UNIVERSITAT DE LLEIDA

ESCOLA POLITÈCNICA SUPERIOR

Sistemes Concurrents i Paral·lels - Curs 2022-2023

Pràctica 1: Simulación de partículas (problema N-Body)

Java threads

Informe presentado por: Alexandru Cristian Stoia Y2386362B

Pol Triquell Lombardo 48054396J

Grupo PraLab: GM2

Profesor: Manuel Fernando Cores Prado

Fecha de entrega: 14/11/2022

Grau en Enginyeria Informàtica

Universitat de Lleida

Índice de contenidos

Capítulo 1: Java Threads 3

A. ***Elección de la concurrencia.*** 3

B. ***División del trabajo.*** 3

C. ***Implementación del código.*** 3

I. ***Creación threads.*** 3

II. ***Modificación de la función calculateAllNewValues.*** 4

D. ***Tests de funcionamiento.*** 5

E. ***Comparativa de ejecución.*** 5

Capítulo 1: Java Threads

A. *Elección de la concurrencia.*

Tras ver las diferentes clases del proyecto, nos hemos decidido por una misma concurrencia tanto para la clase *SimulationLogicDouble*, *SimulationLogicFloat* y *SimulationLogicAP*.

Esta concurrencia se usará para calcular los valores de los objetos usando diversos hilos, para ello, modificaremos la función *calculateAllNewValues*, en la cual al principio se llama a la función que calcula los objetos de forma secuencial, si nosotros conseguimos llamar esta misma función de forma concurrente, obtendremos un *speedup*.

En las funciones que usan la GPU, la función trabaja por partículas, mientras que en la función que usa la CPU se le pasa una lista para el cálculo, este detalle se comenta más adelante, diferenciando el pequeño cambio hecho en el run de cada hilo.

B. *División del trabajo.*

Para obtener una división del trabajo equilibrada para cada *thread*, calcularemos la división entera entre el número de partículas y los *threads* utilizados. Una vez tenemos asignados esas partículas, si la división no es entera, tendremos partículas sin asignar, estas están calculadas con el módulo entre el número de partículas y los *threads* utilizados, asignando una partícula a cada *thread* hasta que no queden partículas por asignar.

De esta forma, cada *thread*, como máximo, tendrá una partícula extra, considerando esto una carga de trabajo dividida de forma óptima.

C. *Implementación del código.*

I. Creación threads.

Para crear el *thread*, hemos creado una nueva clase llamada *CalculateThread (ThreadCalculator en la clase SimulationLogicAp)*, que tiene como parámetros un inicio y un final, siendo estos, el rango de partículas que va a calcular.

En el caso de la función *SimulationLogicAp*, se le pasaba una lista de todos los elementos que contiene la simulación, por lo que, si partimos en trocitos esta lista (un *divide and conquer*), obtendremos el mismo resultado que si el programa calcula los elementos de forma secuencial, pero con hilos y speedup.

```
public class ThreadCalculator extends Thread{
    int start;
    int end;
    public ThreadCalculator(int start, int end){
        this.start = start;
        this.end = end;
    }
    @Override
    public void run(){
        //System.out.println("Thread started: " + start + " " + end);
        calculateNewValues(start, end);
    }
}
```

Como vemos, en la clase *ThreadCalculator*, Tiene dos parámetros: El inicio y el final de los elementos de la lista que calculará, con estos parámetros llama a la función principal para calcular los nuevos valores de los objetos con esta *sub-lista*.[1]

II. Modificación de la función calculateAllNewValues.

a. Simulation Logic AP

En esta clase, primero obtenemos el número de *threads* que se usarán en la ejecución (Se ha implementado un método estático *getNumberOfThreads* en la *clase Simulation Properties*), una vez tenemos los *threads* disponibles, hacemos el cálculo del trabajo por *thread* y el trabajo que no se ha asignado:

```
int numberOfThreads = SimulationProperties.getNumberOfThreads();
int taskPerThread = simulation.getObjects().size() / numberOfThreads;
int remainder = simulation.getObjects().size() % numberOfThreads;
```

Posteriormente, creamos el array de *threads* y asignamos el trabajo para cada uno:

```
int start = 0;
int end = taskPerThread;

for(int i = 0; i < numberOfThreads; i++){

    if(remainder > 0){ //If there is a remainder, assign one more job to the thread
        end++;
        remainder--;
    }

    threads[i] = new ThreadCalculator(start, end); //Create the thread with the given
sub array
    threads[i].start();

    start = end;
    end =start + taskPerThread; //Calculate the next sub array
}
```

Finalmente, hacemos el *join* de los *threads* y en caso de error, *printeamos* un *StackTrace* el cual crea un informe de los elementos activos en la pila de ejecución en un momento determinado durante la ejecución de un programa, permitiéndonos saber hasta qué punto se ha ejecutado correctamente.

---

[1] Código de la clase SimulationLogicAp, pero las otras clases tienen un aspecto muy similar, incluyendo en su caso un bucle *for* para iterar el número de partículas.

b. Simulation Logic Double y Float

En estas funciones, la base es la misma que en la anterior, el único cambio, es el funcionamiento de la llamada a la función desde el método run del thread, en el cual en vez de pasarle unos rangos para calcular una sub-lista, se le pasan unos rangos para hacer un bucle *for* y calcular cada uno de ellos (a causa de cómo está definida la función *CalculateNewValues*):

```
 @Override
public void run(){
    for(int i = start; i < end; i++){ //for each object call the method (range
of the thread)
        calculateNewValues(i);
    }
}
```

D. *Tests de funcionamiento.*

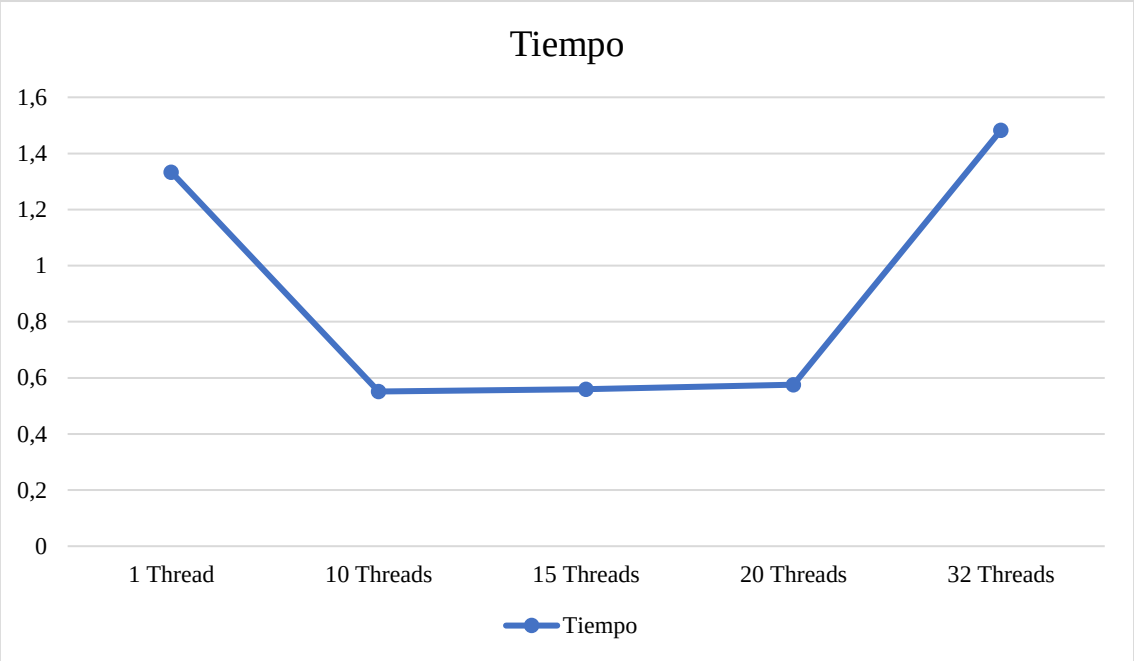
Para ver el correcto funcionamiento del código, asegurarnos que es determinista y que sigue funcionando de la misma manera que la versión original, hemos utilizados los archivos de entrada proporcionados en el campus y los hemos comparado los archivos de salida de los mismos con los nuestros, estos coinciden, por lo tanto, podemos asegurar que nuestra implementación del código no ha modificado el funcionamiento del original.

E. *Comparativa de ejecución.*

Para poder comprobar si tenemos un *speedup* real, hemos decidido hacer diferentes pruebas con un mismo *input*, utilizando un número diferente de *threads* en cada caso:

Simulación de 2000 objetos, 1000 iteraciones usando *float* y la CPU.

Threads	**Tiempo**
1	1 m. 33.305 s.
10	55.139 s.
15	55.992 s.
20	57.52 s
32	1 m. 4.831 s



Viendo la tabla y el gráfico, observamos claramente como el tiempo con diez y quince *threads* es menor al resto, con un *thread* todo el trabajo se calcula con el mismo hilo, a partir de los veinte hilos, es cuando se produce un *overhead*, provocando la ralentización del programa.