Programmazione concorrente e distribuita
Assignment 01

jiahao Guo matricola 0001173814

10 aprile 2025

1 Analisi del problema

Il problema affrontato riguarda la parallelizzazione dell'algoritmo Boids, con particolare attenzione alla fase di calcolo delle velocità e delle posizioni dei boid. Poiché la determinazione delle nuove posizioni dipende dalle velocità aggiornate, è necessario introdurre un meccanismo di sincronizzazione tra queste due operazioni per evitare race condition e/o altri effetti indesiderati. Un ulteriore aspetto da considerare è l'aggiornamento della vista (view) del sistema, che deve avvenire solo dopo l'aggiornamento delle posizioni dei boids, al fine di evitare letture incoerenti. Come punto di riferimento ci è stato fornito la versione sequenziale dell'algoritmo, nella quale i boids vengono aggiornati uno dopo l'altro. Questa soluzione si dimostra sufficientemente efficiente quando il numero di boid è contenuto, ma non è scalabile: all'aumentare del numero di elementi, le prestazioni calano drasticamente, a causa della mancanza di parallelismo e dei crescenti tempi di attesa. Per superare questi limiti e per realizzare le versioni concorrenti, si è deciso di operare progettare tre diverse implementazioni, ciascuna basata su un differente paradigma di programmazione concorrente:

- **Multi-Threads**: utilizza i platform threads del sistema operativo per parallelizzare l'algoritmo.
- **Task/Executor-based**: sfrutta API di Java Executor Framework.
- **Virtual Threads**: impiega i threads virtuali di Java per avere una parallelizzazione light ed efficiente.

2 Multi-Threads

Implementazione con platform threads

Per una soluzione ottimale basata su *threads* si è deciso di utilizzare un numero di threads pari al numero di **processori disponibili** del sistema aumentato di uno. La lista dei Boid è stata quindi suddivisa in un numero di **mini-batch** pari al numero di thread, in modo che ogni thread possa elaborare indipendentemente un mini-batch di Boid. Per la gestione della concorrenza, venogno utilizzate tre barriere di sincronizzazione: `barrierSync`, `barrierVel` e `barrierPos`, condivise tra il *main thread* e i *worker thread*. L'obiettivo è di coordinare in modo preciso l'esecuzione delle fasi dell'algoritmo Boids evitando che il main thread aggiorni la view prima che i worker finiscono di aggiornare.

- `barrierSync`: sincronizza l'inizio di ciascuna iterazione tra tutti i thread.
- `barrierVel`: sincronizza la fase di aggiornamento delle velocità dei Boid.
- `barrierPos`: sincronizza la fase di aggiornamento delle posizioni.

In particolare, barrierSync e barrierPos hanno un numero maggiore di counter per mandare in avanti l'esecuzione, in quanto viene considerato anche il main thread.

Comportamento del `main thread`

Il `main thread` controlla lo stato della simulazione (esecuzione, pausa e reset) mediante una variabile di condizione (`restartCondition`) associata a un `lock`. Quando la simulazione è attiva, entra nel ciclo principale dove:

1. Chiama `barrierSync.hitAndWaitAll()` per dare il via all'iterazione si mette in attesa.
2. Attende che tutti i worker abbiano completato la fase di aggiornamento delle velocità con `barrierVel.hitAndWaitAll()`.
3. Attende che sia completato anche l'aggiornamento delle posizioni con `barrierPos.hitAndWaitAll()`.
4. Se la `view` è presente, aggiorna la visualizzazione.

In caso di interruzione (stop), il `main thread` invia un `interrupt()` a ciascun worker e ne attende la terminazione con `join()`. Nel caso di una sospensione, utilizza una `restartCondition` per attendere finché non riceve un segnale di ripresa.

Comportamento dei `BoidWorker`

Ogni `BoidWorker` esegue continuamente il ciclo di aggiornamento dei propri Boid. Il ciclo prevede:

1. Attende su `barrierSync` finché tutti i worker thread siano pronti.
2. Aggiorna le velocità dei Boid nel proprio *mini-batch*, richiamando `boid.updateVelocity(model)`.
3. Attende su `barrierVel` che tutti abbiano concluso la fase precedente.
4. Aggiorna le posizioni tramite `boid.updatePos(model)`.
5. Attende su `barrierPos` prima di iniziare una nuova iterazione.
6. Quando l'ultimo worker termina, spinge anche il main thread in avanti.

In caso di interruzione (`Thread.interrupted()`), il thread termina correttamente la sua esecuzione.

3 Task/Executor-based

Implementazione con Java Executor Framework

Il *Java Executor Framework* è un'API introdotta per semplificare la gestione dei thread e dei task concorrenti. Fornisce un'astrazione di alto livello rispetto alla creazione manuale dei thread, consentendo l'esecuzione asincrona di compiti (`Runnable` o `Callable`) tramite pool di thread riutilizzabili. Tra le implementazioni principali vi è `ExecutorService`, che offre metodi per la sottomissione, gestione e terminazione ordinata dei task.

Nel contesto della simulazione, è stata realizzata una classe `Executor` che utilizza un pool di thread con dimensione pari al numero di core disponibili sulla macchina. Tale classe si occupa della gestione dei task relativi all'aggiornamento delle velocità e delle posizioni degli elementi simulati, organizzandoli in due distinte liste di oggetti `Future`.

Una volta generati, i task vengono restituiti al thread principale sotto forma di due liste separate: una per l'aggiornamento delle velocità e una per quello delle posizioni. Per garantire una corretta sequenza di esecuzione, il thread principale procede dapprima con l'attesa della terminazione dei task relativi alla velocità (mediante invocazione del metodo `get()` su ciascun `Future`), e solo successivamente gestisce l'aggiornamento delle posizioni con analoga modalità. Al termine delle due fasi di elaborazione, viene aggiornata la vista grafica della simulazione.
Le operazioni di sospensione, ripristino e avvio della simulazione sono gestite in modo analogo alla versione basata su `PlatformThread`, attraverso l'utilizzo di un meccanismo di sincronizzazione basato su `Lock` e `Condition`. Rispetto all'approccio basato su `PlatformThread`, l'utilizzo dell'`Executor Framework` si è rivelato più semplice da gestire e, allo stesso tempo, più efficiente. Gli aggiornamenti delle entità risultano infatti più rapidi grazie alla migliore gestione delle risorse offerte dal pool di thread. Infine, si nota che in caso di `reset` della simulazione, l'`ExecutorService` attivo viene arrestato tramite il metodo `shutdown()`, e successivamente viene istanziato un nuovo esecutore per garantire un corretto riavvio del sistema.

4 Virtual Threads

Implementazione con Virtual Threads

La soluzione basata su `VirtualThread` prevede l'utilizzo di un thread per ciascun boid. Questo approccio è reso possibile dalla natura estremamente leggera dei virtual thread, che consente di gestirne un numero molto elevato rispetto ai tradizionali `PlatformThread`.

Durante l'inizializzazione, per ogni boid viene creato un `VirtualThread` in stato *unstarted*. Nella fase di simulazione, viene eseguito prima l'aggiornamento delle velocità: tutti i virtual thread associati a questa fase vengono avviati, e il thread principale attende la loro terminazione mediante il metodo `join()`.

Successivamente, lo stesso procedimento viene applicato all'aggiornamento delle posizioni: i virtual thread relativi vengono avviati e il thread principale ne attende la conclusione.

Infine, completati gli aggiornamenti di stato, viene effettuato il rendering della vista della simulazione. Questo approccio consente una gestione concorrente efficiente e strutturata.

5 Dettagli implementativi

5.1 CustomWorker

```
public class BarrierImpl implements Barrier {

    private final int nTotal;
    private int nArrived = 0;
    private int generation = 0;
    /*Tracking if they are on same round generation*/

    public BarrierImpl(int nTotal) {
        this.nTotal = nTotal;
    }

    @Override
    public synchronized void hitAndWaitAll() throws InterruptedException {
        int actualGeneration = this.generation;
        this.nArrived++;
        if (this.nArrived == this.nTotal) {
            this.nArrived = 0;
            this.generation++;
            notifyAll();
        } else {
            while (this.generation == actualGeneration) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }
        }

    }
}
```

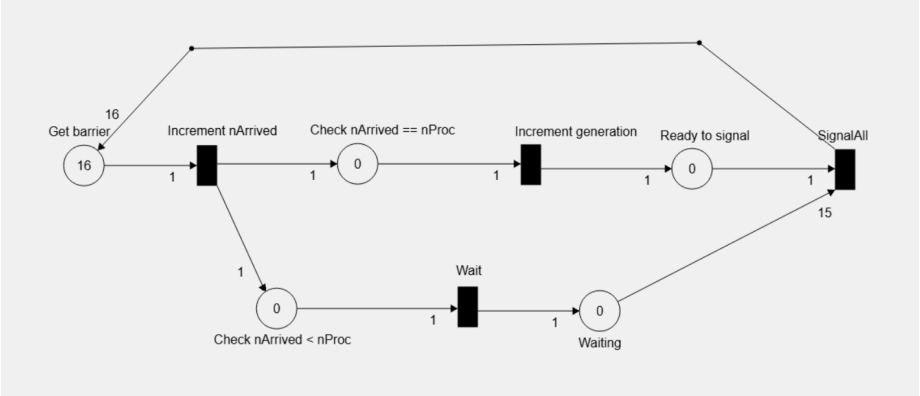

Figura 1: Petri net della barriera, in caso di 16 workers + 1 main thread

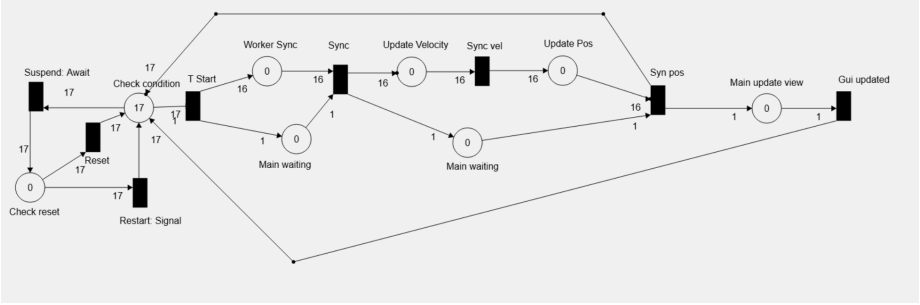

Figura 2: Petri net di Platform-simulation

5.2 CustomExecutor

```
public class CustomExecutor {
    private final BoidsModel model;
    private List<Boid> boids;
    private ExecutorService executor;

    public CustomExecutor(List<Boid> boids, int nProc, BoidsModel model) {
        this.boids = boids;
        this.executor = Executors.newFixedThreadPool(nProc);
        this.model = model;
    }

    public List<Future<?>> computeVelocity() {
        List<Future<?>> velocityResults = new ArrayList<>();
        if (executor.isShutdown()) return null;
        for (Boid boid : boids) {
            try {
                Future<?> res = executor.submit(() -> boid.updateVelocity(this.model));
                velocityResults.add(res);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return velocityResults;
    }

    public List<Future<?>> computePosition() {
        List<Future<?>> positionResults = new ArrayList<>();
        if (executor.isShutdown()) return null;
        for (Boid boid : boids) {
            try {
                Future<?> res = executor.submit(() -> boid.updatePos(this.model));
                positionResults.add(res);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        return positionResults;
    }

    public void shutdown() {
        executor.shutdown();
        try {
            if (!executor.awaitTermination(800, TimeUnit.MILLISECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }
    }

}
```

5.3 Virtual Thread

```
public void runSimulation() throws InterruptedException {
        ...

            var t0 = System.currentTimeMillis();
            var listVelocity = new ArrayList<Thread>();
            var listPosition = new ArrayList<Thread>();
            for (Boid boid : this.model.getBoids()) {
                Thread v = Thread.ofVirtual().unstarted(() -> {
                    try {
                        boid.updateVelocity(this.model);
                    } catch (Exception ex) {
                    }
                });
                v.start();
                listVelocity.add(v);
            }

            listVelocity.forEach(thread -> {
                try {
                    thread.join();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            });
            for (Boid boid : this.model.getBoids()) {
                Thread p = Thread.ofVirtual().unstarted(() -> {
                    try {
                        boid.updatePos(this.model);
                    } catch (Exception ex) {
                    }
                });
                p.start();
                listPosition.add(p);
            }

            listPosition.forEach(thread -> {
                try {
                    thread.join();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            });
        ...
        }
    }
```

6 Verifica JPF della versione Platform Threads

Obiettivo di JPF

L'obiettivo dell'utilizzo di Java PathFinder (JPF) è quello di verificare in modo sistematico il comportamento concorrente della simulazione basata su `PlatformThread`, individuando potenziali condizioni di errore come deadlock e race condition che potrebbero non emergere durante test manuali o esecuzioni tradizionali.

Setup della verifica

Poiché JPF supporta solo fino a Java 11, è stato necessario adattare parte del codice originale. In particolare, le classi `P2d` e `V2d`, inizialmente implementate come record (introdotti in Java 14), sono state convertite in classi tradizionali. Questo ha permesso la compatibilità con l'ambiente di verifica fornito da JPF.

Risultato della verifica

Durante l'esecuzione della verifica con JPF, è stato individuato un deadlock tra thread, causato da una condizione `while` bloccante all'interno dell'implementazione della barriera di sincronizzazione. Tale condizione impediva il proseguimento dei thread, bloccandoli in attesa di uno stato che non poteva più essere raggiunto.

Una volta individuato il problema, la barriera è stata corretta, risolvendo definitivamente il deadlock.

Successivamente, la verifica ha evidenziato una condizione di *data race* durante la lettura della posizione di alcuni boid nel corso degli aggiornamenti. Questo difetto di concorrenza non è stato ancora risolto, ma è emerso solo grazie alla verifica formale con JPF: senza l'analisi automatica, sarebbe stato difficile rilevarne l'esistenza.

Tali risultati dimostrano l'utilità di strumenti di verifica come JPF per l'analisi di sistemi concorrenti complessi, anche su codice apparentemente funzionante.

Prestazioni e Misurazioni

Per valutare le prestazioni del sistema di simulazione, è stata effettuata una serie di misurazioni su più esecuzioni, utilizzando un numero crescente di boid. Gli esperimenti sono stati condotti variando il numero di agenti simulati secondo i seguenti valori: 1000, 1500, 3000, 5000, 8000, 10000 e 12000 boid. Il Fps massimo impostato a 50.

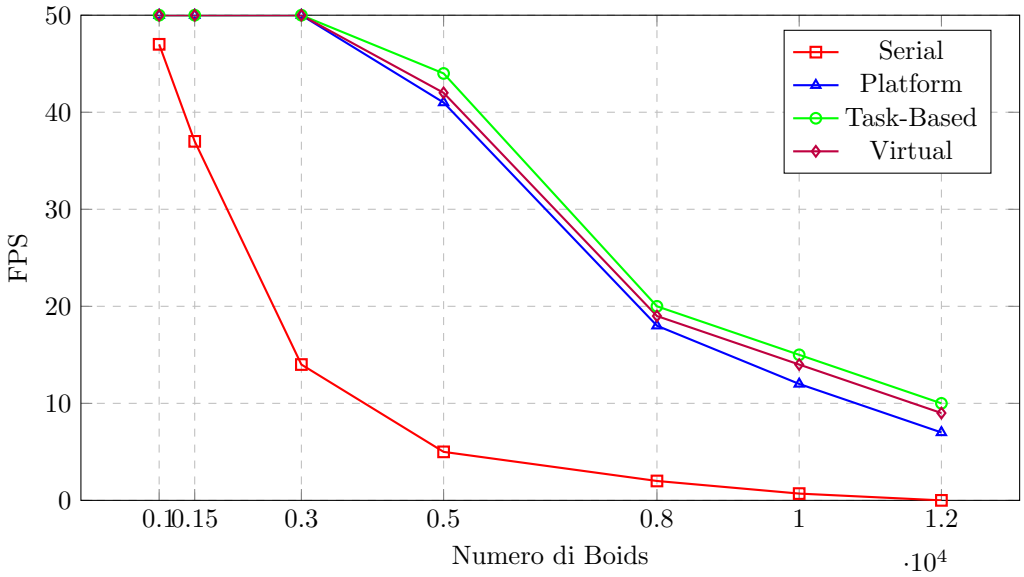

Figura 3: FPS medi in base al numero di Boids per ciascuna strategia di parallelizzazione.

Per avere un'ulteriore livello di precisione, è stata valutata anche il tempo computazionale medio delle iterazioni.

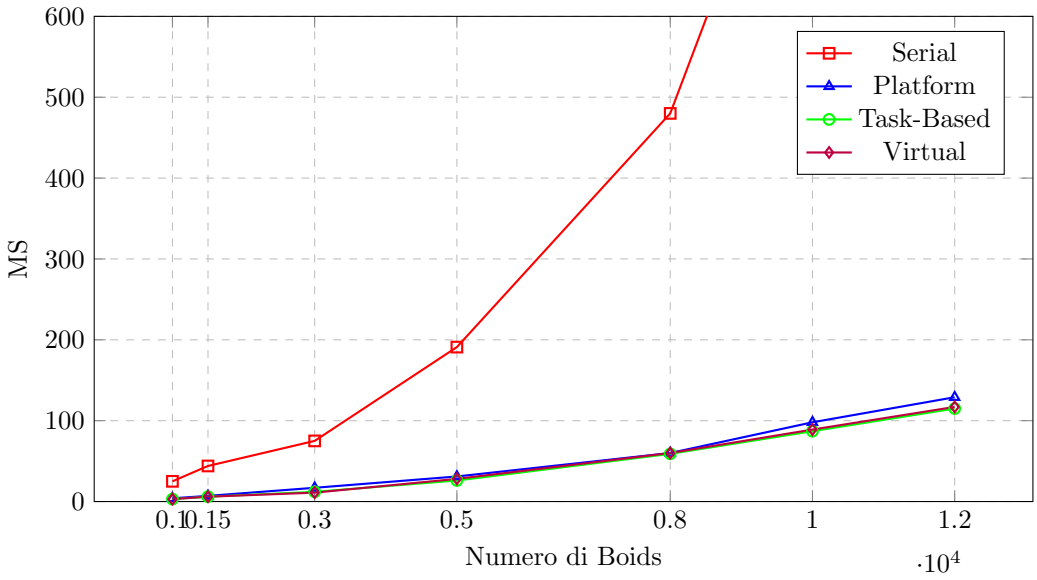

Figura 4: Millisecondi medi in base al numero di Boids per ciascuna strategia di parallelizzazione.

7 Conclusioni

Come si può osservare dal primo grafico, la versione seriale dimezza il framerate già al raggiungimento di 3000 *Boids*. Al contrario, le versioni concorrenti riescono a mantenere un framerate medio accettabile anche con carichi di 6000-8000 *Boids*. Tra queste, la versione basata su `Executor` (Task-Based) si distingue come la più performante, mostrando un numero di FPS leggermente superiore rispetto alle altre due. Questo vantaggio può essere attribuito al livello di astrazione più elevato con cui opera, che consente alla *JVM* e alle librerie sottostanti di effettuare ottimizzazioni più efficaci. Le implementazioni basate su *Task-Based* e *Virtual Thread* offrono prestazioni molto simili tra loro, risultando entrambe più efficienti rispetto alla versione con *Platform Thread*. Questo evidenzia come i *Virtual Thread*, pur essendo notevolmente più leggeri rispetto ai thread di sistema, siano in grado di gestire un numero elevato di istanze (uno per ogni *Boid*) senza un calo significativo delle prestazioni. In generale, tutte e tre le versioni concorrenti hanno portato ad un miglioramento delle prestazioni rispetto all'approccio seriale. Tuttavia, si può affermare che la gerarchia di efficienza sia:

$$\texttt{Executor} >= \texttt{Virtual Thread} > \texttt{Platform Thread} >>> \texttt{Sequential}$$

tenendo presente che nel *Platform Thread* le prestazioni possono variare sensibilmente in base alle caratteristiche hardware della macchina utilizzata.