Universidad Carlos III de Madrid

Escuela Politécnica Superior (Leganés)

Grado en Ingeniería Informática (3er curso)

Asignatura: Arquitectura de computadores

Práctica de programación paralela con OpenMP

Docente:

- Caino Lores, Silvina

Elaborado por:

- Estévez Fernández, Andrés Javier (NIA: 358857 / Grupo 83 / 100358857@alumnos.uc3m.es)

Leganés, Madrid. 18 de noviembre de 2018.

Índice

Introducción

El presente documento sirve como memoria para la práctica de la asignatura Arquitectura de Computadores, basada en la optimización de una versión secuencial y paralela que resuelven el problema de los n-asteroides con una física ficticia. Para ello, se utilizó el lenguaje C++ (incluyendo las mejoras de C++11/14) y la herramienta que permitirá emplear las técnicas de paralelización, OpenMP.

El documento está dividido en distintas secciones: primero se desarrollan las decisiones tomadas en el diseño de la versión secuencial (órdenes de bucles, estructuras de datos utilizadas, etc.); posteriormente, se explica lo mismo para la versión paralela y optimizada con respecto a la secuencial, pues se detallan los cambios efectuados en las estructuras empleadas y bucles paralelizados. En la evaluación del rendimiento se compara mediante una serie de tablas y gráficas, los resultados obtenidos por cada una de las versiones, así como también se describen ciertas explicaciones sobre los motivos de estos rendimientos.

Luego, se exponen algunas de las pruebas realizadas para comprobar el funcionamiento del software desarrollado. En esta sección también se trata de argumentar las posibles causas de los resultados. Finalmente el documento culmina con las conclusiones personales.

Diseño de la versión secuencial

El programa secuencial comienza con la comprobación del número de argumentos esperado para que en caso de error, este sea detectado y se imprima el mensaje correspondiente. Posteriormente, se procede a generar todos los elementos (asteroides y planetas) utilizando las distribuciones aleatorias especificadas en el enunciado. Para la creación de los planetas, se utilizó el operador módulo 4 para saber, según el número de planeta en cuestión (variable de un bucle *for*), el borde que le corresponde (izquierdo, superior, derecho o inferior). Para representar los elementos se implementaron dos estructuras de datos de tipo

"*asteroide*" y "*planeta*" con una serie de atributos como *posición_x*, *posición_y*, *masa*, entre otros; y 2 vectores del tipo de dichas estructuras.

Interacción asteroide - asteroide:

La algoritmia principal se lleva a cabo en parejas de bucles anidados que a su vez se hallan dentro del bucle más externo que controla las iteraciones; resultando así una complejidad computacional de $O(n^3)$. Se tiene entonces para empezar un bucle que fija un asteroide *j* y dentro de éste otro bucle que operará sobre el resto de asteroides a partir del *k=j+1*; esto es porque la fuerza que se calcula entre el *asteroide[2]* y el *asteroide[5]* (por ejemplo) es positiva para el número 2 pero negativa para el 5, y luego no se vuelve a calcular la fuerza entre los mismo asteroides en orden inverso (entre *asteroide[5]* y *asteroide[2]*).

El cálculo de la interacción entre dos elementos engloba diversas operaciones, entre ellas y en orden de ejecución se encuentran: primero que nada se calcula la distancia entre los dos asteroides y se tienen dos posibles casos:

- La distancia es mayor a la mínima requerida (2): se procede a calcular la pendiente, el ángulo y finalmente la fuerza para cada eje. Como se mencionó anteriormente, esta fuerza se añadirá de forma positiva al sumatorio de fuerzas del asteroide *j* y de forma negativa al del asteroide *k*. Es importante mencionar que en la versión secuencial este sumatorio para cada eje es una variable que pertenece a la estructura *asteroide*; es decir, cada uno de los asteroides tiene un sumatorio para el eje *x* y uno para el eje *y*, que se actualizarán a 0 al finalizar cada iteración.

- La distancia es menor o igual a la mínima requerida (2): en lugar de calcular la fuerza entre ambos asteroides se intercambian las velocidades de la velocidad anterior de cada elemento; así, si el asteroide A colisiona con el B en la iteración *i*, la velocidad de A será la de B en la iteración *i-1* y de la misma forma B adquirirá la velocidad de A de dicha iteración. Es importante mencionar que tras una ardua tarea de depuración y aproximación a los resultados originales, se detectó un posible error en el binario proporcionado como base relacionado a la no consideración del asteroide 0 a la hora de gestionar estas colisiones; por esta razón y para acercar (aunque no igualar) los outputs del código propuesto y el base mencionado, se tomó la misma consideración.

Interacción asteroide-planeta:

Con otro bucle anidado se fija nuevamente un asteroide *z* para calcular la interacción que tiene con cada planeta *q*. A diferencia de las interacciones entre asteroides, en estas siempre se calcula la fuerza entre el planeta y el asteroide; es decir, no se toman en cuenta las colisiones entre estos elementos. Los cálculos realizados son los mismos que para obtener la fuerza entre asteroides, con la

salvedad de que dicha fuerza resultante sólo se le suma al asteroide en cuestión puesto que lo planetas son fijos.

Cálculo de valores y evaluación de rebote contra los bordes:

Dentro de un bucle se itera sobre los asteroides *j* y se obtienen los distintos valores necesarios para la actualización de sus propiedades (posición, velocidad, etc.).

Se comienza entonces calculando la aceleración de cada eje a partir del sumatorio de fuerzas correspondiente y la masa del asteroide. Luego, con la aceleración computada se calcula la nueva velocidad provisional utilizando la velocidad de la iteración anterior (si hubo una colisión entre asteroides esta velocidad será la que le otorgó el asteroide contra el que colisionó). Por último se halla la nueva posición provisional empleando la velocidad recién calculada y la posición de la iteración anterior. A continuación se evalúa para ese asteroide *j* si la nueva posición sale de alguno de los bordes y, si es el caso, se corrige y se multiplica su velocidad por -1.

Evaluación colisión entre asteroides:

Esta evaluación se realiza, como se indica en la sección “interacción asteroide – asteroide”, en ésta última pero, se detallan a continuación los algoritmos que se implementaron en un principio y que no se establecieron como definitivos ya que ninguno se acercaba más a los resultados que el planteado previamente:

Una vez calculado los nuevos valores de todos los asteroides y estudiado los rebotes contra los bordes, se estudiaba la colisión entre asteroides. Este estudio se llevaba a cabo dentro de dos *for* anidados fijando un asteroide *m* en el externo mientras que el interno iteraba sobre los *n=m+1* restantes (ya que el choque entre A y B es lo mismo que el choque entre B y A). Se obtenía la distancia entre ambos y si esta era menor a la mínima (2) se distinguía entre dos casos:

- Los asteroides no han estado colisionando desde el inicio: este es el caso “más común” y se resuelve básicamente como se gestiona en el programa final (intercambiando velocidades de la iteración anterior).

- Estos asteroides colisionaron en la primera iteración y aún no se han separado: Esta situación se detectaba al calcular la fuerza entre asteroides y, si la distancia no era mayor que la mínima, se otorgaba a la pareja un número único e identificativo que almacenarían en un atributo de la estructura; esto servía para en este punto poder identificar qué par de asteroides se encuentran en colisión desde el inicio. Se sabrá que esto es así y no se han separado porque este atributo de su estructura será diferente de 0 y además igual al del otro asteroide. Cuando se separen lo suficiente (distancia mayor a 2), el valor de la variable se pondrá a 0 y más nunca se alterará.

Ahora bien, el motivo por el que se gestionaba este segundo caso de esta manera es porque si se siguiera el algoritmo del caso anterior, dos asteroides que colisionen en la primera iteración intercambiarían velocidades nulas entre sí, si en la siguiente iteración siguen en choque volverán a intercambiar velocidades iguales a 0, y así sucesivamente.

Otra posible alternativa que también traía problemas es que dos asteroides que colisionen, si es la primera iteración, intercambien las velocidades de la iteración actual. El inconveniente con esto surge en que si en las próximas iteraciones siguen en choque estarán intercambiándose la misma velocidad calculada en la primera iteración por seguir el algoritmo del primer caso.

Finalmente se creía haber llegado a la solución de este escenario: dos asteroides que hayan colisionado desde la primera iteración y aún no se hayan separado se intercambiarían la velocidad calculada en la iteración actual.

Tal y como se implementó esto no otorgaba resultados muy cercanos a los originales, y es que si por ejemplo un mismo asteroide colisiona en la primera iteración con más de un asteroide, el número identificativo que tendrá será el correspondiente a su última colisión y la detección de choques previos se habrá perdido.

Todos estos son escenarios que no vienen explícitamente contemplados en el enunciado en cuanto a resolución se refiere, por una cuestión de acercarse lo más posible a los resultados base, se gestionaron como se explica en la sección “interacción asteroide – asteroide”.

Actualización final de los valores de la iteración actual:

Para finalizar la iteración se copian los valores de las posiciones y velocidades que se tenían como provisionales a las definitivas y además, el sumatorio de fuerzas en cada eje y de cada asteroide se iguala a 0.

Diseño de la versión paralela

La forma en la que comienza la versión paralela no difiere con respecto a la secuencia. Pues se realiza la comprobación de errores en el número de argumentos y se declaran variables auxiliares y estructuras de datos que no se tenían en la versión secuencial y aquí son necesarios para optimizar la paralelización. Estas estructuras se detallarán a medida que se introduzcan en las secciones posteriores.

Interacción asteroide – asteroide:

Al igual que en la versión secuencial, esta parte del programa se lleva a cabo dentro de un bucle anidado. El bucle que se ha paralelizado es el exterior que itera sobre *j* para que de esta manera los distintos hilos no coincidan en este índice (es

decir, no calculen las interacciones para con el mismo asteroide fijado). Donde podrán coincidir es en el índice *k=j+1* (del bule interior) por más que se haya establecido dicha variable como privada a través de una cláusula, puesto que es posible que existan dos hilos A y B que estén evaluando la interacción entre el par de asteroides *(a,c)* y *(b,c)* respectivamente.

Lo más importante de esta interacción recae en la forma en la que se almacenan los sumatorios y registran las colisiones encontradas. En un principio se había planteado de la siguiente manera: se tenían dos vectores (uno para los sumatorios en el eje *x* y otro para el *y*) de tamaño del número de asteroides y, la idea se basaba en establecer en el *#pragma* del bucle paralelizado mediante una cláusula, que estos vectores fueran privados para cada hilo de manera que cada uno tuviera una copia del vector y escribieran sobre ellos sin condiciones de carrera. Al final de los bucles se establecía una sección crítica para que antes de abandonar las regiones paralelas cada hilo, y uno por uno, volcará los sumatorios calculados y almacenados en su copia del vector al vector original que se mantendría en la posteridad del programa. Por alguna extraña razón y sin poder ser detectada con precisión, se generaban condiciones de carrera puesto que los resultados obtenidos entre una serie de pruebas con los mismos argumentos eran diferentes entre sí.

Con el fin de subsanar este problema que alteraba los resultados en comparación con los arrojados por la versión secuencial, se pensó en cambiar la estructura de datos en la que se acumulaban los sumatorios para los asteroides. Se llegó entonces a la conclusión de que una matriz era la estructura que cumplía con los requisitos del problema y fue lo que se implementó. Se buscaba una estructura que permitiera, sabiendo con seguridad que los hilos no coincidirían en al menos uno de los índices (el del bucle más externo y el que fue paralelizado), identificar una tupla de dos índices de manera única como si de una clave primaria se tratara, y es justamente lo que un vector bidimensional ofrece. Con esto, aunque se diera el ejemplo utilizado anteriormente en el que dos hilos evalúan los pares *(a,c)* y *(b,c)* respectivamente, pese a que el segundo índice es el mismo nunca se accederá a la misma posición de la matriz; pues, se almacena en la posición *matriz[j][k]* la fuerza (positiva) entre el asteroide *j* y el asteroide *k*, y con signo negativo en la posición *matriz[k][j]*.

En cuanto a la gestión de colisiones dentro del bucle anidado, estaba claro que no podía realizarse de la misma forma que la versión secuencial debido a que el orden en el intercambio de velocidades importa, y por más que ese trozo de código se encapsulara dentro de una sección crítica, lo que único que se conseguiría es la exclusión mutua más no se garantizaría el orden en el que los hilos acceden a ella y alteran las velocidades.

Para tratar esto y después de comprobar la efectividad de las matrices en el problema anterior, se decidió utilizar una estructura de datos del mismo tipo para solventarlo. Así, con la matriz previamente inicializada a 0 se almacena en la

posición *colisiones[j][k]* un 1 si existe una colisión entre dichos asteroides. Es importante destacar que con esto únicamente se detecta la colisión, más no se gestiona (no se intercambian las velocidades). El intercambio de velocidades es algo que debe realizarse necesariamente en orden secuencial puesto que es este orden el que determinará la obtención de unos resultados u otros, y es por ello que al finalizar el bucle anidado se tiene otro (no paralelizado) que recorre la matriz *colisiones* e intercambia las velocidades de los asteroides *j* y *k* si en dicha posición hay un 1.

También, se tiene un bucle anidado para volcar los sumatorios de las matrices en los vectores "originales" que se utilizan durante el resto del programa para hacer los cálculos. Se vuelca entonces el sumatorio de la fila *i* a la posición *i* del vector, y el valor obtenido representa el sumatorio de las fuerzas entre el asteroide *i* con el resto de asteroides (que estén a una distancia mayor a 2). Este último bucle anidado si está paralelizado ya que cada hilo tiene su propia copia del índice *i* y al acceder a la matriz (fila *i* columna *j*) no habrá condiciones de carrera; además, como los hilos no coinciden con seguridad en el índice *i*, tampoco la habrá al acceder a la posición *i* del vector.

En la eterna discusión relacionada con la paralelización de un conjunto de instrucciones, es habitual que se acabe sacrificando una de las dos principales características de las que debe estar dotado un programa: fiabilidad (precisión y calidad en los resultados) y tiempo de ejecución. En la versión paralela entregada estos dos estándares se mantienen; es decir, los resultados coinciden con los de la versión secuencial y se obtiene un tiempo de ejecución notable (se detallará más adelante en la evaluación de las versiones y speedup conseguido) pero, otra propiedad de un programa que quizás a priori no se tiene tanto en cuenta a la hora de paralelizar se ve sacrificada: la memoria consumida por el mismo.

Con las 3 matrices utilizadas para lograr la paralelización (una para almacenar los sumatorios en el eje *x*, otra para el eje *y*, y otra para marcar las colisiones), la memoria reservada dependerá cuadráticamente del número de asteroides con el que se ejecute el programa. Si cada matriz es cuadrada de dimensión $num\_asteroides_x num\_asteroides$ se tendrán $num\_asteroides^2$ posiciones de doble precisión (8 Bytes) en cada una; es decir, la memoria ocupada por estas 3 matrices será → $memoria\_ocupada = 3 * \left(\frac{num\_asteroides^2 * 8}{2^{20}}\right) MB$. Con una ejecución de 1000 asteroides, por ejemplo, la memoria ocupada ascendería a los 22,88 MB. Por último, así como se reserva esta cantidad de espacio en la memoria, también se libera al finalizar el programa.

Interacción asteroide – planeta:

Se evalúa esta interacción igual que en la versión secuencial, con la diferencia de que el bucle más externo se encuentra paralelizado.

Cálculo de valores y evaluación de rebote contra los bordes:

Se lleva a cabo de la misma forma que en la versión secuencia pero con el bucle paralelizado. Como sólo se acceden a atributos ya actualizados de asteroides, los hilos accederán a atributos distintos ya que no habrá más de un hilo con el mismo índice del bucle. Así mismo, dentro del bucle se valoran los rebotes contra los bordes.

Actualización final de los valores de la iteración actual:

Antes de empezar una nueva iteración se copian los valores de las posiciones y velocidades que se tenían como provisionales a las definitivas, las matrices de sumatorios se igual a 0 y lo propio para la matriz de colisiones.

Evaluación de rendimiento y comparación de resultados

Todas las pruebas de rendimiento se llevaron a cabo en los ordenadores de las aulas del departamento de informática de la UC3M en Leganés, cuyos parámetros más relevantes son los siguientes:

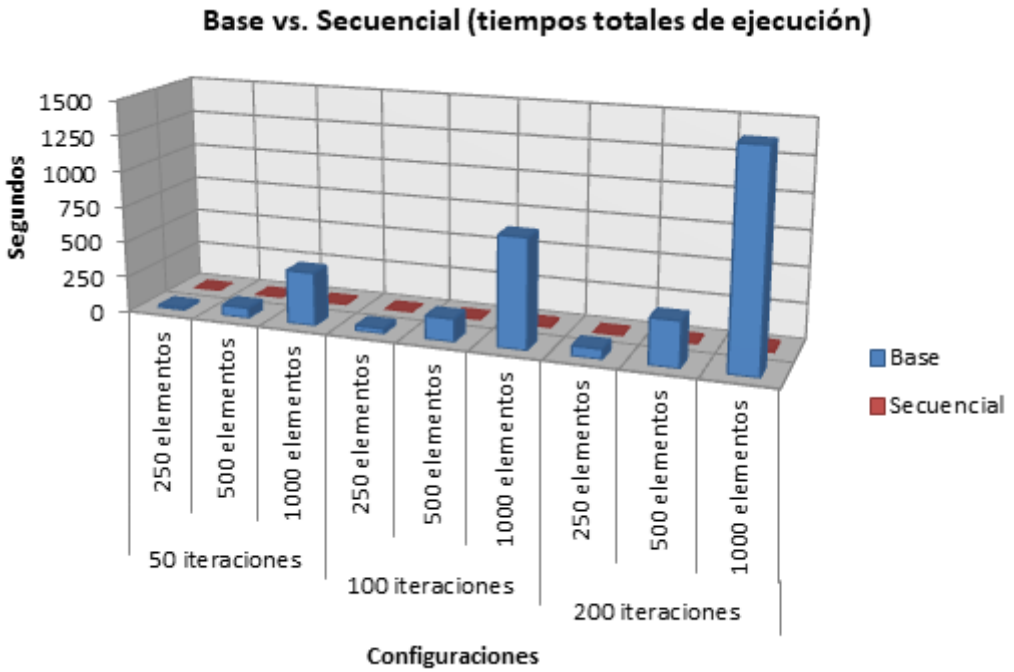
- Modelo del procesador: Intel® Core™ i5-4460 CPU @ 3.20GHz × 4
- Número de cores: 4
- Tamaño de memoria principal: 15.6GB
- Jerarquía de la memoria caché: 6MB SmartCache (caché compartida por todos los cores del procesador y que normalmente es de 2 o 3 niveles)
- Versión del sistema operativo: Debian GNU/Linux 9 (stretch) 64-bit
- Versión del compilador: gcc versión 6.3.0

En la siguiente tabla se muestran los distintos tiempos de ejecución (medidos en segundos) del programa base y el programa secuencial que servirán como punto de partida para una primera comparación. Por cada número de iteración y elementos se tomó el valor medio del obtenido en 10 ejecuciones, y todas en las pruebas se utilizó la misma semilla (10). Un número de elementos de "*x*" indica que tanto para los asteroides como para los plantes habrá esa cantidad.

		Programa base (segundos)	**Programa secuencial (segundos)**
50 iteraciones	250 elementos	17.787	0.387
	500 elementos	72.155	1.284
	1000 elementos	373.777	4.634
100 iteraciones	250 elementos	36.044	0.667
	500 elementos	160.354	2.422
	1000 elementos	763.352	8.874
200 iteraciones	250 elementos	70.604	1.375
	500 elementos	317.045	4.703
	1000 elementos	1475.219	17.546

Tabla 1 – Tiempos totales de ejecución (Base vs. Secuencial)

La información mostrada en la tabla anterior se observa de manera más clara en la siguiente figura:

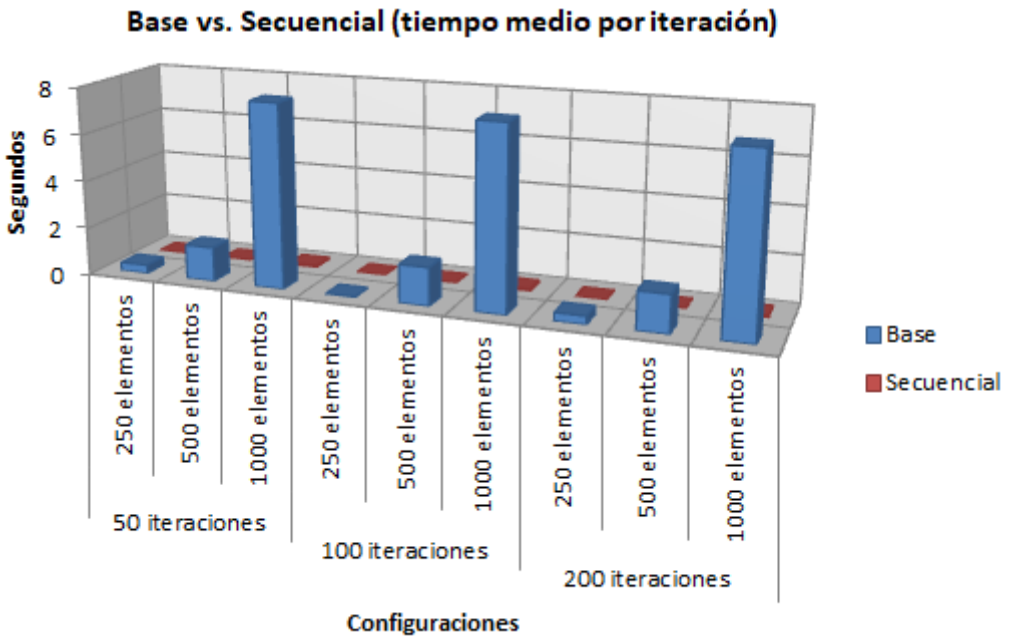


Es posible apreciar una considerable diferencia entre ambos programas en cuanto a tiempo total de ejecución se refiere por cada pareja de configuración (iteraciones, elementos). Como dato peculiar, se tiene que para la más exigente configuración (200 iteraciones, 1000 elementos), el tiempo que tarda en ejecutar la versión secuencial es incluso menor a lo que tarda el programa base para la configuración menos exigente (50 iteraciones, 250 elementos). Además, se observa que para cada programa y la misma cantidad de elementos, el tiempo se duplica (aproximadamente) con las diferentes iteraciones (algo que era de esperarse ya que las iteraciones también se van duplicando).

Algo que hay que tener en cuenta y afecta a los tiempos obtenidos con el programa original, es la escritura en el fichero *step_by_step.*txt. Si se analiza el tiempo medio por iteración se tiene que:

		Programa base (tiempo medio por iteración en segundos)	**Programa secuencial (tiempo medio por iteración en segundos)**
50 iteraciones	250 elementos	0.356	0.008
	500 elementos	1.443	0.026
	1000 elementos	7.476	0.093
100 iteraciones	250 elementos	0.360	0.007
	500 elementos	1.604	0.024
	1000 elementos	7.634	0.089
200 iteraciones	250 elementos	0.353	0.007
	500 elementos	1.585	0.024
	1000 elementos	7.376	0.088

Tabla 2 – Tiempo medio por iteración (Base vs. Secuencial)



En esta evaluación la cantidad de iteraciones no influye en el valor medio puesto que lo que afecta directamente al mismo es el número de elementos que se tienen que procesar en dicha iteración. Es por ello que si se observa con detenimiento es posible localizar aproximadamente el mismo tiempo medio de un mismo programa para una misma cantidad de elementos (independientemente del número de iteraciones), por lo que en este estudio sí se puede establecer una relación entre ambos programas: en el caso de 250 elementos, la versión secuencial tiene un tiempo medio por iteración del 2,25% del tiempo de la versión original; con 500 elementos equivale a un 1,8%; y con 1000 elementos a un 1,2%.

Una vez estudiado la mejora (en tiempo) que existe entre la versión original y la secuencial, es momento de hacer lo propio entre esta última y la paralela. A continuación se detallan en forma de tabla los tiempo totales de ejecución y medios por iteración para las mismas parejas de configuraciones utilizadas hasta el momento y además, empleando un número distinto de hilos.

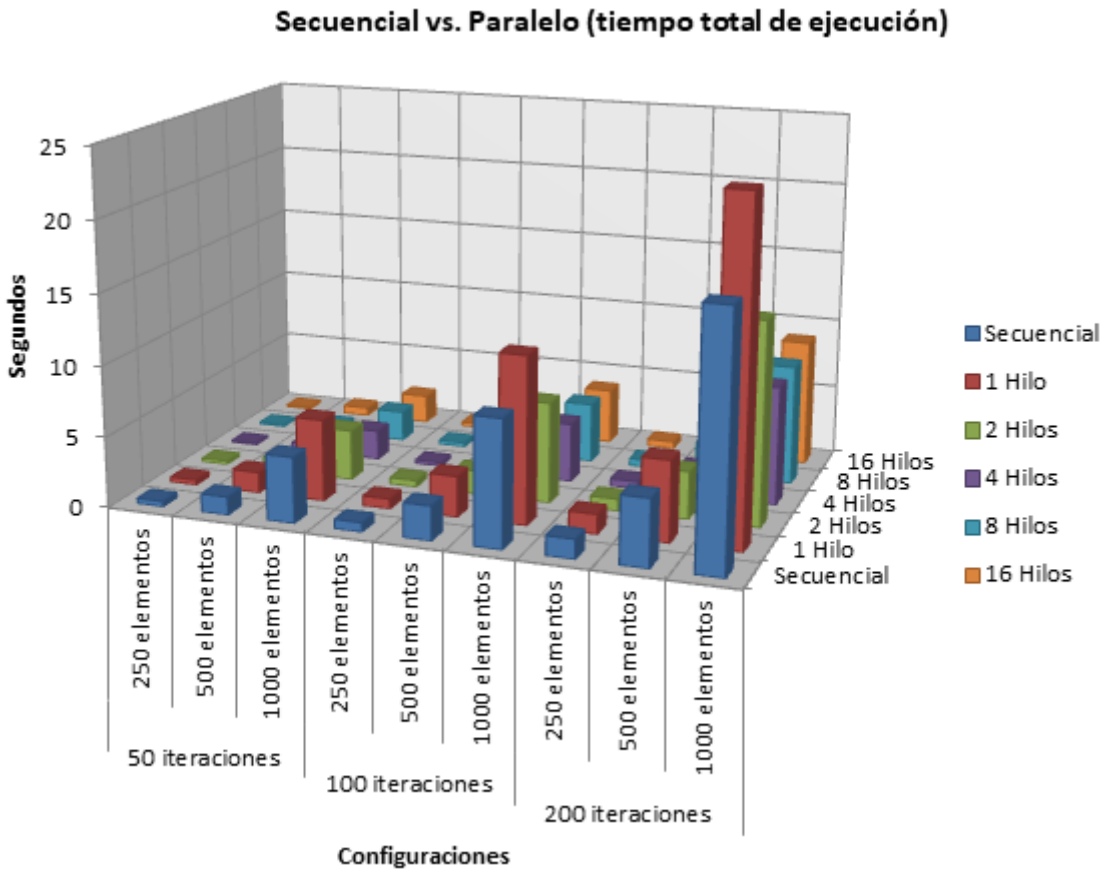
			Programa paralelo (segundos)				
Iteraciones	**Elementos**	**Programa secuencial (segundos)**	**1 Hilo (seg)**	**2 Hilos (seg)**	**4 Hilos (seg)**	**8 Hilos (seg)**	**16 Hilos (seg)**
50	250	0.387	0.368	0.255	0.143	0.159	0.148
	500	1.284	1.479	0.862	0.511	0.626	0.581
	1000	4.634	5.770	3.577	2.110	2.153	2.024
100	250	0.667	0.710	0.463	0.264	0.282	0.266
	500	2.422	2.884	1.689	0.989	1.214	1.080
	1000	8.874	11.750	7.115	4.190	4.243	3.932
200	250	1.375	1.401	0.845	0.521	0.511	0.501
	500	4.703	5.678	3.360	1.987	2.307	2.141
	1000	17.546	23.651	14.208	8.338	8.452	8.946

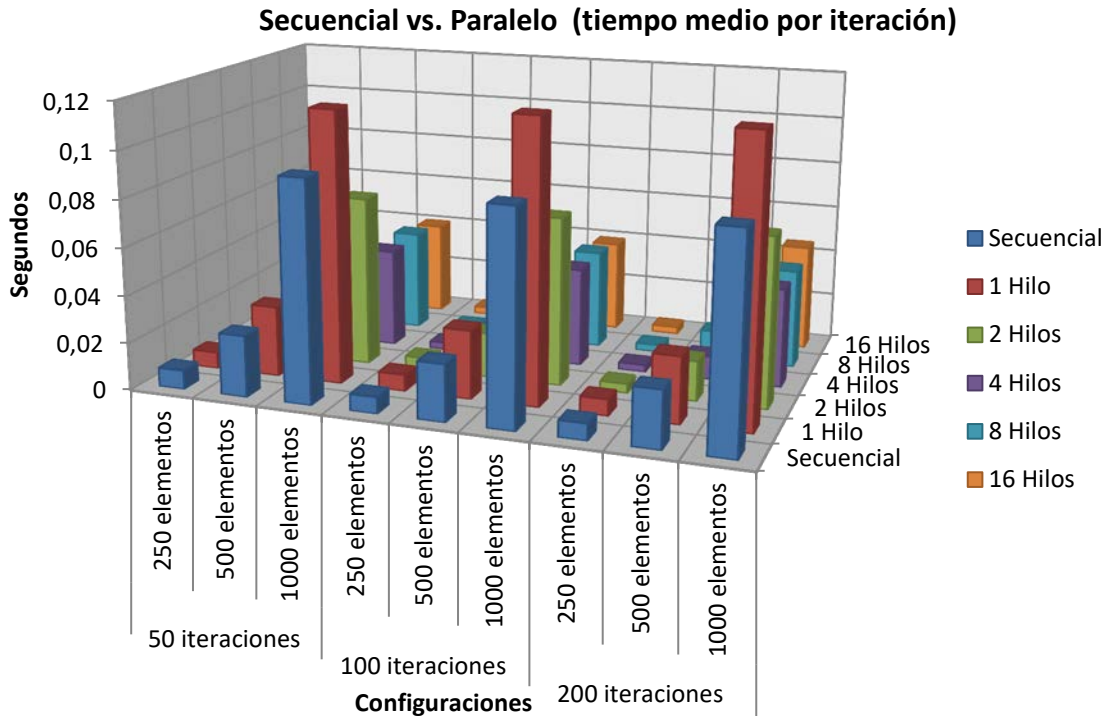
Tabla 3 – Tiempo total de ejecución (Secuencial vs. Paralelo)

			Programa paralelo (segundos)				
Iteraciones	Elementos	Programa secuencial (tiempo medio por iteración en segundos)	1 Hilo (tiempo medio por iteración en seg)	2 Hilos (tiempo medio por iteración en seg)	4 Hilos (tiempo medio por iteración en seg)	8 Hilos (tiempo medio por iteración en seg)	16 Hilos (tiempo medio por iteración en seg)
50	250	0.008	0.007	0.005	0.003	0.003	0.003
	500	0.026	0.030	0.017	0.010	0.013	0.012
	1000	0.093	0.115	0.072	0.042	0.043	0.040
100	250	0.007	0.007	0.005	0.003	0.003	0.003
	500	0.024	0.029	0.019	0.010	0.012	0.012
	1000	0.089	0.118	0.071	0.042	0.042	0.039
200	250	0.007	0.007	0.004	0.003	0.003	0.003
	500	0.024	0.028	0.017	0.010	0.012	0.012
	1000	0.088	0.118	0.071	0.042	0.042	0.045

Tabla 4 – Tiempo medio por iteración (Secuencial v. Paralelo)

Gráficamente:



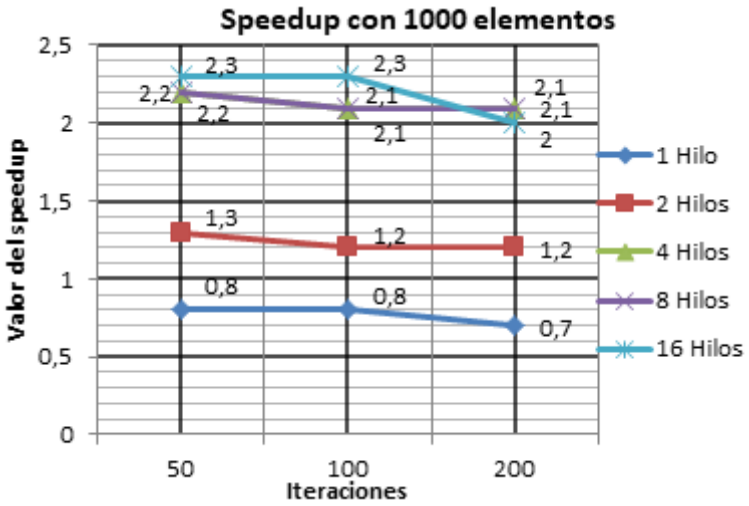


A través de la gráfica que compara los tiempos totales de ejecución de la versión secuencial y paralela según el número de hilos, se pueden establecer ciertas conclusiones. Primero que nada, se observa que en todos los casos la ejecución paralela con 1 hilo (color rojo) tarda más que la secuencial, y esto es porque al estar ejecutando con un hilo realmente es como si el hilo maestro ejecuta todo el programa como ocurre en secuencial, y además porque los programas paralelos y secuenciales no son idénticos, recordemos que en el diseño de ambas versiones se explicó que en la versión paralela se introdujeron matrices y mayor número de bucles anidados para garantizar la correcta paralelización y exclusión mutua, siendo entonces esto lo que provoca este resultado.

Por otro lado, como el procesador del ordenador en el que se realizaron las pruebas tiene 4 núcleos, era de esperarse que los rendimientos más óptimos se consiguieran con 4 hilos (color morado); pues se observa que para todas las parejas de configuraciones es esta barra la de menor tamaño en la gráfica.

Con respecto a la ejecución con 8 y 16 hilos, se aprecia que ambos valores son similares entre sí en todos los casos y a su vez ligeramente mayores que con 4 hilos. Esto se puede deber a que como cada núcleo debe ejecutar más de un hilo, se producen cambios de contexto constantemente en un mismo core al cargar las variables privadas y demás recursos de cada thread, derivando en una ralentización de la ejecución.

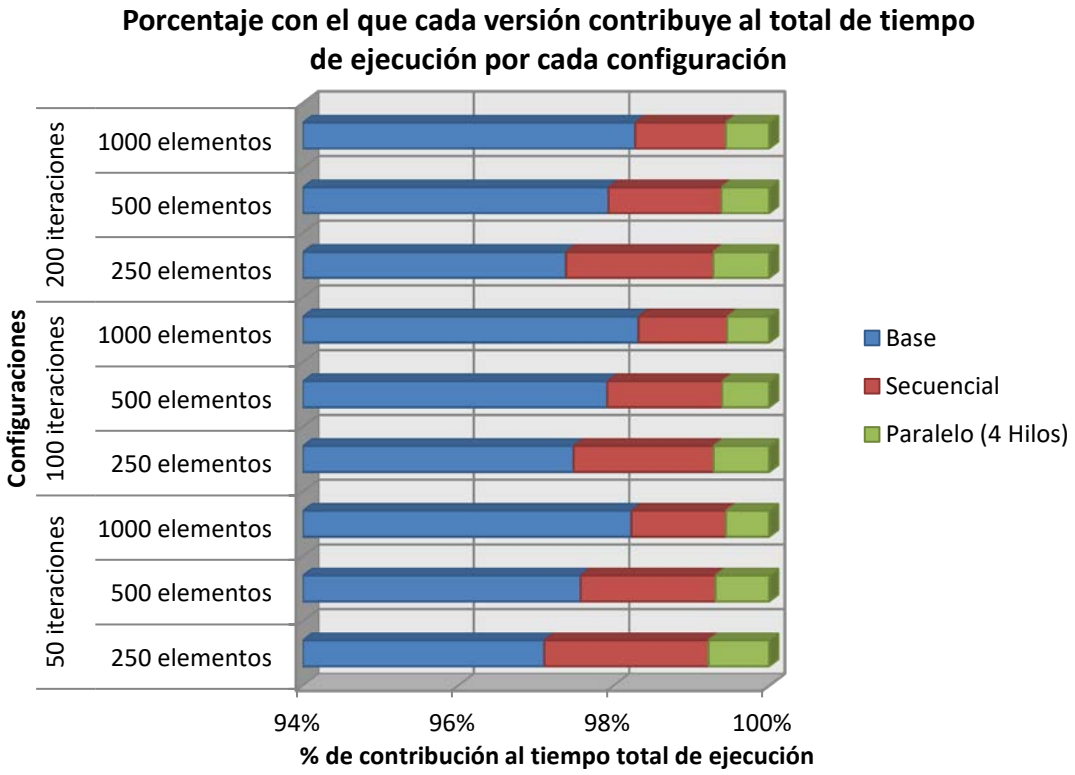
En la siguiente gráfica se representa el speedup que alcanza una ejecución según el número de hilos, para 1000 elementos, y con las variaciones en las iteraciones (50, 100 y 200):



Todos los speedups conseguidos están por encima de la unidad exceptuando el producido por la ejecución con 1 hilo; y es que como se explicó anteriormente, el programa paralelo al tener más estructuras de datos que recorrer y más bucles anidados que efectuar, termina teniendo más instrucciones y más accesos a memoria que la versión secuencial, por lo tanto se esperaba este speedup para el caso de 1 hilo.

Otro hecho que contribuye con la efectiva paralelización llevada a cabo es la arquitectura del computador utilizado. Este computador tiene una memoria caché de 6 MB SmartCache, lo que quiere decir que es una memoria compartida por todos los núcleos y no dedicada para cada uno de ellos. Con esto se obtiene un ratio global de fallos menor cuando no todos los núcleos necesitan el mismo bloque de caché y además un núcleo puede utilizar completamente el nivel 2 o 3 de la memoria cuando los demás cores están inactivos.

Para finalizar con la comparación entre los dos programas entregados, se muestra la siguiente gráfica de a continuación que sirve para ver claramente cómo se comporta cada versión según la configuración y sobre un total de tiempo de ejecución del 100%:



Enfocando el análisis en los distintos modos de planificación que ofrece la herramienta OpenMP, se estudian brevemente los resultados arrojados por una ejecución paralela con 4 y 8 hilos, variando el modo de planificación en todos los bucles paralelizados. La siguiente tabla muestra dichos tiempos:

		Paralelo con 4 hilos (segundos)			Paralelo con 8 hilos (segundos)		
Iteraciones	**Elementos**	**Static**	**Dynamic**	**Guided**	**Static**	**Dynamic**	**Guided**
50	250	0.163	0.201	0.177	0.166	0.215	0.143
	500	0.534	0.733	0.546	0.617	0.741	0.477
	1000	2.139	2.892	2.160	2.167	2.928	1.915
100	250	0.327	0.379	0.288	0.303	0.384	0.255
	500	1.007	1.439	1.010	1.176	1.439	0.891
	1000	4.208	5.763	5.730	4.278	5.733	3.987
200	250	0.530	0.729	0.527	0.549	0.842	0.464
	500	1.952	2.836	1.979	2.305	2.831	1.756
	1000	8.349	11.429	8.420	8.546	11.429	7.510

Tabla 6 – Tiempos totales de ejecución según el modo de planificación con 4 y 8 hilos

Las distintas ejecuciones se ejecutaron sin especificar el tamaño, con lo que cada hilo ejecutaría en un principio $\frac{num\_asteroides}{num\_hilos}$ iteraciones. Se observa que los tiempos con el modo *static* son prácticamente iguales a los tiempos sin ningún tipo de planificación analizados previamente. Esto sucede ya que si no se especifica un tamaño a la cláusula, la ejecución sería equivalente a no tener ninguna cláusula (se hace una partición del número de iteraciones por el número de hilos disponibles).

En cuanto a los otros dos modos, *dynamic* y *guided* tienen una característica común y es que bajo ambas planificaciones no se asignan bloques fijos de

iteraciones a cada hilo (como sí ocurre con *static*), lo que permite que a medida que estos vayan culminando con su bloque, tomen otro. La diferencia entre estos dos está en el tamaño del bloque que toman si terminan con el inicial; para *dynamic* este tamaño es el especificado en la cláusula y para *guided*, el bloque se irá reduciendo hasta llegar también al tamaño especificado. Ninguno de estos dos últimos modos conlleva a un tiempo de ejecución menor que *static* puesto que con ellos puede darse el caso de que un núcleo ejecute muchas menos iteraciones que otro o simplemente permanezca inactivo según sea la velocidad con la que los otro núcleos ejecuten su bloque, provocando que no se aproveche al 100% y de forma "equitativa" la arquitectura del computador y la paralelización no se aprovecharía al máximo.

Pruebas realizadas

Input	Output
Número incorrecto de argumentos (en ambas versiones)	Mensaje de error correspondiente
Comparación entre los ficheros iniciales del binario base y de las versiones entregadas	Ficheros *init_conf.txt* iguales
Ejecución de las 3 versiones con los argumentos 100 100 100 10	Los 3 ficheros *out.txt* iguales
Ejecución de las 3 versiones con los argumentos 250 100 250 10	Los ficheros *out.txt* del secuencial y paralelo coinciden entre sí pero difieren del original en 26 líneas
Ejecución de las 3 versiones con los argumentos 500 200 500 10	Los ficheros *out.txt* del secuencial y paralelo coinciden entre sí pero difieren del original en 427 líneas

Además de estas pruebas se realizaron muchas más que no se especifican individualmente por lo siguiente: los ficheros de salida generados por el programa secuencial y paralelo siempre coinciden, las diferencias con el fichero que produce el binario base se generan cuando ocurren colisiones, lo cual sucede conforme se aumenta la cantidad de elementos y/o iteraciones. Por ellos se especifica únicamente un caso de ejemplo en el que los ficheros coinciden y otros dos en los que no. Por redundancia tampoco se han especificado todas las pruebas con las que se evaluó el rendimiento y se elaboraron las gráficas (ya que son pruebas de medición de tiempo).

No es posible establecer una relación fija porcentual del número de líneas distintas ya que un error o cálculo distinto relacionado a un asteroide, variará sus atributos y hará que este "error" se propague hacia todos los demás. Cabe destacar que si se ejecuta el comando *diff* entre los ficheros de salida, estas diferencias son del orden de milésimas y por dígitos normalmente iguales a la unidad.

Las diferencias que se producen en la salida puede deberse a distintos factores que quizás no se encuentran especificados explícitamente en el enunciado, como el orden en el que se resuelven las colisiones, la gestión de estas en la iteración 0, entre otros. A pesar de ello se hizo un máximo esfuerzo para llegar a replicar el output original, pero hay que entender que lograr reproducir un mismo algoritmo sin conocer los detalles en cuanto a orden de operaciones y demás, es una labor muy complicada y que al final no tiene mucho sentido o no desemboca en un aprendizaje directamente relacionado con la asignatura.

Conclusiones

OpenMP es una herramienta relativamente sencilla de utilizar si se tiene cuidado al establecer los *#pragmas* y teniendo un conocimiento previo sobre la concurrencia entre varios threads y los escenarios que se pueden dar. Al mismo tiempo es un muy potente instrumento que permite aprovechar todos los núcleos de un procesador y optimizar los tiempos de ejecución de un programa.

A modo personal, se dedicaron probablemente más de 30 horas reales de trabajo en la práctica, y aproximadamente un 70% habrá sido destinado al intento de obtención de la misma salida que arroja la versión original. Me encuentro satisfecho con lo que aprendí sobre la paralelización y sobre cómo se puede utilizar OpenMP.