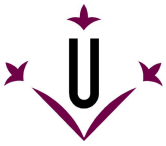



**Universidad de Lleida**

Prácticas 3 y 4

# Concurrencia con sincronizaciones

*Sistemas Concurrentes y Paralelos*

Pere Muñoz Figuerol

Enero 2023

# Índice de contenidos

# 1 Introducción

En estas dos prácticas se nos pedía implementar las concurrencias de las pràcticas 1-2, pero esta vez creando los hilos de manera estática (una sola vez al inicio de la aplicación), y que se controlase el flujo de ejecución utilizando sincronizaciones.
La 3 está implementada en C, utilizando como base la 1. Y la 4 está implementada en Java, y como base la 2.

Primeramente se hará un análisis de cómo se ha implementado la solución, a posteriori se mostrarán algunos gráficos analizando el rendimiento de las diferentes versiones, para finalmente sacar unas conclusiones.

# 2 Diseño de la solución

En este apartado se explicarán como se han implementado las soluciones a los dos lenguajes, así como un pequeño *pseudocódigo* ejemplo para dejarlo lo más claro posible.
Incidir en que no se ha implementado una estrategia de balanceo dinámica al cien por cien. Sí que en cada iteración se va recalculando cuántas partículas quedan por tratar, y así dividir los índices con este criterio. Pero lo que sería un balanceo activo, comprobando qué hilo es el más perjudicado y así reajustar índices, no se ha implementado por falta de tiempo. Aún así, la estrategia sí que se ha pensado y se mencionará más adelante.

## 2.1 Implementación C

En la implementación C, de igual forma que en la práctica 1, se ha implementado concurrentemente dos cómputos distintos. El primero es el cálculo del árbol cuaternario para almacenar las partículas. Debido a que era muy complejo gestionar esta parte solo con sincronizaciones, se ha dejado exactamente igual a la práctica 1.

El segundo es el cálculo de las fuerzas de las partículas. Para esta parte, se ha eliminado completamente la antigua función, y se ha creado una de nueva, ejecutada exclusivamente por los hilos creados. En la cual, mediante semáforos, variables de condición, barreras y mutex, se ha conseguido que los hilos respetasen el flujo de ejecución adecuado, y los resultados fuesen los correctos. A continuación se muestra un *pseudocódigo* resumiendo su funcionamiento:

```
void threadFunction(id)
    while(1)
        P(BuildTreeSemaphore)

        startIndex = id * (TotalParticles / TotalThreads)

        endIndex = (id+1) * (TotalParticles / TotalThreads)
```

```
CalculateForce(startIndex, endIndex)

GatherStatistics()

if (actualIteration % M == 0)
    PrintStatistics()
```

Este código será ejecutado concurrentemente por cada hilo y para cada iteración de la simulación. En caso que ocurra algún error, o se llegue a la última iteración, se finalizarán los hilos, el hilo principal hará los *joins* correspondientes y se finalizará la aplicación. Obviamente, todos los accesos a memoria compartida se han aislado utilizando mutexs y/o semáforos, así como las sincronizaciones de operaciones inter e intra iteracionales.

Por último, y como novedad a la anterior práctica, se ha implementado concurrencia en la IU. Se crea un hilo adicional (que no contabiliza para el total de hilos), que se encarga de refrescar las partículas que se muestran por pantalla, calculadas por los otros hilos. De igual forma, a continuación se muestra un *pseudocódigo* para ver la ejecución de este hilo:

```
void uiThread()
    while(1)
        Wait(PrintParticlesConditionVariable)

        copyOfTheParticles = particlesUpdated

        RefreshScreen(copyOfTheParticles)
```

Cada vez que el hilo termine una iteración (por lo tanto, se habrán actualizado todas las partículas), se informará al hilo-interfaz para que las actualice en pantalla. El resultado de la iteración $i$ se almacena en una copia dentro de esta función para que no interfiera con la *i+1* mientras se está calculando. De igual forma a la anterior, se finalizará cuando ocurra un error o se acabe la simulación.

## 2.2 Implementación Java

En la implementación Java, de igual forma a la práctica 2, se realizan dos cálculos concurrentes distintos.
El primero calcula las nuevas posiciones de las partículas y el segundo comprueba si se ha producido alguna colisión entre ellas.

Para la realización de estas concurrencias, se ha creado una función que ejecutarán todos los hilos creados. A continuación el *pseudocódigo* correspondiente:

```
void threadFunction(id)
```

```
while(true)
    P(NextIterationSemaphore)

    startIndex = id * (TotalParticles / TotalThreads)

    endIndex = (id+1) * (TotalParticles / TotalThreads)

    CalculateNewValues(startIndex, endIndex)

    V(NewValuesCalculatedSemaphore)

    P(StartCheckCollisionsSemaphore)

    CheckCollisions(startIndex, endIndex)

    V(EndedCheckingCollisionsSemaphore)

    P(PrintStatisticsSemaphore)

    PrintStatistics()

    V(EndedPrintingStatisticsSemaphore)
```

Básicamente se espera en todo momento a tener "permiso" del hilo principal antes de realizar cualquier nuevo cálculo. En este código se han simplificado muchas cosas, pero igual que en el anterior, todas las dependencias se han tenido en cuenta, y se han controlado correctamente con *Locks* y secciones críticas *synchronized*.
Comentar que no se ha implementado ninguna variable de condición ya que, aunque siendo requisito en el enunciado, no he encontrado la oportunidad para hacerlo de manera óptima y eficiente. Por lo tanto, he decidido no incluirla.

### 2.3 Estrategia de balanceo dinámico

Tal y como se ha comentado en la introducción de este apartado, solo se ha implementado el cálculo de índices con el número de partículas restantes lo más actualizado posible. La estrategia que se explicará a continuación no, pero cómo se pensó en su día, la voy a incluir en este informe para futuras mejoras en las implementaciones.

Lo que se pensó fue en crear un índice, que se calcularía uno por cada hilo, que intentaría revertir o reducir el desbalanceo generado. Este índice se recalcularía cada M iteraciones, para evitar sobrecargar mucho la aplicación. Básicamente, lo que haría cada hilo sería, comparar el tiempo que se ha tardado en ejecutar esas M iteraciones, respecto a la media (desbalanceo en tiempo). Dependiendo de lo grande o pequeño que sea este desbalanceo, se establecería un índice más alto o más pequeño, y dependiendo de si es un debalanceo

negativo o positivo, este índice iría al contrario con el signo.
Una vez tenemos calculado el índice, el número de partículas a calcular por cada hilo se reajustaría con este índice para aumentar o disminuir el número. A continuación se muestra un ejemplo para entenderlo mejor:

Supongamos que tenemos dos hilos ejecutando la aplicación concurrente, y al cabo de $M$ iteraciones, nos genera estas estadísticas:

| **Hilos** | **Tiempo/M iteraciones** | **Desbalanceo** | **Índice** |
|---|---|---|---|
| Hilo 1 | 2s | -1s | **+0.25** |
| Hilo 2 | 4s | +1s | **-0.25** |

Table 1: Ejemplo estrategia balanceo dinámico

Como se puede observar, existe un desbalanceo de un segundo positivo y negativo para el Hilo 2 y el Hilo 1, respectivamente. Por lo tanto, se puede calcular un índice de balanceo con el signo contrario, dependiendo de lo brusco que queremos que sea el cambio, puede ser más o menos grande. Una vez calculado, el índice de inicio y final de cada hilo se puede reajustar. Si en un inicio los rangos eran estos:

```
Bodies = 100

Hilo 1 -> Start: 0, End: 49.

Hilo 2 -> Start: 50, End: 99.
```

Después de aplicar la técnica de balanceo de carga, podrían quedar así:

```
Bodies = 100

Hilo 1 -> Start: 0, End: 74.

Hilo 2 -> Start: 75, End: 99.
```

Para calcular los índices actualizados se puede aplicar el índice de reajuste como se quiera, pero se pensó como el porcentaje del número total de partículas restantes. Por tanto, para el ejemplo de arriba, se ha utilizado este criterio. Así que, de manera general, se podría calcular así:

```
NewStart = Hilo-1.End + 1

NewEnd = PrevEnd + (Indice * TotalBodies)
```

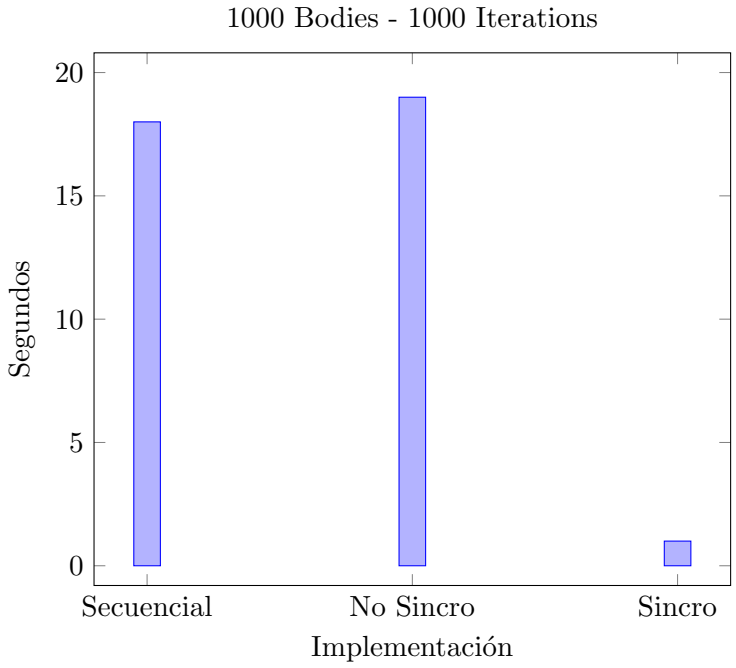
# 3 Análisis de mejoras

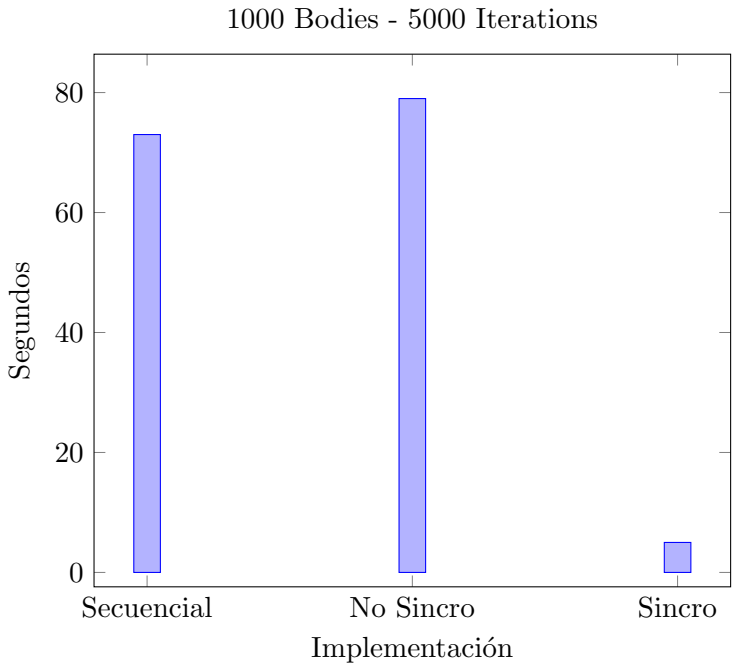
En este apartado se muestran gráficos comparativos entre los tiempos de ejecución de las diferentes implementaciones (secuencial, no sincronizado y sincronizado) así como con

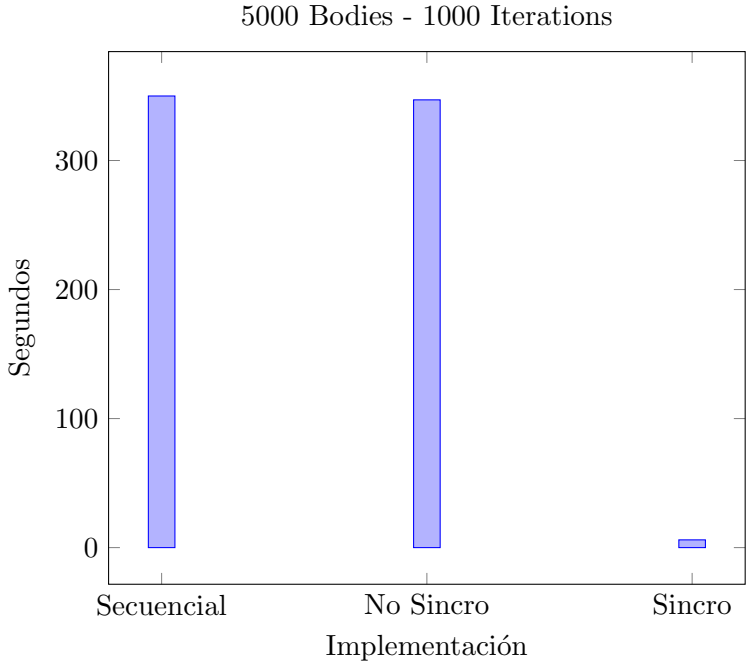
distintos hilos sincronizados.

## 3.1 Implementación C

### 3.1.1 Comparación entre las distintas implementaciones



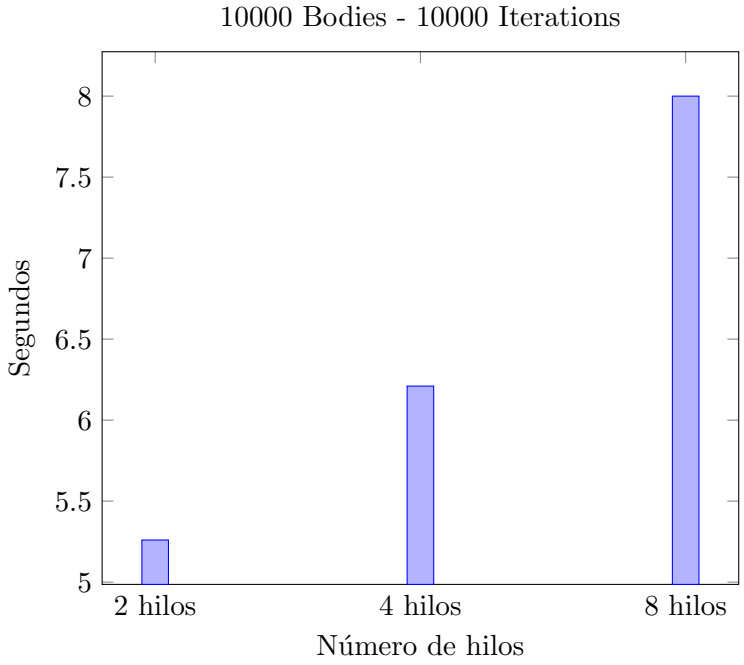




Tal y como se puede observar en estos gráficos, la mejoría es más que cuantiosa. Se rebajan los tiempos de ejecución de manera drástica, pasando de numerosos minutos a escasos segundos. En la mejora más significativa, se reduce el tiempo en hasta un 2000 por cien. Tener en cuenta también que la versión No Sincronizada no estaba del todo bien implementada, así que sí se hubiera hecho correctamente, la mejoría no sería tan espectacular.

### 3.1.2 Comparación entre ejecuciones con distintos hilos

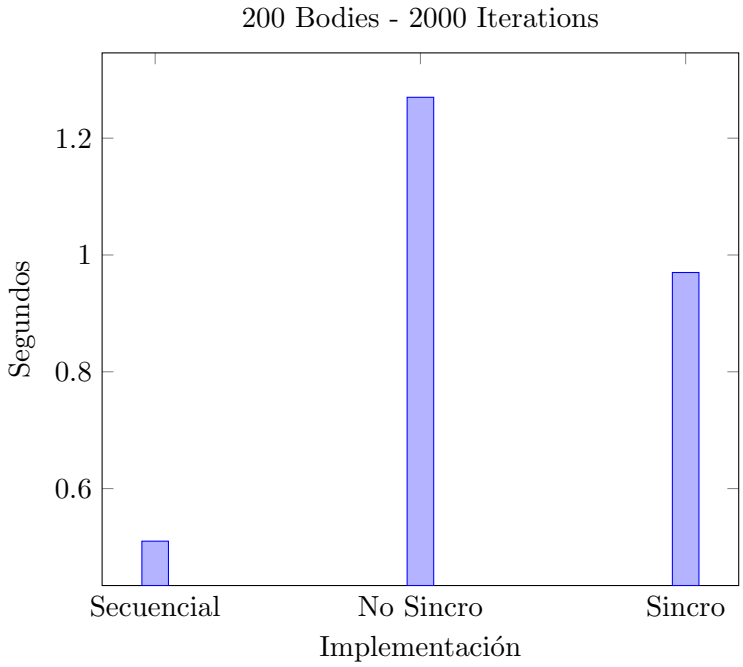
Para realizar la prueba de este apartado, se ha ejecutado una simulación con 10.000 partículas iniciales y 10.000 iteraciones máximas. Los resultados han sido los siguientes:

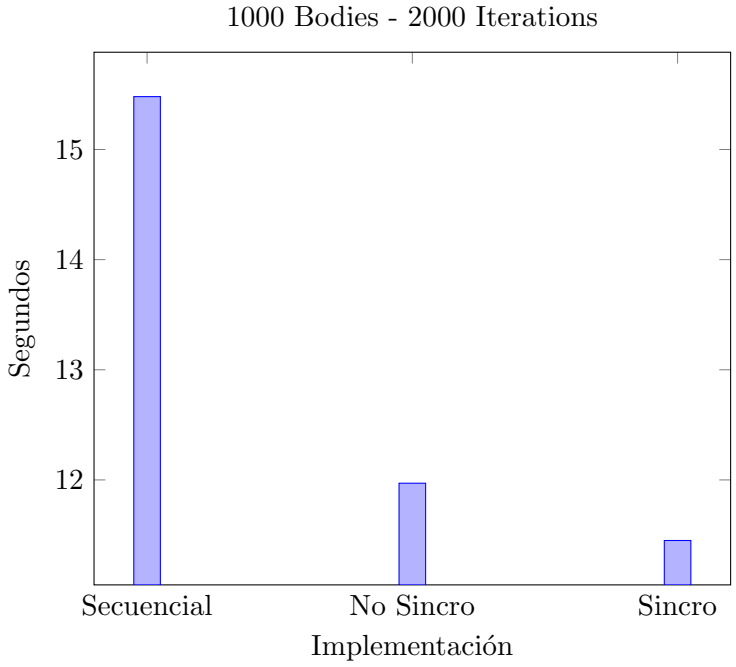


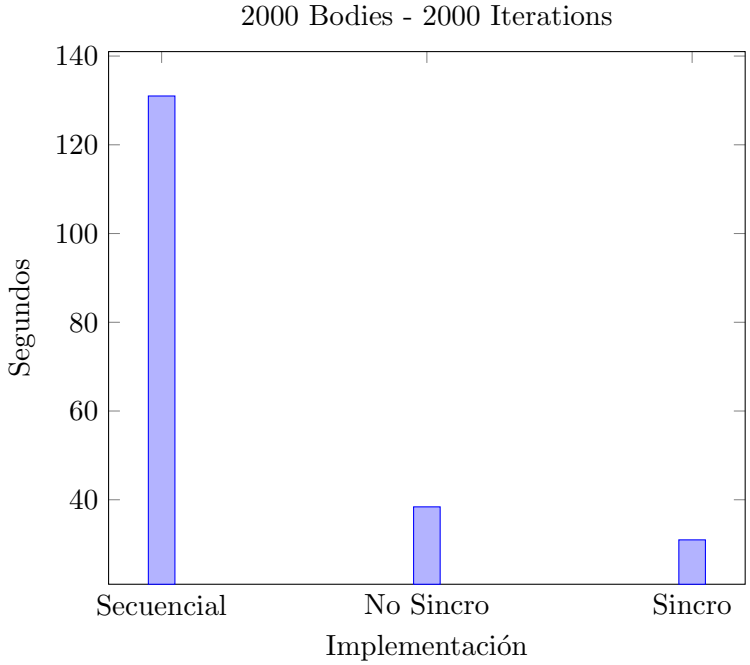

Como se puede observar, a medida que se va aumentando los hilos, aumenta también el tiempo de ejecución. El mejor rendimiento se aprecia con 2 hilos. Esto nos da a pensar que, con este tipo de configuración concreta, 2 hilos son totalmente suficientes para obtener el mejor rendimiento posible. Y que poner de más sería generar sobrecoste inútilmente.

## 3.2 Implementación Java

### 3.2.1 Comparación entre las distintas implementaciones

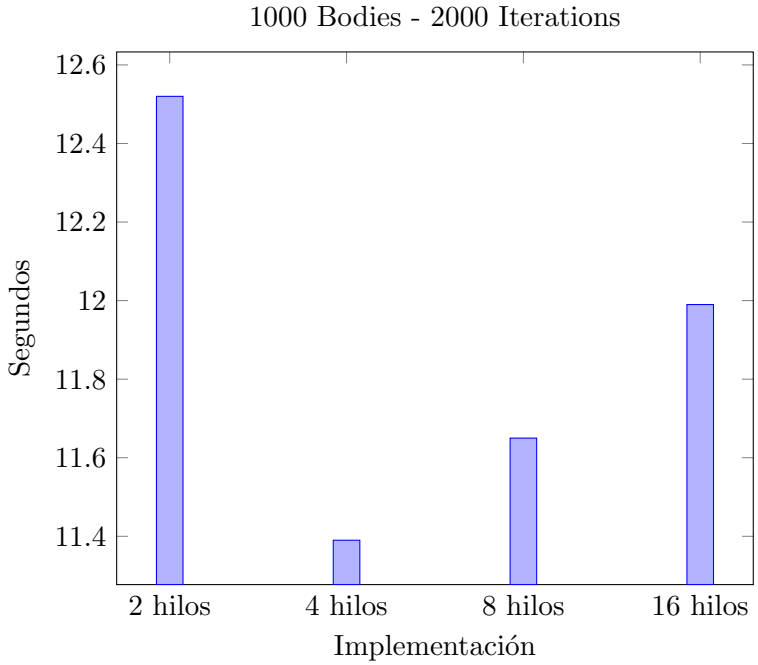





Como se aprecia en los distintos gráficos proporcionados, los tiempos mejoran progresivamente en las tres distintas implementaciones. Sí que es cierto que en las primeras el cambio no es muy notorio, o incluso empeora, respecto a la secuencial, sea por el número de partículas, de iteraciones o por el sobrecoste generado. A pesar de esto, de manera general y cuando se tiende a números más elevados, el cambio es mucho más claro y beneficioso.

### 3.2.2 Comparación entre ejecuciones con distintos hilos




Tal y como se ve en el gráfico, el mejor rendimiento se consigue en la ejecución de 4 hilos. A partir de ese punto, debido al sobrecoste generado de crear más hilos, va aumentando el tiempo de ejecución linearmente.
A pesar de todo esto, son resultados muy similares, y pueden haber sido influidos por el estado de ese instante concreto del procesador o del sistema en general.

## 4 Descripción de las prestaciones

Para la prueba de ejecución de las diferentes implementaciones, se ha utilizado un ordenador portátil con el sistema operativo *Ubuntu versión 22.04.1 LTS*. Especificamente, sus componentes principales son los siguientes:

- Procesador: Intel Core i5-10210U 1.6GHz-2.11GHz 4 núcleos i 8 procesadores lógicos.
- Memoria RAM: 16GB.

## 5 Conclusiones

Después de analizar las distintas ejecuciones he podido sacar las siguientes conclusiones:

- Implementación C: En esta práctica se ha mejorado muchísimo el rendimiento. En un principio me pareció muy raro, pensaba que había hecho alguna cosa mal, pero

comprobé los resultados de las ejeecuciones y los gráficos y todo parecía cuadrar... También me acordé que la parte del cálculo de las fuerzas en la práctica 1 no la hice del todo bien (me generaba muchísimo sobrecoste), así que con esto ya me cuadró del todo.

- Implementación Java: Se mejoran mucho los tiempos respecto a la implementación secuencial, pero respecto a la versión sin sincronizar no tanto. Puede que sea normal, que no hay más margen de mejoría, o que puede que haya implementado algunas sincronizaciones no necesarias (aunque no me lo parece).

Finalmente mencionar que esta práctica me ha ayudado a asentar conceptos de cara al examen final. He disfrutado mucho haciéndola y solucionando problemas de sincronizaciones.