







Índice

[**0. Introducción**](#_xpstqyo51v97) **2**

[**1. Evaluación del rendimiento secuencial**](#_e77bb2y2rb1x) **2**

[1.1 Introducción](#_kxinyjtsbpvx) 2

[1.2 Justificación de las decisiones de diseño](#_e0ovh7u1ghrl) 2

[1.3 Gráficas](#_qwk51uwheirh) 2

[1.4 Conclusiones parte secuencial](#_87anlduow166) 4

[**2. Evaluación del rendimiento paralelo**](#_6ljlnpgdxlkv) **4**

[2.1 Introducción](#_3plc5etl0mzo) 4

[2.2 Justificación de las decisiones de diseño](#_inw70bc23met) 5

[2.3 Gráficas](#_a4ia3pbcbshd) 5

[2.4 Impacto de la planificación (static, dynamic, guided) sobre 4 y 8 hilos](#_9fcamgskz8tq) 13

[2.4.1 Introducción](#_62rbjys0qn5v) 13

[2.4.2 Impacto de la planificación sobre 4 hilos](#_bt8b36zf9xcv) 14

[2.4.3 Impacto de la planificación sobre 8 hilos](#_mebuh4rtz5uh) 14

[2.5 Conclusiones de la parte paralela](#_rvtphr93e76b) 14

[**3. Evaluación de rendimiento**](#_ofxjt8vyfwhl) **14**

[**4. Pruebas realizadas**](#_1p7giu611y69) **15**

[**5. Conclusiones**](#_kamcfmbj5y7o) **15**

# 

# 0. Introducción

Está práctica tiene como objetivo que los estudiantes observen las diferencias entre la programación secuencial y la programación paralela, analizando así sus diferencias y las ventajas e inconvenientes de cada una de ellas.

El lenguaje de programación que se va a utilizar en esta práctica es C++ y, además, se utilizará OpenMP, una API que permite la programación multiproceso en memoria compartida.

Se nos pide simular el “juego de los asteroides” donde en un espacio bidimensional se encuentran asteroides que se van desplazando a lo largo del tiempo debido a las fuerzas de los planetas y a la fuerzas que ejercen entre ellos. Además, hay un rayo láser que se dispara en cada instante de tiempo, destruyendo aquellos asteroides que estén en su camino.

# 1. Evaluación del rendimiento secuencial

En este primer apartado se presentan las gráficas de los tiempos de ejecución de la versión secuencial. Seguidamente se recogen las conclusiones que se pueden inferir de dichos resultados.

## 1.1 Introducción

Para la realización de las gráficas hemos usado la herramienta de diseño integrada en Excel. Cada combinación posible entre los diferentes tamaños de población e iteraciones vienen escritas en las propias gráficas para facilitar su comprensión. Los tiempos de ejecución mostrados están expresados en microsegundos.

## 1.2 Justificación de las decisiones de diseño

En el código se utilizan vectores (vector), hemos estimado que era la forma más simple de asignar memoria de forma dinámica para los “arrays” tanto de planetas como de asteroides. El resto del código sigue en orden las operaciones del enunciado utilizando la librería estándar de matemáticas de c++.

## 1.3 Gráficas

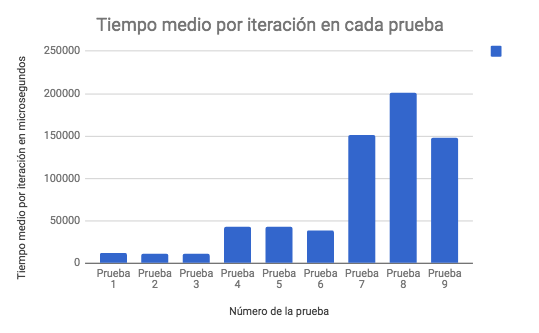












## 1.4 Conclusiones parte secuencial

La primera conclusión que se extrae es que hay un incremento del tiempo de ejecución a medida que se aumenta el número de iteraciones. Es decir, hay una correlación positiva entre el número de iteraciones y el tiempo de ejecución. Con un breve análisis del código secuencial presentado se puede ver que la realización de las iteraciones tiene una complejidad lineal, esto explica los grandes incrementos de tiempo se producen con el número de asteroides o planetas.

En la realización de las iteraciones se fuerza a un elevado número de accesos a memoria, se solicitan de forma constante elementos de vectores de planetas y asteroides lo que implica que nuestro programa se ejecute con una mayor lentitud.

Se puede entender como con el aumento de los cuerpos en nuestra ejecución se incrementa el número de accesos a memoria necesarios para la ejecución, nuestra memoria caché no puede almacenar la totalidad de los datos y esto supone mayor número de fallos y accesos a memoria lo que también aumenta el tiempo de ejecución (la complejidad exponencial del código ayuda a este efecto).

# 2. Evaluación del rendimiento paralelo

En este segundo apartado se presentan las gráficas de los tiempos de ejecución de la versión paralela, siguiendo las mismas pautas que en el secuencial pero variando el número de hilos (1,2,4,8,16). Seguidamente se recogen las conclusiones que se pueden inferir de dichos resultados.

## *2.1* *Introducción*

Para el diseño de la parte paralelizable se ha usado OpenMP, una API para la programación multiproceso de memoria compartida. Para el diseño de las gráficas se ha usado la herramienta de diseño integrada en Excel. Los tiempos de ejecución están medidos en microsegundos.

## 2.2 Justificación de las decisiones de diseño

Este apartado pretende detallar y justificar las modificaciones realizadas sobre el código secuencial.

La modificación más sustancial ha consistido en paralelizar los bucles que colocan los asteroides y los planetas, así como los bucles de las iteraciones y cálculo de las nuevas posiciones de los asteroides.

Se ha decidido implementar las estructuras del tipo *pragma omp parallel for schedule* al inicio de cada bucle para la colocación, esto se debe a que xdist, ydist y mdist tienen que ejecutarse en el mismo orden que el código secuencial para obtener los mismos resultados, lo que permite que los hilos se repartan el total de iteraciones. Además en ciertas regiones del código como la asignación de una posición *x*, posición *y* así como la masa *m*, se incluye *ordered* para que esas instrucciones se ejecuten en orden secuencial y no haya problemas de malas asignaciones o errores cuando las funciones se pasen parámetros.

En el caso de los bucles de las iteraciones las decisiones tomadas son distintas, hemos colapsado los bucles anidados, después de ejecutarlos del mismo modo que los de colocación y colapsados resultó ser ligeramente más rápida esta segunda opción. Cuando se calculan las distintas velocidades hemos optado por la misma operación ordenada que en los de colocación aunque en estos casos no resulta relevante el orden no tenía un fuerte impacto en los tiempos de ejecución. Por último el borrado de asteroides es una sección crítica puesto que se borra en orden.

La otra sección crítica que se ha localizado en el código es el cálculo de las aceleraciones para los planetas, la posibilidad de que varios hilos accedan de manera simultánea a la aceleración de un asteroide puede producir fallos.

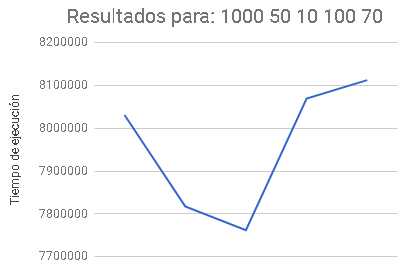
## 2.3 Gráficas

*Con 1 hilo:*

**

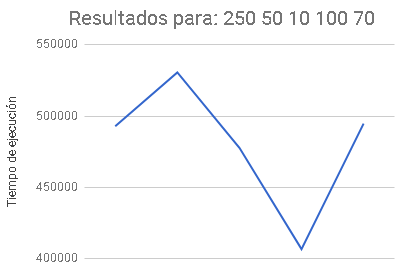
**

**

**

**

*Con 2 hilos:*

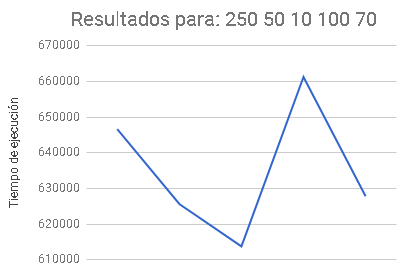
**

**

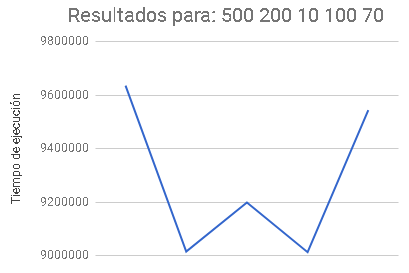
**

**

*Con 4 hilos:*

**

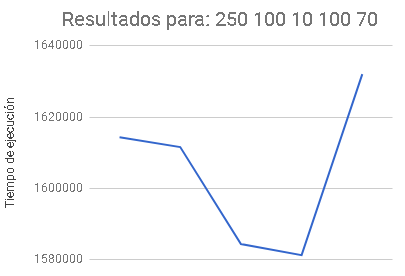


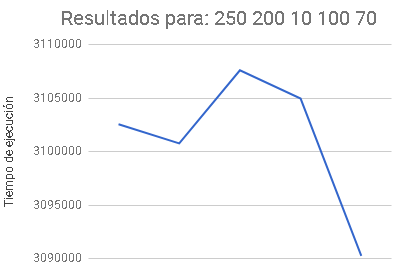


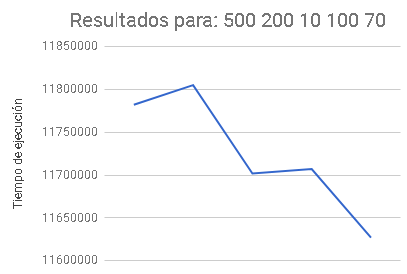


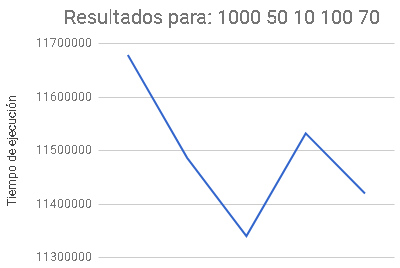


*Con 8 hilos:*

**

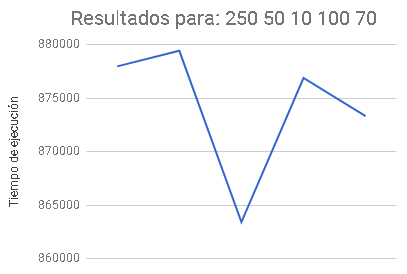
**

**

**

**

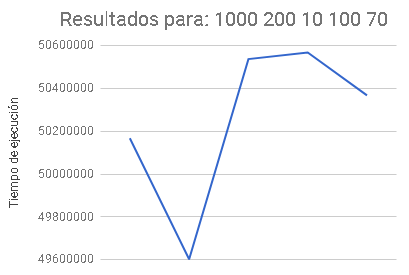
*Con 16 hilos:*

**

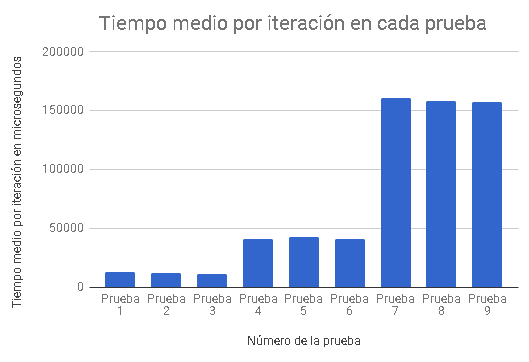
**

**

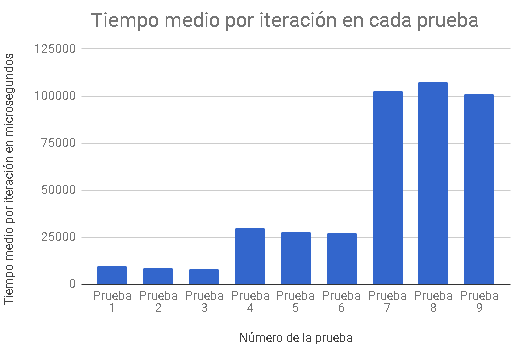
**

**

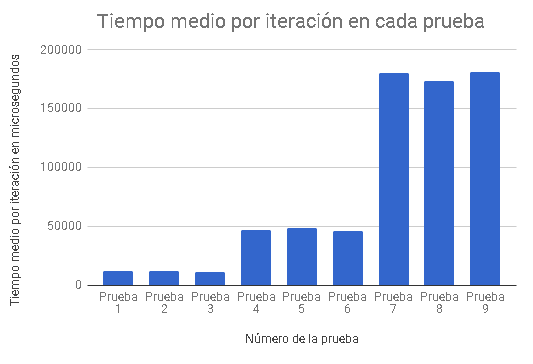
*Tiempo medio para 1 hilo:*

**

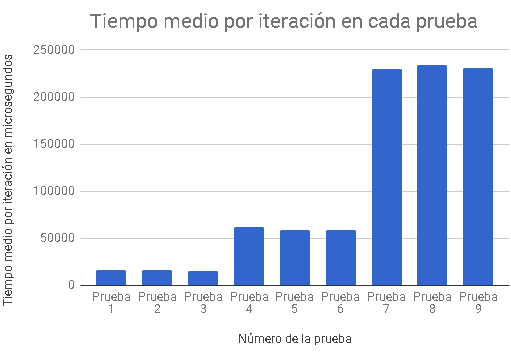
*Tiempo medio para 2 hilos:*

**

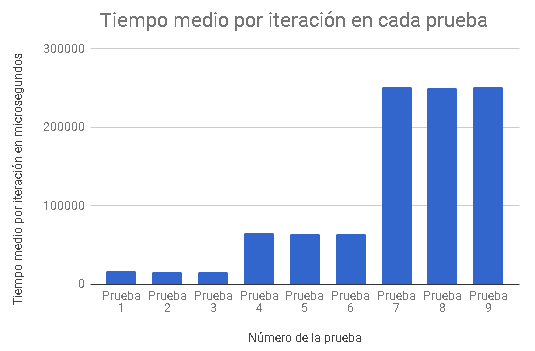
*Tiempo medio para 4 hilos:*

**

*Tiempo medio para 8 hilos:*

**

*Tiempo medio para 16 hilos:*

******

## 2.4 Impacto de la planificación (static, dynamic, guided) sobre 4 y 8 hilos

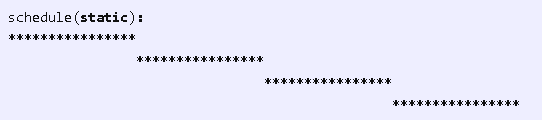
### 2.4.1 Introducción

La planificación describe cómo puede el programador determinar el tipo de *scheduling* de un loop. Hay varios tipos: *static, dynamic, guided, auto, runtime*.

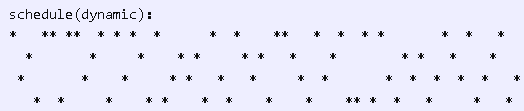
Con un *scheduling* static lo que obtenemos a nivel interno se puede resumir de la siguiente forma: OpenMP divide las iteraciones en fragmentos de tamaño *chunk-size* y distribuye los fragmentos a los hilos en un orden circular.  
 Cuando no se especifica el *chunk-size* OpenMP divide las iteraciones en fragmentos que son aproximadamente del mismo tamaño y distribuye como máximo un fragmento a cada hilo.

El *scheduling static* es apropiado cuando todas las iteraciones tienen el mismo costo computacional.

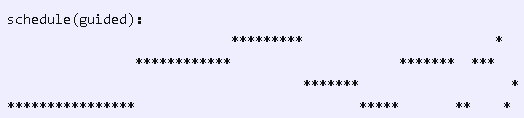
A continuación incluimos una representación gráfica de este *scheduling*



Con el *scheduling* dynamic OpenMP divide las iteraciones en fragmentos de tamaño *chunk-size*. Cada subproceso ejecuta un fragmento de iteraciones y luego solicita otro fragmento hasta que no haya más disponibles.  
 No hay un orden particular para distribuir los fragmentos a los hilos. El orden cambia cada vez que se ejecuta el *for*. Si no se especifica el tamaño del *chunk-size*, el valor predeterminado es uno.



Por último el tipo de *scheduling* guided es similar al dynamic. OpenMP vuelve a dividir las iteraciones en fragmentos. Cada subproceso ejecuta un fragmento de iteraciones y luego solicita otro fragmento hasta que no haya más disponibles.  
 La diferencia con el tipo de planificación dinámica está en el tamaño de los fragmentos. El tamaño de un fragmento es proporcional al número de iteraciones no asignadas dividido por el número de subprocesos. Por lo tanto, el tamaño de los trozos disminuye. Si no especificamos el tamaño del fragmento, el valor predeterminado es uno.



### 2.4.2 Impacto de la planificación sobre 4 hilos

Tras realizar pruebas con las diferentes configuraciones no hemos podido apreciar un impacto de la planificación sobre el tiempo en este caso, ha seguido el patrón expresado anteriormente pero no ha cambiado el tiempo de ejecución. Esto se puede deber a que las partes del código que hemos paralelizado de forma que se puedan planificar no suponen un elevado porcentaje del tiempo de ejecución.

### 2.4.3 Impacto de la planificación sobre 8 hilos

Tras realizar pruebas con las diferentes configuraciones no hemos podido apreciar un impacto de la planificación sobre el tiempo en este caso, ha seguido el patrón expresado anteriormente pero no ha cambiado el tiempo de ejecución. Esto se puede deber a que las partes del código que hemos paralelizado de forma que se puedan planificar no suponen un elevado porcentaje del tiempo de ejecución.

## 2.5 Conclusiones de la parte paralela

Hemos observado que la utilización de hilos que no existen en el procesador suponen un incremento en el tiempo de ejecución, mientras que las cifras de threads que están presentes en nuestro procesador reducen los tiempos de ejecución, cuando el número es mayor se incrementa gradualmente el tiempo de ejecución, independientemente de la configuración.

La paralelización con OPENMP nos permite paralelizar los bucles for de mayor importancia en nuestro código, sin embargo, no hemos conseguido reducir de manera significativa el tiempo de ejecución de la parte que supone un mayor porcentaje de la ejecución, la necesidad de incluir una sección crítica para almacenar las aceleraciones incrementa el tiempo de ejecución de nuestro programa como se podría esperar de cualquier código paralelo.

# 3. Evaluación de rendimiento

Las salidas de “init\_cong.txt” coinciden en ambos casos, tanto para la versión secuencial y la versión paralela. En cambio, las salidas del “out.txt” no hemos podido replicarlo incluso con el margen de error de 0.01.

También, hemos visto que el tiempo de ejecución del código base es menor que el nuestro debido probablemente a que no hacemos las operaciones exactamente en el mismo orden. Por ejemplo, para el caso de la versión secuencial vs la versión base, quedaría como 7,83s vs 1,961s (para 500 200 10 100 70), lo que indica que nuestro programa está mucho menos optimizado con respecto al caso base.

Por otro lado, las salidas de nuestros ficheros secuencial y paralelo, sí que coinciden.

# 4. Pruebas realizadas

Se han realizado las pruebas correspondientes a las gráficas de los apartados anteriores. Se ha ejecutado cada prueba al menos 10 veces. El entorno operacional ha sido guernika y sus especificaciones pueden ser encontradas en /proc/cpuinfo.

Otro tipo de pruebas realizadas para comprobar la similitud de los archivos de salida tanto con los nuestros como en el base, se ha usado el comando diff -s.

# 5. Conclusiones

En esta práctica hemos podido observar las diferencias que existen entre la programación secuencial y la paralela. Con el análisis que hemos realizado hemos podido observar que un programa tan simple como este puede mejorar su rendimiento si se asignan determinadas tareas a hilos, es decir, se distribuye el trabajo, provocando una asignación de los recursos más eficaz. Si en un futuro tuviéramos que mejorar el funcionamiento de un programa (de mayores dimensiones), la primera opción a la que podríamos recurrir para mejorar su rendimiento sería la paralelización ya que si uno conoce cómo se ejecuta cada línea, es decir, cómo se procesan en el ordenador, se puede mejorar la asignación de recursos.

Además, al realizar esta práctica hemos tenido que aprender C++, un lenguaje de programación que no habíamos utilizado hasta ahora, aunque guarda ciertas similitudes con otros lenguajes. Aunque hay que decir que no hemos podido hacer un código lo suficientemente igual al proporcionado para la práctica, cosa que es fácil de ver analizando los resultados de ambos códigos, probablemente debido a que no asignamos bien los recursos y a que no estemos siguiendo el mismo orden en las operaciones.