Práctica de programación paralela con OpenMP

María Benavente Gómez 100332832
Irene Martínez Castillo 100346051
Jorge San Martín López 100346154

G82, Grupo 17

28 November 2018

Table of Contents

1 Introducción **2**

2 Versión secuencial **2**
2.1 Implementación y decisiones de arquitectura 2
2.2 Problemas encontrados 3

3 Versión paralela **3**
3.1 Implementación y decisiones de arquitectura 3
3.2 Problemas encontrados 4

4 Evaluación del rendimiento **4**

5 Pruebas realizadas **9**

6 Conclusiones **9**

1 Introducción

El objetivo de este proyecto es crear dos versiones de un programa en C++ 14, una de forma secuencial y otra de manera paralela usando las herramientas aprendidas en clase. Esta última se realizará usando la API *OpenMP*. Ambas versiones deberán ser equivalentes desde un punto de vista semántico, pero tendrán distintos rendimientos a la hora de ejecutarse. El programa a desarrollar es un simulador, no demasiado complejo ni lleno de detalles, de la física de los asteroides en el espacio, plasmando en los resultados las diferentes interacciones gravitacionales que ocurren entre cuerpos.

Tanto en la carpeta *seq* como *par* se han incluido los archivos $Makefile$ para compilar el código y para limpiar el directorio tras la ejecución. Además, todos los archivos adicionales que se mencionan en la memoria se han entregado junto con el código en una carpeta adicional llamada *archivos_auxiliares*.

2 Versión secuencial

2.1 Implementación y decisiones de arquitectura

Aparte del propio algoritmo implementado en el programa, el cual se ha realizado siguiendo los pasos detallados en el enunciado, se han tomado ciertas decisiones sobre la taxonomía, la organización y creación de nuestras variables y estructuras de datos de nuestro programa.

Para empezar, hemos declarado las constantes que se usarán en el código usando solamente enteros (*width*, *height*, la distancia mínima d_{min}, la media de la *masa* y su desviación típica SDM) y números en coma flotante de doble precisión (la constante gravitatoria *gravity* y el intervalo de tiempo entre iteraciones Δt).

Los asteroides son implementados con un *struct* que almacena: la posición (x, y), su masa, su velocidad (v_x, v_y) y su aceleración (a_x, a_y). Estos últimos son inicializados a 0 previo a su cálculo. De igual manera son implementados los planetas, con un *struct*, pero a diferencia con los asteroides, no necesitamos guardar la velocidad ni la aceleración.

Las posiciones y masas iniciales de los cuerpos son generados a través de distribuciones de probabilidad ya implementadas en la librería *random*. El programa recibe una semilla, la cual se implementa para crear un generador que usa la distribución apropiada (uniforme para las posiciones y normal para las masas) Además, debemos especificar los rangos de valores para generar los datos iniciales. Cabe resaltar la importancia de hacer las llamadas en orden correcto para que coincidan los datos iniciales con la versión final proporcionada. Los planetas deben encontrarse en los bordes, y como hay 4 bordes hacemos $a \equiv i \mod 4$, siendo i el número del planeta; si $a = 0$ se encuentra en el borde de la izquierda ($x = 0$), $a = 1$ aparece en el eje inferior ($y = 0$), $a = 2$ para el borde derecho ($x = width$) y $a = 3$ para el eje inferior ($y = height$) y mientras que el resto de parámetros se generan según la misma distribución anterior. Todos estos datos generados son guardados en el archivo de texto *init_conf.txt*, que se crea si no existe o se sobrescribe en caso contrario. A este fichero se le incluye una cabecera con los argumentos pasados al ejecutar el programa.

En un principio, no sabíamos qué estructura usar para almacenar las distancias y ángulos de incidencias entre los cuerpos. Por ello creamos un pequeño programa llamado *mini_vector.cpp* con tres parámetros para especificar: la estructura de datos (1 para los vectores y 2 usando memoria dinámica), el tamaño de ésta y el número de veces que debía ejecutarse. Este programa simplemente crea la estructura, la recorre dándole valores e imprimiendo por pantalla, se calculan los tiempos de todas las ejecuciones y al terminar imprime la media de los tiempos. La siguiente tabla resume los resultados obtenidos en las pruebas:

	Vector tam. 1000	**Vector tam. 2000**	**M. dinámica tam. 1000**	**M. dinámica tam. 2000**
5000 iter.	2.411400 ms	4.93779 ms	2.371256 ms	4.831160 ms
	2.477350 ms	4.907590 ms	2.36218 ms	4.748170 ms
10000 iter.	2.526570 ms	5.064720 ms	2.236320 ms	4.759620 ms
	2.504210 ms	4.873300 ms	2.345980 ms	4.649610 ms

A simple vista no parece que haya una diferencia significativa (aunque se ve cierta mejora usando memoria dinámica) y atribuimos las posibles fluctuaciones a la ejecución del ordenador (este programa fue ejecutado en local, en un MacBook Pro con un Intel i5 y 8GB de RAM), las cuales se encuentran fuera de nuestro alcance. Como en un principio empezamos usando memoria dinámica y hemos encontrado una serie de argumentos a nuestro favor, decidimos usar esta estructura.

Otra decisión de diseño ha sido el uso de `++i` en lugar de `i++` dentro de los bucles `for`. Cuando el incremental se escribe delante, no se guarda el estado actual, sino que se incrementa el valor y retorna el estado anterior. Como para usarlo como contador del bucle no necesitamos que se guarde el estado actual, hacemos uso de `++i` y así se realiza una operación menos.

2.2 Problemas encontrados

Durante el desarrollo de la versión secuencial nos encontramos principalmente con dos problemas: uno relacionado con la distribución por el espacio de los asteroides y el otro a la hora de calcular las fuerzas que se ejercen sobre ellos.

El primer problema se debió a la forma en la que hacíamos las llamadas al generador de números aleatorios, que crea una lista a raíz de una semilla introducida. Nuestro código hacía todas las llamadas posibles al generador a la hora de crear los valores de los planetas, por lo que se tomaban de la lista de números valores que finalmente no se utilizaban.

Debido a esto, los planetas no se distribuían correctamente por el espacio ya que los números aparecían desplazados. Además de este problema, algunas de las llamadas que hacíamos al generador de números se realizaban desde dentro del constructor, y en C++ no se puede prever el orden en el que los argumentos van a ser utilizados en la llamada al constructor. Estos errores provocaban que nuestro $init\_config.txt$ no coincidiera con el que nos proporcionaron.

El segundo problema se debió a una interpretación errónea del guión de la práctica. A la hora de calcular las fuerzas ejercidas sobre los asteroides aplicábamos el máximo de `200N` sobre las componentes de la fuerza, cuando este límite debía aplicarse sobre el módulo de la fuerza. Con nuestra interpretación, los módulos de las fuerzas tenían una cota superior mucho más elevada que lo que en realidad pedía el guión, llegando a ser hasta un 41% mayores. A causa de esto, nuestros asteroides alcanzaban velocidades bastante más elevadas, que producían un importante error el cual se propagaría a través de las sucesivas iteraciones.

3 Versión paralela

3.1 Implementación y decisiones de arquitectura

En esta versión del programa no sólo hemos paralelizado el código existente, sino que hemos hecho modificaciones sobre el mismo para que resultase más "paralelizable". El ejemplo más claro de esta decisión es la forma en la que agregamos las diferentes fuerzas ejercidas sobre un mismo asteroide. Esta suma debe hacerse en el mismo orden siempre, ya que esta operación no es conmutativa cuando se realiza sobre coma flotante puesto que se pierde precisión con cada suma que hacemos.

En la parte secuencial, decidíamos hacer las sumas de las fuerzas en el mismo bucle en el cual las calculábamos y por tanto el bucle no era paralelizable, pues los hilos alterarían el orden de la suma. Por ello decidimos crear una matriz que relacionase las fuerzas ejercidas por los planetas sobre los asteroides en cada iteración al igual que hacíamos con las distancias y las pendientes. De esta forma conseguimos que las lecturas y escrituras fuesen independientes al bucle y así los hilos pudieran realizar estas operaciones sin provocar condiciones de carrera. Una vez terminados todos los cálculos, se suman las distintas fuerzas ejercidas para cada asteroide de manera secuencial, evitándose de esta forma su desorden. Gracias a esto, hemos podido paralelizar sin miedo a condiciones de carrera todo el cálculo de fuerzas, que es la parte del código que más accesos a memoria hace y además computacionalmente más costosa.

En cuanto a la propia paralelización, en un principio se intentó aplicar a todos los bucles de nuestro programa, pero esta implementación no fue exitosa ya que para acceder a algunos datos necesitamos hacerlo de manera secuencial (por razones explicadas previamente). Si paralelizamos estas partes, necesitaríamos establecer el *scheduler* a tipo *atómico* para que sólo uno de los hilos acceda a esa parte en cada instante, suponiendo esto una demora en la ejecución. Estos bucles paralelizados han sido comentados en el código fuente.

Los únicos bucles que han sido paralelizables con éxito son:

- El bucle en la línea 164, que itera sobre todos los asteroides y a su vez itera sobre los asteroides de nuevo para calcular las fueras entre asteroides-asteroides y después itera sobre los planetas para calcular las fuerzas entre asteroides-planetas. Este bucle es uno de los que tiene más carga computacional de todo el programa, ya que realiza funciones exponenciales y funciones trigonométricas. Al paralelizar este bucle, se han conseguido mejoras del rendimiento.
- El bucle de la línea 243, calcula las colisiones de los asteroides. Si paralelizamos por separado este bucle de la misma manera que el anterior, con un *scheduler* de tipo *dynamic* al principio observamos que realiza una ejecución similar a la versión secuencial. Sin embargo, al paralelizar ambos bucles descubrimos que obtenemos una versión paralela mucho más rápida que la secuencial. Estos datos serán comentados en detalle en la sección 4 de este documento.

3.2 Problemas encontrados

El principal problema al que nos enfrentamos fue ajustar los resultados, ya que no teníamos en cuenta diferencia de decimales causada por la coma flotante. Este problema lo tuvimos al crear la parte paralela, ya que al acceder a la memoria en distinto orden no obtuvimos los mismos resultados que en la parte secuencial.

Además, también vimos mucha inconsistencia en el tiempo de ejecución entre programas, pero esto se debía a que ejecutábamos en Guernika, donde se comparten los recursos entre todos los usuarios conectados. Este problema se solucionó cuando empezamos a ejecutar el código un ordenador local, situado en el Laboratorio 4.0.F16.

4 Evaluación del rendimiento

Para mejorar la calidad y eficiencia de este paso, hemos automatizado el proceso de evaluación del rendimiento a través de dos *scripts* escritos en el lenguaje de scripting *Python*: el primero, recibe los parámetros del programa (número de asteroides, iteraciones, ...), el número de veces de ejecución y la versión del programa (secuencial o paralela), y al ejecutarse escribe en un *.txt* los parámetros introducidos y la media de los tiempos de ejecución; el segundo *parsea* estos resultados y los almacena en un archivo *.xlsx*.

En un principio empezamos evaluando la práctica desde Guernika, pero los resultados no eran consistentes ya que en el servidor se comparten los recursos con el resto de usuarios conectados. La evaluación de rendimiento final se realizó en los ordenadores del laboratorio de la universidad. Estos ordenadores tienen un procesador *Intel Core i5-4460* compuesto de 4 Cores y 4 hilos. Para más información sobre estos ordenadores, adjuntamos el archivo *ordenador_info.txt*.

Para evaluar el rendimiento del código hemos hecho uso de la librería *chrono*, que permite saber los microsegundos que tarda en ejecutarse un programa. Todos los resultados presentados son la media de tiempo tras 20 ejecuciones. Para compilar las dos versiones del programa utilizamos los siguientes comandos, que incluyen diversos *flags* que personalizan la compilación:

- *Versión secuencial:* `g++ -std=c++14 nasteroids-seq.cpp -Wall -Wextra -Wno-deprecated -Werror -pedantic -pedantic-errors -o salida-seq.out`
- *Versión paralela:* `g++ -std=c++14 nasteroids-par.cpp -Wall -Wextra -Wno-deprecated -Werror -pedantic -pedantic-errors -fopenmp -o salida-par.out`

A continuación presentamos las pruebas realizadas a las dos versiones del código, junto con sus resultados y algunos gráficos. Los datos obtenidos en los experimentos para realizar las gráficas se encuentran en */archivos_auxiliares/gráficas.xlsx* (también existe una versión *.pdf* aunque su visualización es peor). Todos los datos generados son la media de los tiempos de 20 repeticiones.

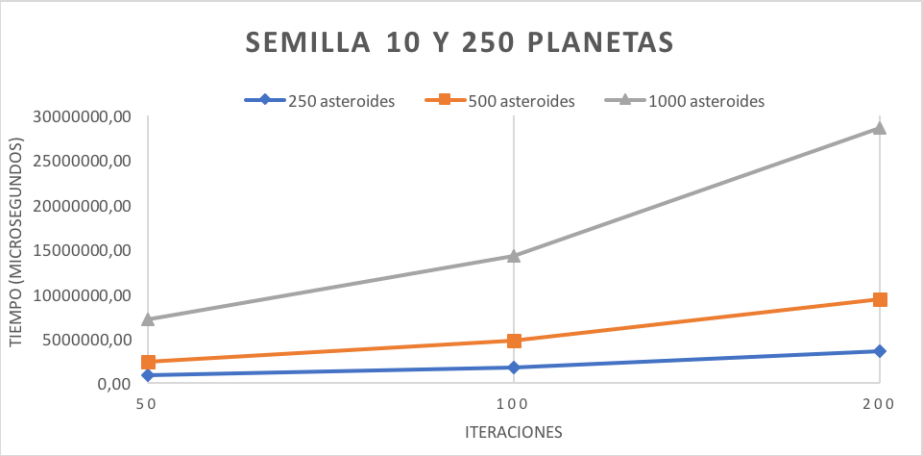

Figure 1: Prueba versión secuencial variando iteraciones

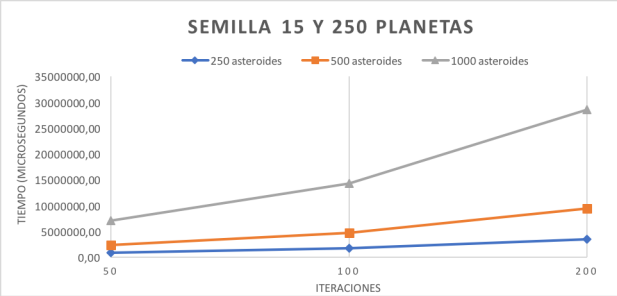

Figure 2: Prueba versión secuencial variando semillas

En la primeras pruebas realizadas, comprobamos en nuestra versión secuencial que una variación de semilla no supone gran variación del tiempo de ejecución. De igual manera, observamos que al aumentar el número de iteraciones y de asteroides, el tiempo de ejecución crece de manera (el código contiene 2 bucles anidados), tal y como se puede observar en las dos primeros gráficos.

Si variamos el número de planetas y dejamos fijo el número de asteroides en 250, observamos un crecimiento cuadrático, igual que en el caso anterior. Sin embargo, debemos tener en cuenta que la magnitud que alcanza la variable tiempo llega a ser casi 3 veces menor en este caso que en el anterior.

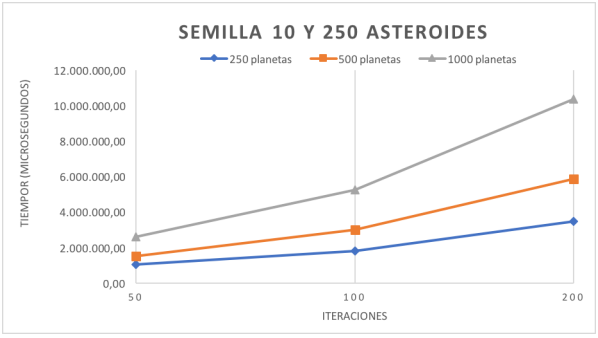

Figure 3: Prueba versión secuencial variando planetas

Para determinar los bucles que son convenientemente más rápidos hemos hecho las siguientes pruebas con un cambio de asteroides, que se pueden ver representadas en los dos siguientes gráficos. Observamos cómo hay uno de los dos bucles que es más lento que el otro, pero a su vez la implementación más rápida es la conjunta.

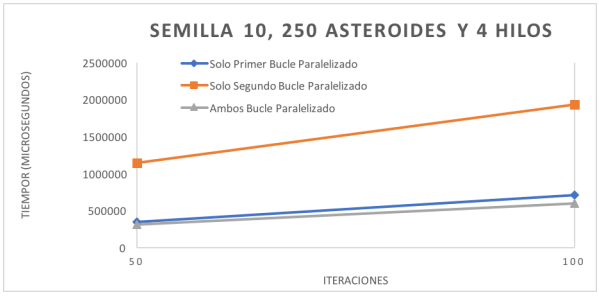

Figure 4: Pruebas diferentes versiones paralelas

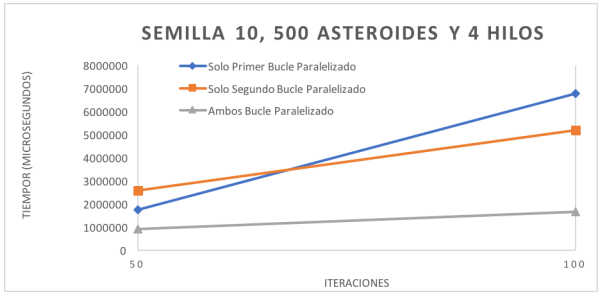

Figure 5: Pruebas diferentes versiones paralelas

En estas pruebas podemos apreciar cómo son resultados experimentales y no siempre siguen los resultados esperados, aún así siguen las líneas que esperábamos para comprobar que la combinación de paralelización de ambos bucles es la óptima.

A continuación probamos el tipo de *scheduler* óptimo para nuestros bucles. Como se puede observar en el siguiente gráfico, la mejor configuración consiste en establecer ambos *scheduler* en tipo *dynamic*.

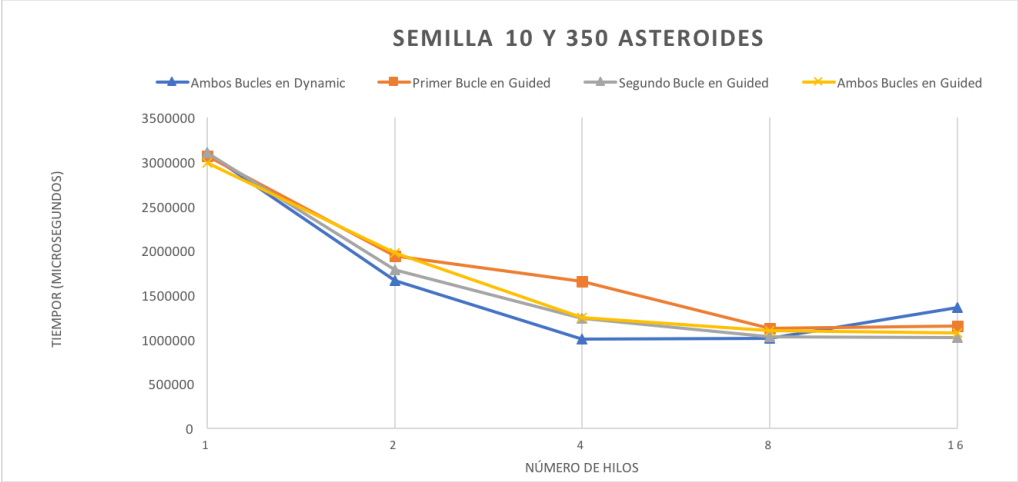

Figure 6: Prueba versión paralela variando *schedulers*

En este gráfico también observamos cómo de todos los hilos que hemos implementado el que mejor rendimiento tiene es el de 4. Esto se debe a que el ordenador donde hemos realizado estas pruebas tenía el mismo número de cores que hilos en ejecución. Al aumentar el número de hilos, el *scheduler* tiene que repartir los hilos restantes entre los cores que ya están siendo utilizados.

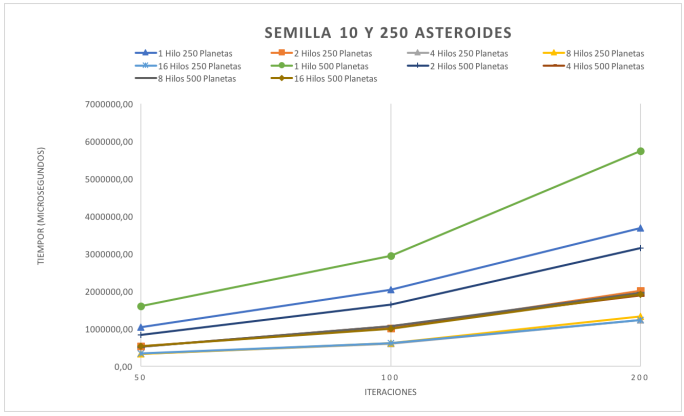

Figure 7: Prueba versión paralela variando planetas

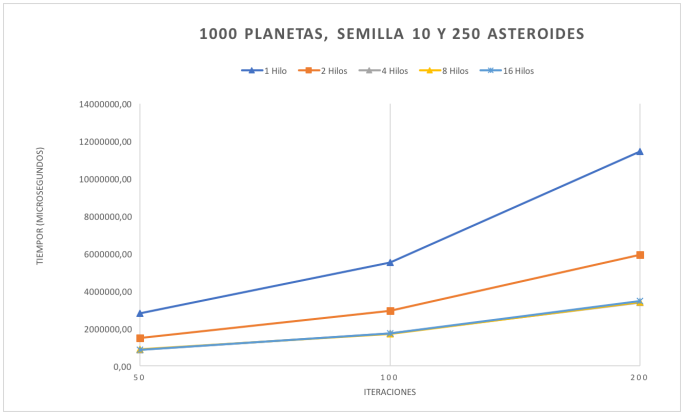

Figure 8: Prueba versión paralela con 1000 planetas

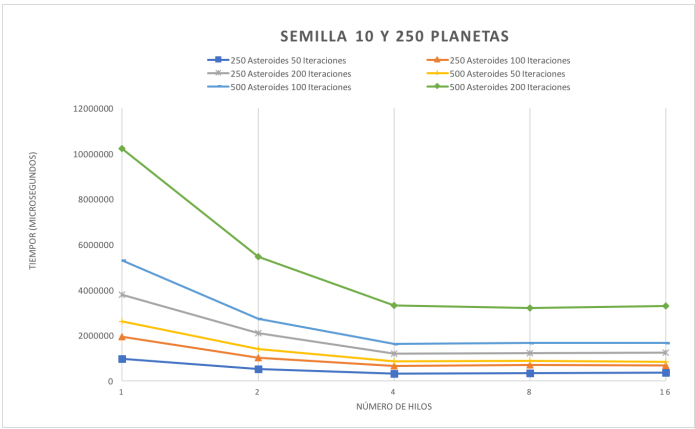

Figure 9: Prueba versión paralela variando asteroides

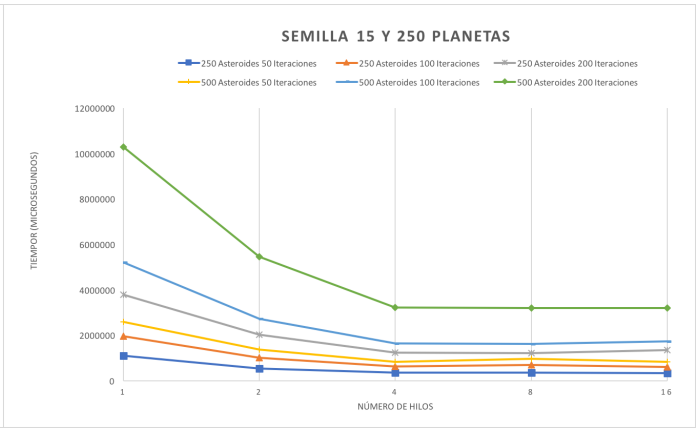

Figure 10: Prueba versión paralela variando semillas

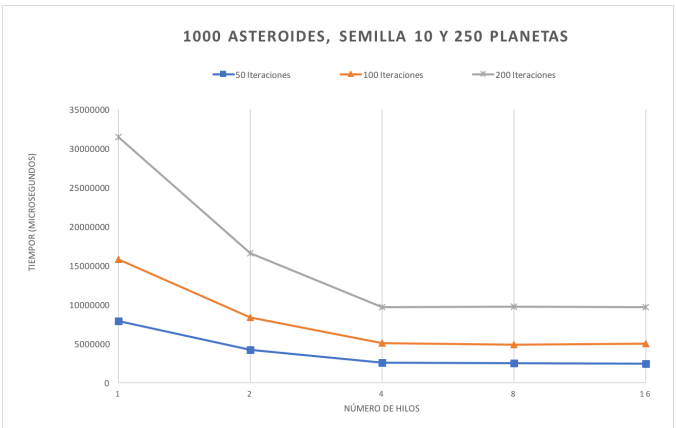

Figure 11: Prueba paralela, 1000 asteroides y seed 10

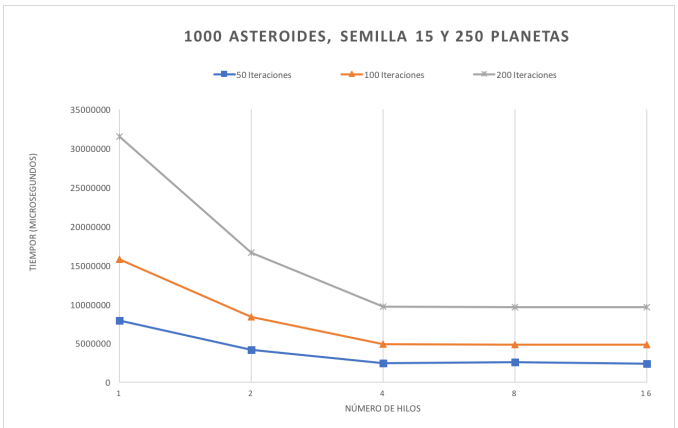

Figure 12: Prueba paralela, 1000 asteroides y seed 15

De estas gráficas concluimos que los resultados obtenidos no dependen de la semilla usada, que la paralelización consigue resultados bastante mejores si aumentamos el número de hilos a 4 (con 8 y 16 no hay diferencia notable debido a la configuración del ordenador) y que aún variando el número de cuerpos las mejoras se mantienen (resaltamos que el aumento del número de asteroides produce tiempos 3 veces más

grandes que haciendo lo mismo con los planetas).

Hemos calculado el *speedup* obtenido con la mejora, hallando el ratio entre el tiempo de la ejecución del código secuencial y el paralelo:

$$Speedup = \frac{T(secuencial)}{T(paralelo)} \tag{1}$$

Con los datos previos obtenidos en las evaluaciones, se han calculado los distintos *speedup* en cada situación como se muestra en la *Figure 13*.

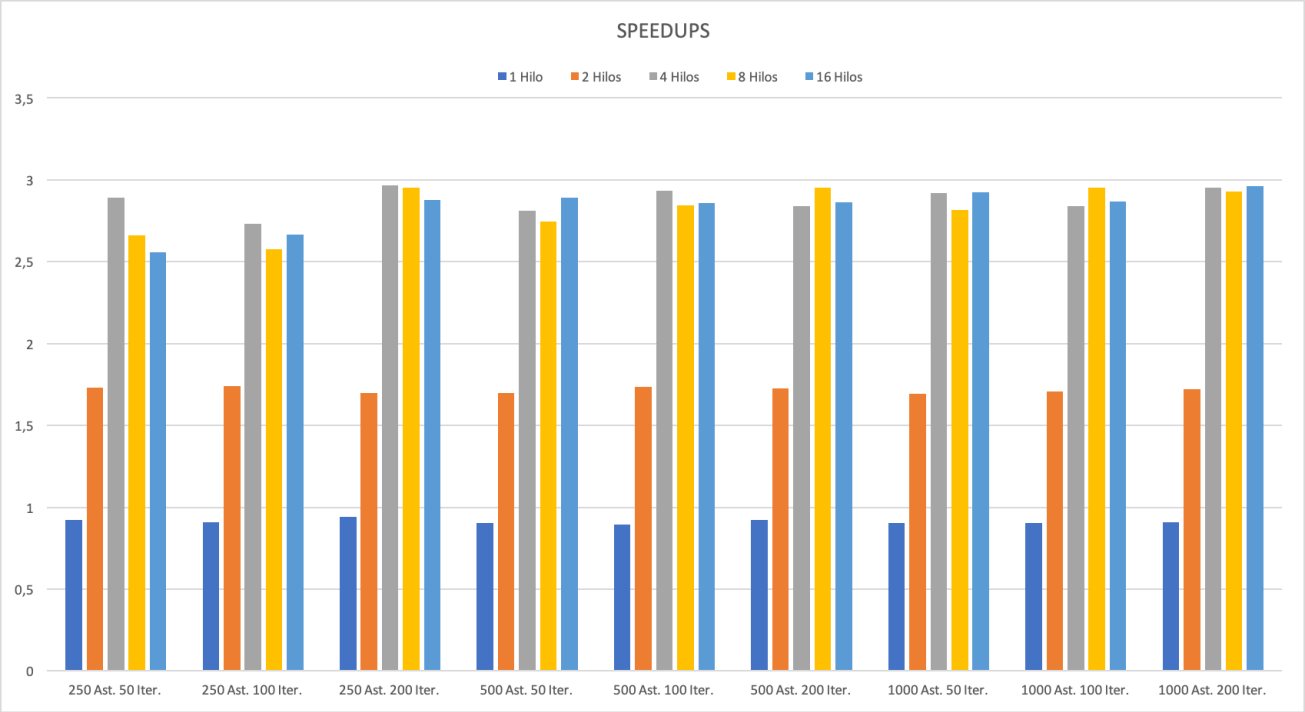

Figure 13: Speedup para diferentes números de hilos

Para comparar los diferentes *speedup*, se han calculado las medias de estos para cada hilo y plasmado en la siguiente tabla:

	1 Hilo	**2 Hilos**	**4 Hilos**	**8 Hilos**	**16 Hilos**
Media Speedup	0,908	1,719	2,865	2,810	2,824

Como se observa, el programa paralelizado ejecutándose con 4 hilos es el que produce el mayor *speedup*, produciéndose una concordancia con los argumentos descritos en la *Figure 6*. Cabe resaltar, el hecho de que ejecutándose con sólo un hilo, el programa se ralentiza debido a las modificaciones en el código para poder paralelizar un mayor porcentaje del código y que con solamente 2 hilos, los tiempos mejoran de media un 71%.

5 Pruebas realizadas

Para asegurarnos del correcto funcionamiento del código, utilizamos las siguientes pruebas:

- Ejecución del comando `diff` para comprobar de que la salida de nuestro código era igual a la del ejecutable que teníamos como referencia.
- Impresión por pantalla del estado de los asteroides y planetas en cada momento de tiempo, y posterior comparación del valor de las posiciones, velocidad y aceleración con el archivo *step_by_step*. Gracias a esto podíamos hacer un *debugging* más concreto al comparar los errores que aparecían en cada iteración.

Haciendo uso de estos dos mecanismos pudimos ir arreglando los problemas de programación o interpretación del guión que fueron surgieron a lo largo de la realización de la práctica.

6 Conclusiones

La principal conclusión que hemos sacado de la realización de esta práctica, es que gracias a *OpenMP* se pueden alcanzar *speedups* muy altos de manera sencilla. Cuando se analizan con detalle bloques de código se puede ver que gran parte de ellos son paralelizables, y explotando este recurso se puede mejorar mucho la eficiencia de un programa. Además, la API *OpenMP* facilita mucho la implementación de paralelismo, ya que utilizando sus `pragmas` se pueden crear y juntar hilos con sólo un par de líneas de código.