aunque sólo hemos mostrado gráficamente los casos extremos. para ver así la escalabilidad de la paralelización.

Para la realización de las pruebas, hemos ejecutado un script para agilizar el trabajo. Se han ido comparando los resultados obtenidos en la ejecución secuencial y paralela para verificar que coincidían.

6. Conclusiones

Como conclusión general, tras estudiar el tiempo de ejecución del código con los casos definidos anteriormente se obtiene que la mejor versión paralela es la de 8 hilos con planificación *guided* y los 4 cores del sistema gestionan mejor los 8 hilos con esta planificación. Además, esta versión es la que mejor escala ya que tanto con 250 objetos como con 500 o 1000 objetos se obtiene el mayor speed up de todas las pruebas realizadas.