Arquitectura de Computadores

Práctica de programación paralela con OpenMP

Sandra Lázaro Pérez 100363849
Sabrina Riesgo Reyes 100363834
Rafael Rus Rus 100363907
Laura Yunta García 100363785

Grupo 82

Índice

1. Introducción.

El objetivo del desarrollo de la práctica es conocer en profundidad la optimización de programas secuenciales y los modelos de programación paralela en arquitecturas de memoria compartida utilizando el lenguaje de programación C++ y las técnicas de paralelismo de OpenMP.

El proyecto consiste en realizar un programa que simule el movimiento de un conjunto de asteroides a lo largo del tiempo dependiendo de la atracción gravitatoria que se produce entre ellos.

La estructura del documento se dividirá en las explicaciones de la implementación y optimización de las partes secuencial y paralela. Además de la evaluación del rendimiento que comparará el programa inicial, la versión secuencial y la versión paralela optimizada. Por último, mostraremos las pruebas realizadas para comprobar el correcto funcionamiento de nuestro código, junto con las conclusiones del trabajo.

2. Versión secuencial.

La implementación de la parte secuencial comienza con la declaración de todas las variables especificadas por el enunciado, además de definir los struct asteroide (que tendrá las componentes x e y para su respectiva posición, velocidad, aceleración y fuerza) y planeta (que tendrá la componente x e y de la posición además de su masa) que representarán a cada planeta o asteroide que compondrán el desarrollo del ejercicio. Posteriormente, se declaran las siguientes funciones para los diferentes cálculos necesarios:

- Calculo_fuerza_atraccion_asteroide: Cuya función es calcular la fuerza de atracción de dos asteroides introducidos por parámetro utilizando la distancia entre ellos y las masas según indica la fórmula del enunciado.
- Calculo_fuerza_atraccion_planeta: Cuya función es calcular la fuerza de atracción de los planetas introducidos por parámetro utilizando la distancia entre ellos y las masas según indica la fórmula del enunciado.
- Calculo_distancia_asteroide: Cuya función es calcular la distancia entre dos asteroides introducidos por parámetro utilizando las posiciones de ambos según indica la fórmula del enunciado.
- Calculo_distancia_planeta: Cuya función es calcular la distancia entre dos asteroides introducidos por parámetro utilizando las posiciones de ambos según indica la fórmula del enunciado.
- Calculo_pendiente_asteroide: Cuya función es calcular la pendiente entre dos asteroides introducidas por parámetro utilizando sus posiciones según indica la fórmula del enunciado.
- Calculo_pendiente_planeta: Cuya función es calcular la pendiente entre dos planetas introducidas por parámetro utilizando sus posiciones según indica la fórmula del enunciado
- Calculo_angulo_asteroide: Cuya función es calcular el ángulo de inclinación entre dos asteroides a partir de su pendiente utilizando la fórmula que aparece en el enunciado de la práctica.
- Calculo_angulo_planeta: Cuya función es calcular el ángulo de inclinación entre dos planetas a partir de su pendiente utilizando la fórmula que aparece en el enunciado de la práctica.

Finalmente, todas las funciones se ejecutarán en la función main a la cual se le pasan el número de argumentos que se han introducido en la sentencia de ejecución y el contenido de cada uno de ellos.

Inicialmente, se comprueba que el número de argumentos es el correcto, es decir 5 donde se incluyen el número de iteraciones a realizar, número de asteroides y planetas y la semilla del escenario. Posteriormente, gracias a los contadores i y j (donde i hace iterar entre los diferentes argumentos y j entre los diferentes caracteres introducidos de cada argumento) se podrá comprobar si cada uno de los campos de la sentencia de entrada son correctos. Es decir, que todos los argumentos cuya i es mayor a 1 deben tener valores entre 0 y 9 (incluidos) excepto el argumento cuya i es 4 que debe tener un valor mayor entre 1 y 9 (incluidos). Para ello se utilizará el contador j que sumará cada carácter del argumento y comprobará que el resultado no es 48 (0 en código ASCII), en caso de que no se cumpla alguna de estas restricciones se mostrará un error por consola.

Finalmente, una vez que se han comprobado que todos los argumentos son válidos los guardamos en sus respectivas variables para su posterior uso, se calcularán las distribuciones aleatorias, es decir la posición en el espacio bidimensional y la masa de cada asteroide y planeta y se definen los vectores de los structs asteroide y planeta cuyo tamaño será el número de asteroides y planetas introducidos en el argumento, respectivamente. Además, se crea un fichero de salida inicial que tendrá por nombre “init_conf.txt” y se enviará la primera cadena al fichero inicial de salida.
A continuación, se envía el resto de cadena al fichero donde inicialmente se enviará los asteroides a los cuales se les asigna una posición bidimensional en el espacio y una masa que tendrá como valor un número decimal con tres decimales de precisión y por último se enviarán los planetas que también se les asignará una posición bidimensional en el espacio y una masa que tendrá como valor un número decimal con tres decimales de precisión. La posición de cada planeta vendrá dada por un orden predeterminado, de forma que el primer planeta se colocará en el borde izquierdo, el segundo en el borde superior, el tercero en el borde derecho y el cuarto en borde inferior. De esta forma, para colocar los siguientes planetas se realizará el módulo 4 del número de planeta que se trate y se colocarán en el orden especificado anteriormente según el resultado de la operación sea 0 (para borde izquierdo), 1 (para borde superior), 2 (para borde derecho), 3 (para borde inferior). Por último, el fichero “init_conf.txt” se cierra.

Posteriormente, el desarrollo del programa entra en un bucle de iteraciones para las cuales se decrementa el contador. Dentro de este bucle, se declaran las “fuerza_total_x” y “fuerza_total_y” de cada asteroide a 0 (ya que ambas actúan como sumatorios de fuerzas y en cada iteración se deben reiniciar y se procede a calcular todas las fuerzas de atracción entre asteroides. Para ello, se declara un bucle que calcula que calculará la fuerza de cada asteroide con el resto de asteroides, de esta forma se calcula la distancia entre ambos asteroides y se comprueba que sea mayor que la constante mínima del mínimo de la distancia; si no lo es, no calcula la fuerza de atracción entre esos dos asteroides). El algoritmo que se realiza finalmente tiene la siguiente forma: Se declara una variable auxiliar para las fuerzas que actúan de forma negativa, se calcula la pendiente entre ambos asteroides y se comprueba que no sea mayor que 1 y menor que -1 (en caso de que ocurriera se realiza un “static_cast <int>” para corregir la pendiente), se calcula el ángulo entre ambos asteroides que se utiliza más tarde para calcular el vector de las fuerzas de atracción, se calcula el módulo de las fuerzas de atracción y en caso de que su valor máximo sea mayor a 200 se trunca por el seno o coseno del ángulo (dependiendo de si es del eje x o y).
A continuación, se calculan todas las fuerzas de atracción entre asteroides y planetas que realiza el mismo algoritmo de cálculo de fuerzas entre asteroides explicado anteriormente, con la salvedad de que no utiliza la variable auxiliar para las fuerzas negativas.
Posteriormente, tras calcular todas las fuerzas se utiliza un bucle que actualiza las posiciones y las nuevas velocidades de cada asteroide partiendo de la fuerza total. Luego se evalúa el rebote de los asteroides con cada borde del espacio (en cada iteración), de esta forma se posicionará a

dos puntos del espacio en cada caso y finalmente, se evalúa el rebote entre asteroides para lo que se creará un vector de booleanos que se encargará de asignar el valor 1 a todos los asteroides que chocan y el valor 0 a aquellos que no lo hacen. Inicialmente, este vector contendrá todos sus valores a 0 y se comprueba el rebote de cada asteroide con el resto del orden, se crea un vector de punteros llamado "choque_asteroides" que se encarga de almacenar las posiciones de los asteroides que chocan y una variable entera "cont" que se encarga de garantizar que en el vector de punteros se almacena en el orden correcto las posiciones de los asteroides que chocan. El desarrollo del algoritmo entrará en bucle que calcula la distancia entre cada par de asteroides y si cada distancia es menor o igual que dos, **"cont" es 0** (el vector "choque_asteroides" está vacío todavía) y el asteroide i tiene valor 0 en el vector "comprobación_rebotes" (todavía no ha chocado con ningún otro asteroides y por tanto puede intercambiarse las velocidades), se añade la posición del asteroide i al vector de punteros, "cont" cambia a 1 y se cambia el valor a 1 del asteroide i en el vector de booleanos "comprobación_asteroides", se añade la posición del asteroide i al vector de punteros y se añaden el resto de asteroides j con los que choca (debido a que "cont" es ahora 1). **En caso de que "cont" sea 1**, es decir, que el vector "choque_asteroides" no esté vacío, se intercambian todas las velocidades, se guardan las del primer asteroide para dárselas al último asteroide; si no es el último, se cambia la velocidad de cada asteroide por la del siguiente en el vector de punteros y si es el último se la da a la del primero.
Para terminar la ejecución de programa, se crea un fichero de salida 3 que se le ha llamado "out.txt" y se le adjudican los datos obtenidos, finalmente se cierra el fichero de salida.

3. Versión paralela.

Respecto a la versión paralela de la práctica cabe destacar que el algoritmo que sigue su desarrollo es el mismo que en la versión secuencial con la modificación de que en los bucles for internos del bucle de iteraciones se ha puesto la línea de código "#pragma omp parallel for" para que se realice su ejecución en paralelo (sólo se han paralizado los bucles for que lo necesitaban, es decir, que se podían paralelizar con el fin de mejorar el rendimiento de la parte secuencial).

También, se añade la línea de código "#pragma omp atomic" para tener condiciones de carrera y además dar atomicidad a los vectores que se utilizan en el bucle donde se calculan las fuerzas de atracción entre cada asteroide con el resto de los asteroides y planetas (esto se hace al actualizar la fuerza_total_x y la fuerza_total_y de cada asteroide en cada iteración).

Por último, se utiliza la línea "#pragma omp ordered" (en la parte donde se calculan los rebotes entre asteroides) para permitir al código al que precede tener una ejecución en orden secuencial. Esto es debido a que se utiliza el método "push_back" a la hora de añadir las posiciones de los asteroides que chocan entre sí en el vector de punteros "choque_asteroides" dentro del bucle for que comprueba los asteroides que chocan (uno por uno).

Después de realizar todas las pruebas de evaluación de la versión paralela y de ver el impacto de la planificación para los casos de 4 y de 8 hilos con los modelos de planificación static, dynamic y guided (tal y como se muestra en los siguientes apartados de la memoria), hemos observado que el código paralelo más eficiente con respecto al secuencial obtenido es utilizando 4 hilos con el modelo de planificación static. Debido a ello, hemos enviado el código paralelo que se ajusta a esas características, añadiendo al inicio del main el "omp_set_num_threads(4)" y junto a cada "#pragma omp parallel for" escribimos "schedule (static)".

4. Pruebas realizadas.

Todas las pruebas han sido realizadas en el mismo ordenador de un aula de informática de la universidad. Las características respecto a la máquina son las siguientes:

Modelo de procesador: Intel(R) Core i5-4460 CPU @ 3.20GHz

Número de cores: 4 cores

Hilos de procesamiento por núcleo: 1

Arquitectura: x86_64

CPU MHz: 1964,648

Caché L1d: 32 KB

Caché L1i: 32 KB

Caché L2: 256 KB

Caché L3: 6144 KB

Ejecutando las diversas pruebas, que se explicarán a continuación, en el sistema operativo Debian 4.9.130-2 y en la versión del compilador 20170516.

El primer tipo de prueba es el de una población de 250 objetos (200 asteroides y 50 planetas) con el número 2000 como semilla. En este caso, el número de asteroides es superior al de planetas.

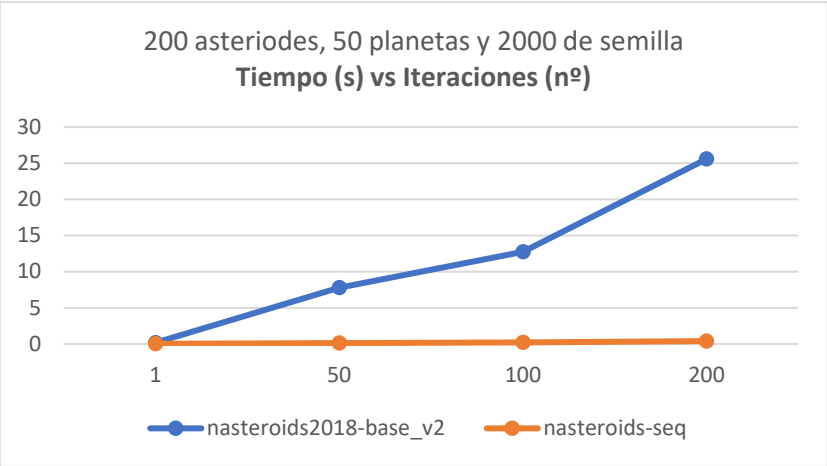
El segundo tipo de prueba es el de una población de 500 objetos (250 asteroides y 250 planetas) con el número 2000 como semilla. En este caso, el número de asteroides y planetas es el mismo.

El tercer tipo de prueba es el de una población de 1000 objetos (450 asteroides y 550 planetas) con el número 2000 como semilla. En este caso, el número de asteroides es inferior al de planetas.

Cada tipo de prueba la hemos realizado con el número de iteraciones marcado en el enunciado: 1, 50, 100 y 200 iteraciones.

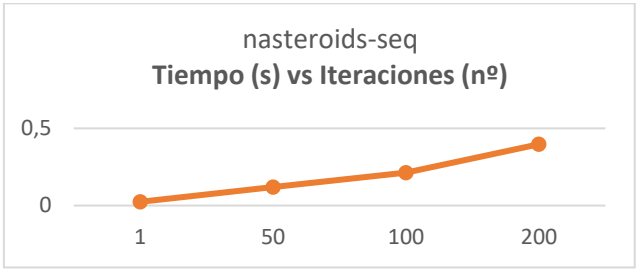
Los programas que hemos ejecutado con estas pruebas son el binario proporcionado en Aula Global, nuestro programa secuencial y nuestro programa paralelo, tanto con diferentes números de hilos (1, 2, 4, 8 y 16) y con los modelos de planificación pedidos (static, dynamic y guided), cada uno de ellos con 4 y 8 hilos.

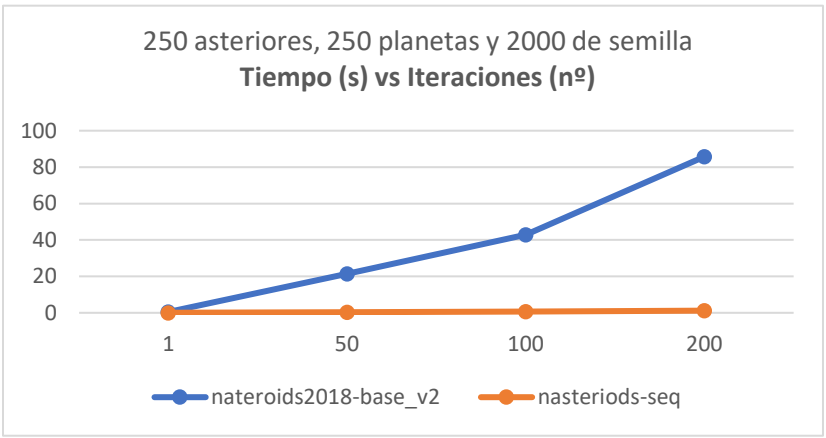
5. Evaluación de rendimiento.



Tras haber realizado las pruebas con la configuración que se muestra en el título de esta gráfica, podemos ver que la eficiencia de nuestro programa secuencial es superior al binario proporcionado, al ser ejecutado en mucho menos tiempo.

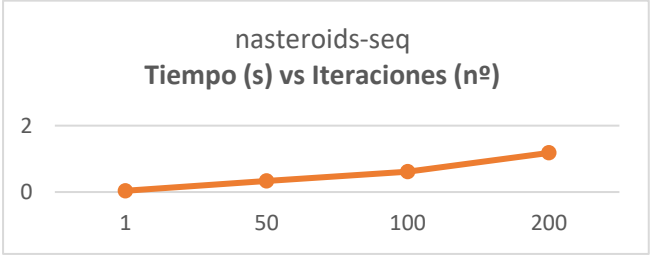
A continuación, se adjunta la gráfica correspondiente al tiempo de ejecución de nuestro programa secuencial, ya que en la anterior gráfica no se aprecian los valores, al ser éstos muy cercanos al 0 y tener una escala menor que el binario.

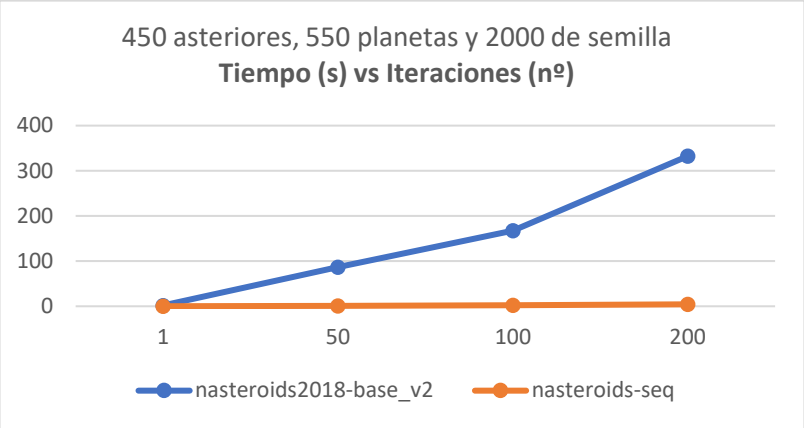




Tras haber realizado las pruebas con la configuración que se muestra en el título de esta gráfica, podemos ver que la eficiencia de nuestro programa secuencial es superior al binario proporcionado, al ser ejecutado en mucho menos tiempo.

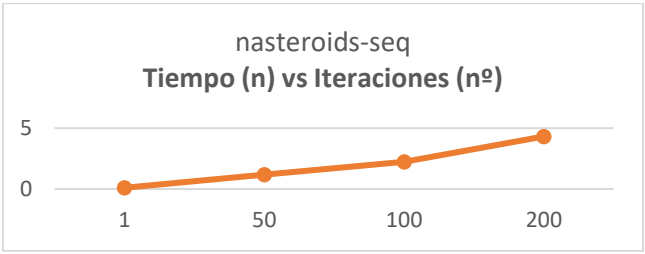
A continuación, se adjunta la gráfica correspondiente al tiempo de ejecución de nuestro programa secuencial, ya que en la anterior gráfica no se aprecian los valores, al ser valores entre 0 y 1,4 y tener una escala menor que el binario.



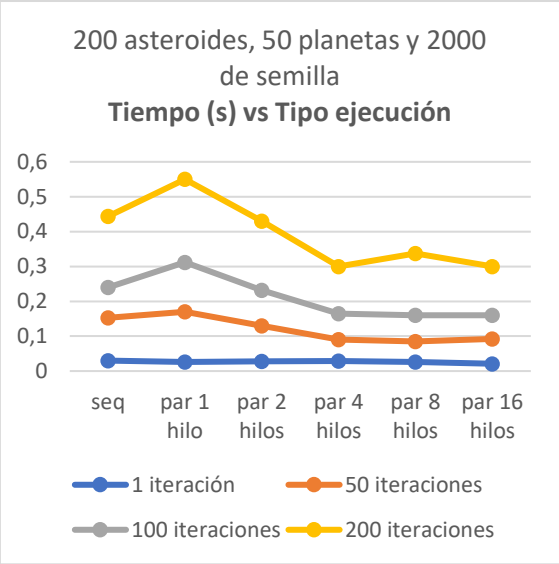


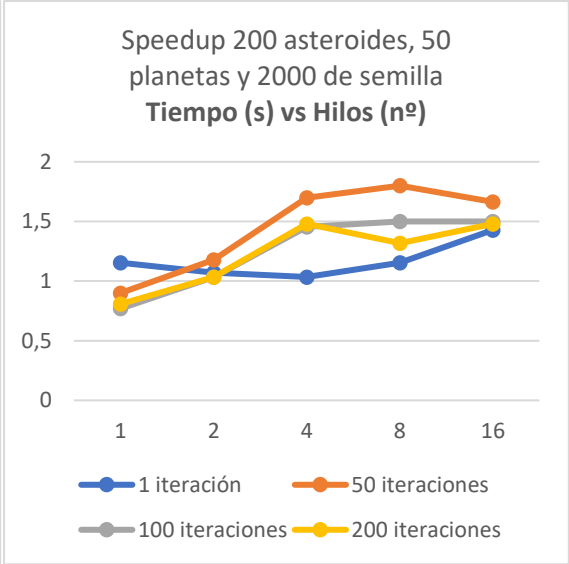
Tras haber realizado las pruebas con la configuración que se muestra en el título de esta gráfica, podemos ver que la eficiencia de nuestro programa secuencial es superior al binario proporcionado, al ser ejecutado en mucho menos tiempo.

A continuación, se adjunta la gráfica correspondiente al tiempo de ejecución de nuestro programa secuencial, ya que en la anterior gráfica no se aprecian los valores, al ser estos valores entre 0 y 5 y tener una escala menor que el binario.

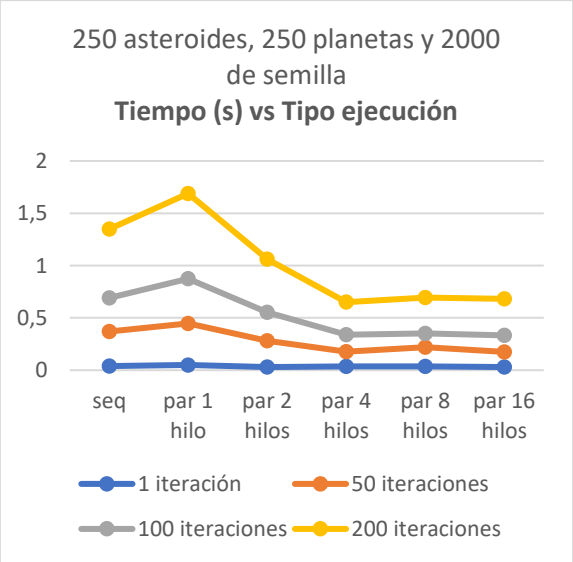


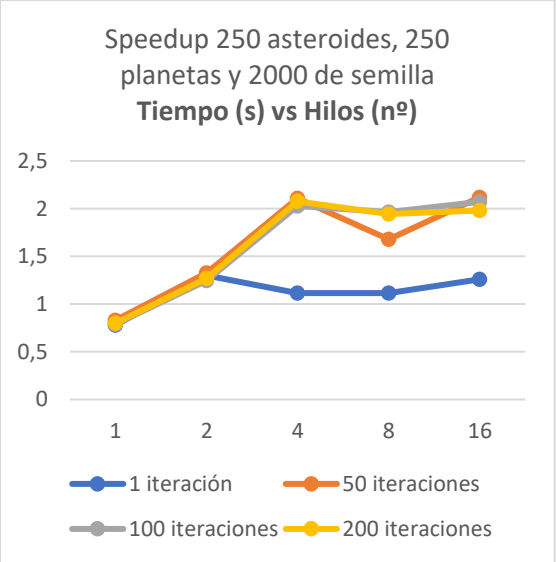
En conclusión, nuestro programa secuencial es más eficiente que el binario proporcionado debido a que éste genera un archivo step_by_step.txt en el que se muestra cada paso en una línea; y al aumentar el número de iteraciones, éste tarda más en terminar de ejecutarse.



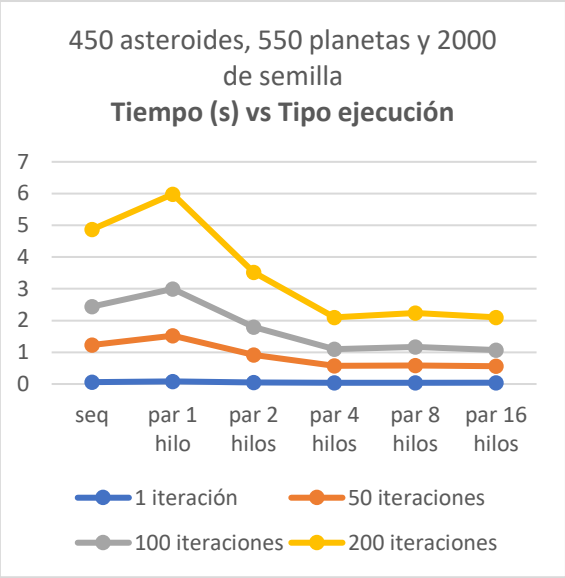


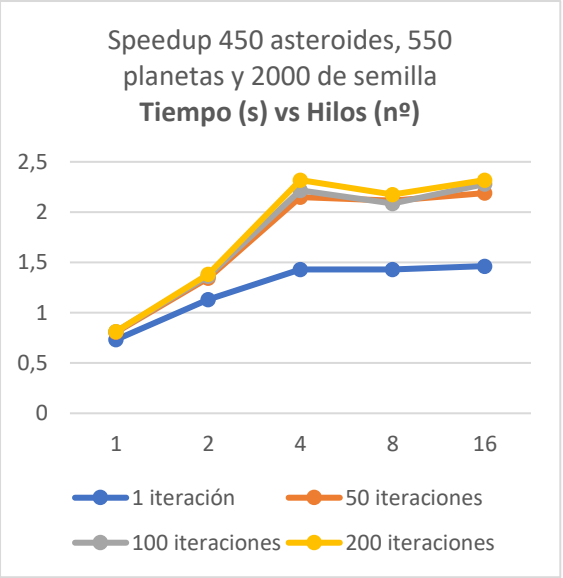
Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos ver que cuanto mayor es el número de iteraciones, mayor es el tiempo de ejecución. Además, podemos observar que el tiempo de ejecución varía dependiendo del número de hilos que utilicemos. En este caso podemos observar que la versión paralela y utilizando un hilo se ejecuta en un tiempo mayor, siendo esta la peor configuración. También podemos concluir que la mejor versión del programa sería la paralela utilizando 4 hilos, aunque no se aprecie una diferencia notable entre todas las configuraciones.





Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos ver que cuanto mayor es el número de iteraciones, mayor es el tiempo de ejecución. Además, podemos observar que el tiempo de ejecución varía dependiendo del número de hilos que utilicemos. En este caso podemos observar que la versión paralela y utilizando un hilo se ejecuta en un tiempo mayor, siendo esta la peor configuración, aunque esto no puede apreciarse con exactitud en el caso de una iteración, ya que los valores son muy parecidos independientemente del número de hilos. También podemos concluir que la mejor versión del programa sería la paralela utilizando 4 hilos, aunque no se aprecie una diferencia notable entre todas las configuraciones y el caso de una iteración el speedup sea ligeramente mayor para 2 y 16 hilos.



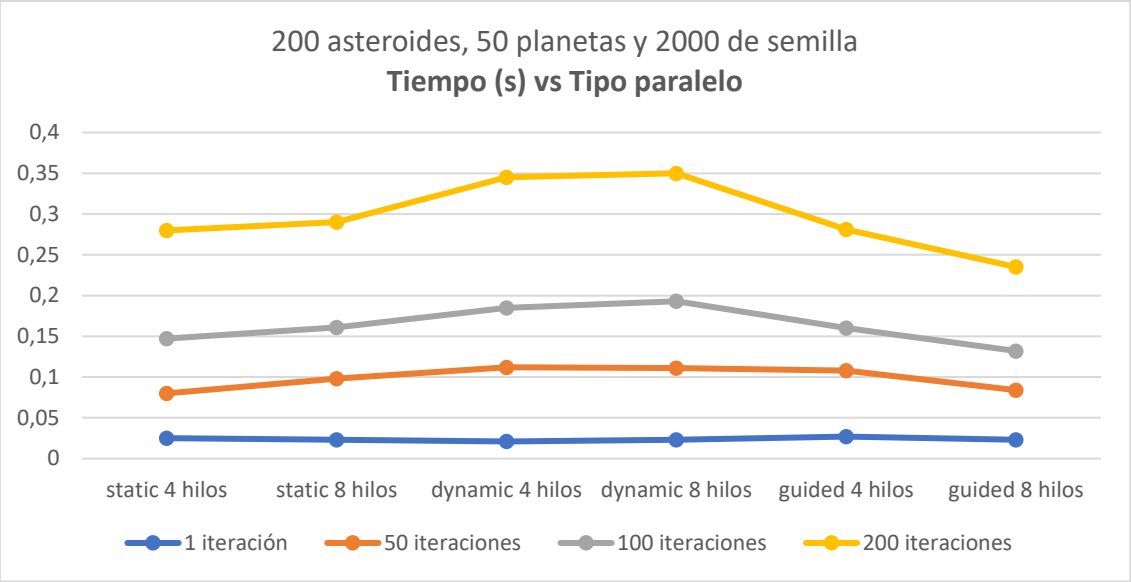


Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos ver que cuanto mayor es el número de iteraciones, mayor es el tiempo de ejecución. Además, podemos observar que el tiempo de ejecución varía dependiendo del número de hilos que utilicemos. En este caso podemos observar que la versión paralela y utilizando un hilo se ejecuta en un tiempo mayor, siendo esta la peor configuración, aunque esto no puede apreciarse con exactitud en el caso de una iteración, ya que los valores son muy parecidos independientemente del número de hilos. También podemos concluir que la mejor versión del programa sería la paralela utilizando 4 hilos, aunque no se aprecie una diferencia notable entre todas las configuraciones.

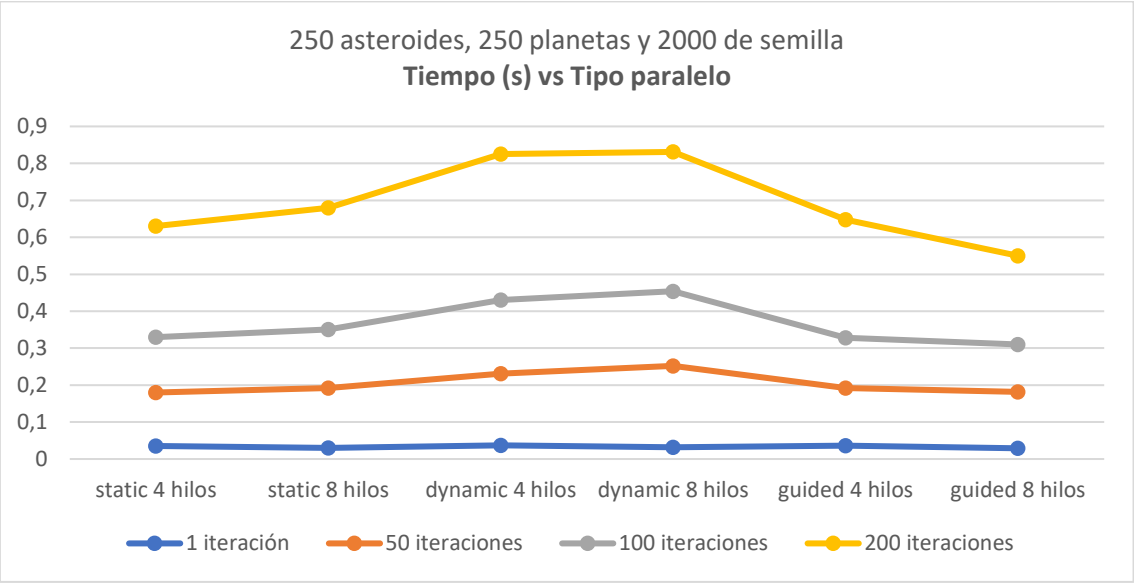
Finalmente, podemos concluir, que, dentro del paralelismo sin planificación establecida, el mejor caso es el programa paralelo con 4 hilos, debido a que el ordenador donde ejecutamos las pruebas tiene 4 cores, de tal manera que se ejecuta 1 hilo por core, por tanto, utiliza toda la CPU.

En el caso de 8 hilos y 16 hilos, aunque no sean los más eficientes, son similares al de 4 hilos debido a que son múltiplos de 4 y pueden ejecutarse simultáneamente. En el caso de 2 hilos, como no utiliza toda la CPU, sino que la mitad, tarda más tiempo en realizar la ejecución.

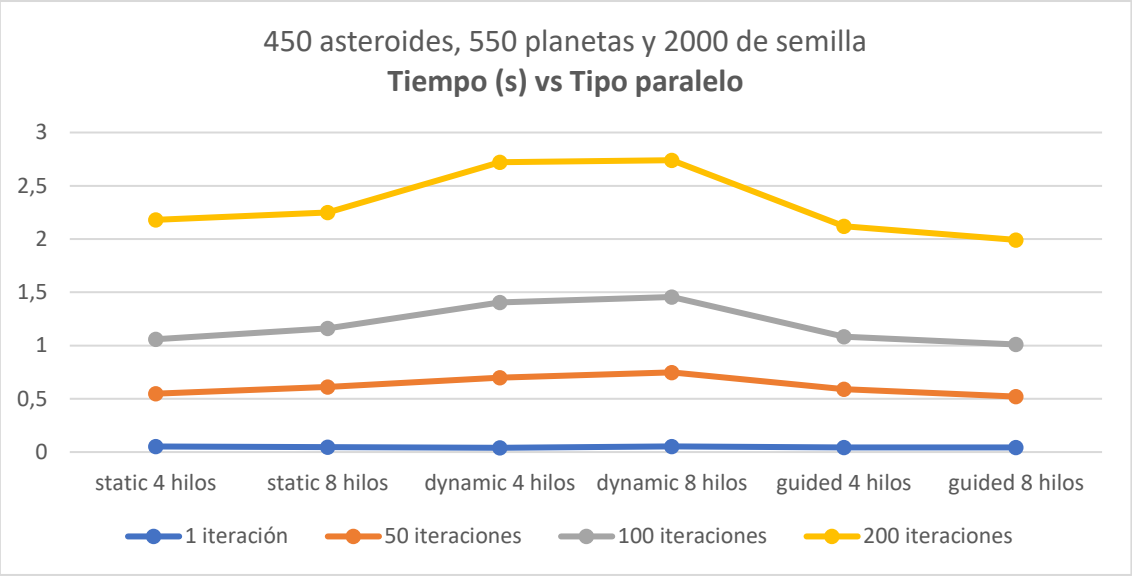
El peor caso es el paralelo de 1 hilo, que pasa como el caso de 2 hilos, ya que no utiliza toda la capacidad de la CPU disponible, por tanto, tarda más tiempo.



Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos observar que la planificación menos eficiente es la configurada con dynamic, independientemente del número de hilos. Donde mejor se aprecia esta diferencia de tiempo con respecto al resto de planificaciones es en el caso más complejo, que es el de 200 iteraciones. En el caso de 1 iteración, no hay ningún cambio significativo entre planificaciones e hilos.



Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos observar que la planificación menos eficiente es la configurada con dynamic, independientemente del número de hilos. Donde mejor se aprecia esta diferencia de tiempo con respecto al resto de planificaciones es en el caso más complejo, que es el de 200 iteraciones. En el caso de 1 iteración, no hay ningún cambio significativo entre planificaciones e hilos.



Tras haber realizado las pruebas con la configuración que se muestra en el título de estas gráficas, podemos observar que la planificación menos eficiente es la configurada con dynamic, independientemente del número de hilos. Donde mejor se aprecia esta diferencia de tiempo con respecto al resto de planificaciones es en el caso más complejo, que es el de 200 iteraciones. En el caso de 1 iteración, no hay ningún cambio significativo entre planificaciones e hilos.

Teniendo en cuenta lo analizado en las gráficas anteriores, podemos concluir que el peor caso es la planificación dynamic debido a que cada vez que se ejecuta la misma configuración de objetos, iteraciones y semilla, los tiempos de ejecución son muy similares. Por esta misma razón, la planificación static es mucho más eficiente en nuestro caso, ya que los tiempos resultantes son muy parecidos.

También hemos concluido que el caso de la planificación guided es más complejo, ya que ajusta los espacios mientras el programa está ejecutándose, de modo que cambia el tamaño cuando la carga de trabajo no es balanceada, por tanto, en cada ejecución varía mucho.

6. Conclusiones.

Con el desarrollo de esta práctica hemos conseguido familiarizarnos con la optimización de programas secuenciales y con los modelos de programación paralela en arquitecturas de memoria compartida, además del uso del lenguaje de programación C++.

En general, nos ha resultado bastante difícil la realización de la práctica debido a que nunca habíamos trabajado en C++ y hemos requerido de diversas tutorías. Además, no hemos conseguido que los resultados de nuestro programa secuencial sean exactamente iguales a los del binario proporcionado; sin embargo, estas diferencias son mínimas, variando muy pocas centésimas.

También, tuvimos problemas de tiempo a la hora de ejecutar pruebas, ya que hicimos muchos casos; y, sobre todo, a la hora de ejecutar las pruebas en el binario proporcionado, debido a que éste no está optimizado.

El mayor problema de la práctica ha sido que el código lo hemos realizado en nuestro ordenador personal, y, por tanto, íbamos comprobando que la parte paralela era más eficiente que la secuencial. Al realizar las pruebas en el ordenador de la universidad, hemos detectado que los valores no se corresponden siendo la parte secuencial más eficiente que la paralela. Finalmente, hemos decidido realizar las pruebas en otro ordenador de la universidad, obteniendo valores similares a los obtenidos en nuestros ordenadores personales y en los cuales se puede apreciar que la parte paralela es más eficiente que la secuencial.

En general, nos ha resultado complicado realizar las pruebas en los ordenadores de la universidad ya que solo teníamos una franja de tiempo limitada para trabajar y por tanto una vez fuera de esa franja no podíamos avanzar independientemente.