Simulazione dei Boids

Analisi e Implementazione Concorrente

Autore: Sajmir Buzi

Anno Accademico 2025

Contents

1 Analisi del Problema

1.1 Descrizione del Problema

L'**Assignment01** riguarda la simulazione dei boids in 3 versioni specifiche(**multithread**,**executor** e **VThreads**). La simulazone dei Boids rappresenta un classico problema nell'ambito dei sistemi multi-agente, in cui ogni agente (boid) segue regole locali per determinare il proprio movimento. Queste regole sono:

- **Separazione**: evitare collisioni con i boid vicini.
- **Allineamento**: allineare la propria direzione con quella dei boid vicini.
- **Coesione**: muoversi verso il centro di massa dei boid vicini.

L'obiettivo è simulare il comportamento collettivo emergente di un gruppo di boid in un ambiente bidimensionale.

1.2 Aspetti Rilevanti dal Punto di Vista Concorrente

La natura intrinsecamente parallela della simulazione dei boid presenta diverse sfide e opportunità:

- **Parallelismo**: ogni boid può aggiornare il proprio stato indipendentemente, rendendo il problema altamente parallelo e adatto a implementazioni concorrenti.
- **Sincronizzazione**: è fondamentale gestire l'accesso concorrente alle strutture dati condivise per evitare condizioni di race e garantire la coerenza dello stato globale.
- **Rendering Grafico**: l'aggiornamento della visualizzazione deve essere coordinato con lo stato del modello per rappresentare accuratamente la posizione e il movimento dei boid.

2 Descrizione del Design e dell'Architettura

Per affrontare il problema, sono state implementate tre diverse tecniche di concorrenza in Java: **Multithreading con Thread Tradizionali**, **Thread Pool con ExecutorService** e **Thread Virtuali con Project Loom**. Ognuna di queste tecniche è stata integrata in un'architettura basata sul pattern **Modello-Vista-Controllo (MVC)**.

2.1 Architettura Modello-Vista-Controllo (MVC)

L'architettura **MVC** è stata adottata per separare le responsabilità all'interno dell'applicazione:

- **Modello (Model)**: gestisce lo stato dei boid e le regole di aggiornamento. La classe principale è `BoidsModel`, che contiene la lista dei boid e fornisce metodi per aggiornarne lo stato.
- **Vista (View)**: responsabile del rendering grafico dei boid. Le classi `BoidsPanel` e `BoidsView` si occupano di disegnare i boid sullo schermo basandosi sullo stato fornito dal modello.
- **Controllo (Controller)**: coordina l'aggiornamento del modello e della vista. La classe `BoidsSimulator` gestisce il ciclo di simulazione, aggiornando lo stato dei boid e richiedendo il rendering alla vista.

2.2 Caso Multithreading

In questa implementazione, ogni boid è gestito da un thread separato. Il processo è il seguente:

1. **Creazione dei Thread**: per ogni boid, viene creato un nuovo thread che esegue un ciclo continuo di aggiornamento della posizione e della velocità.
2. **Sincronizzazione**: l'accesso alle strutture dati condivise, come la lista dei boid nel `BoidsModel`, è protetto mediante blocchi `synchronized` per evitare condizioni di race.
3. **Gestione del Ciclo di Vita**: ogni thread esegue un loop infinito fino a quando non viene interrotto, aggiornando lo stato del boid e dormendo per un breve intervallo per controllare la frequenza di aggiornamento.

Questa prima implementazione si basa sull'approccio classico alla concorrenza, utilizzando direttamente i thread forniti da Java. In particolare, ogni boid viene gestito da un proprio thread indipendente, il che consente a ciascun agente di aggiornare il proprio stato (velocità e posizione) in parallelo agli altri.

Il progetto è stato strutturato seguendo un'architettura di tipo MVC (Model-View-Controller), che aiuta a separare in modo ordinato le responsabilità:

- **Il Modello** (`BoidsModel`): contiene la lista condivisa dei boid ed è progettato per essere *thread-safe*, grazie all'utilizzo di una lista sincronizzata (`Collections.synchronizedList`).
- **La Vista** (`BoidsView`): si occupa della parte grafica e permette all'utente di controllare parametri come numero di boid e i pesi delle forze di separazione, allineamento e coesione.
- **Il Controller** (`BoidsSimulator`): è responsabile dell'avvio, sospensione e interruzione della simulazione, nonché della creazione dei boid e dei rispettivi thread.

2.2.1 Il Comportamento dei Thread

Ogni boid viene inizializzato con posizione e velocità casuali. Una volta creato, viene assegnato a un `BoidThread`, che è una classe che estende `Thread` e si occupa in autonomia del suo aggiornamento.

Il ciclo di vita di ogni thread è semplice: in un loop continuo, aggiorna la velocità del proprio boid in base alle regole del modello (separazione, allineamento e coesione), ne aggiorna la posizione e infine dorme per circa 40 millisecondi per simulare un framerate di 25 FPS. Il thread può anche essere messo in pausa o ripreso, grazie a meccanismi di sincronizzazione basati su `wait()` e `notify()`.

2.2.2 Sincronizzazione e Gestione della Concorrenza

Un aspetto importante è che l'accesso al modello avviene sempre in modo sincronizzato, in modo da evitare condizioni di race o altri problemi legati alla concorrenza. Ad esempio, l'aggiornamento della velocità di un boid, che dipende dalle posizioni e velocità degli altri boid, viene eseguito dentro un blocco `synchronized (model)`.

2.2.3 Considerazioni sul Modello

Questo approccio è efficace soprattutto per simulazioni con un numero moderato di boid, dove il numero di thread rimane gestibile. Il vantaggio principale è che ogni boid è effettivamente autonomo e lavora in parallelo con gli altri, riflettendo bene il comportamento distribuito del sistema.

Tuttavia, questo modello presenta anche alcuni limiti:

- **Scalabilità limitata**: la creazione di un thread per ogni boid non scala bene. Quando il numero di agenti cresce, il carico sul sistema operativo per gestire i thread può diventare eccessivo.
- **Sincronizzazione e Collo di Bottiglia**: poiché i thread accedono a una risorsa condivisa (il modello dei boid), è necessaria una sincronizzazione costante, che può diventare un collo di bottiglia prestazionale.

In sintesi, questa versione rappresenta una buona base di partenza per comprendere la simulazione distribuita dei boid, ma evidenzia anche le sfide legate alla concorrenza tradizionale, soprattutto in termini di scalabilità ed efficienza.

2.3 Caso Executor Framework

In questa implementazione, si sfrutta l'**Executor Framework** di Java per gestire la concorrenza, evitando la gestione manuale dei thread e migliorando la gestione delle risorse. L'Executor Framework consente di gestire un pool di thread, al fine di eseguire i task in modo più efficiente.
La simulazione viene avviata creando un `ExecutorService`, che gestisce i thread per ogni boid. Ogni boid è rappresentato da un `BoidTask`, che è un `Runnable` che aggiorna lo stato del boid in parallelo con gli altri.

2.3.1 Struttura del Codice

- **BoidsExecutorSimulator**: questa classe gestisce la simulazione, creando il modello e la vista e utilizzando un `ExecutorService` per eseguire i thread dei boid.
- **BoidTask**: rappresenta il compito di ogni boid, che viene eseguito in un thread separato. Ogni `BoidTask` aggiorna il boid e rispetta lo stato di pausa, se attivato.
- **Sincronizzazione**: un `ReentrantLock` e una `Condition` sono utilizzati per gestire lo stato di pausa della simulazione, impedendo che i thread aggiornino lo stato del boid quando la simulazione è sospesa.

2.3.2 Dettagli di Implementazione

Il `BoidsExecutorSimulator` utilizza un `ExecutorService` per creare un pool di thread. Quando la simulazione viene avviata, un numero di boid viene creato e assegnato a un thread, che esegue il task definito nella classe `BoidTask`. Ogni `BoidTask` aggiorna la posizione e la velocità del boid, rispettando lo stato di pausa quando necessario.

In particolare:

1. **Creazione del Pool di Thread**: viene utilizzato un `ExecutorService` con un numero fisso di thread (uno per ogni boid) per eseguire i task.
2. **Sincronizzazione**: ogni thread verifica lo stato di pausa della simulazione tramite un `ReentrantLock`. Se la simulazione è in pausa, i thread vengono messi in attesa finché non vengono ripresi.

3. **Task Eseguito da Ogni Boid**: ogni boid esegue un task che aggiorna la sua velocità e posizione in base alle regole di separazione, allineamento e coesione. Ogni thread dorme per 40 millisecondi, simulando un framerate di 25 FPS.

2.3.3 Gestione della Pausa

Quando l'utente mette in pausa la simulazione, i thread vengono temporaneamente bloccati utilizzando la sincronizzazione tramite `lock` e `pauseCondition`. I thread dei boid vengono messi in attesa finché non viene invocata l'azione di ripresa, che segnala a tutti i thread di riprendere l'esecuzione.

2.3.4 Considerazioni sulla Gestione delle Risorse

Utilizzando l'Executor Framework, l'approccio con `ExecutorService` risulta più scalabile rispetto all'utilizzo diretto dei thread. L'Executor gestisce automaticamente il ciclo di vita dei thread, riducendo il carico sul sistema operativo. Inoltre, la gestione centralizzata del pool di thread migliora le prestazioni rispetto alla creazione manuale di thread per ogni boid, come nel caso precedente con i thread tradizionali.

2.3.5 Limiti e Vantaggi

Il principale vantaggio di questa implementazione è che il numero di thread è gestito centralmente dall'Executor, il che riduce il sovraccarico rispetto alla creazione di un thread per ogni boid. Tuttavia, in caso di un numero molto elevato di boid, la creazione di un numero elevato di thread può comunque causare problemi di performance, sebbene l'Executor Framework possa gestirli in modo più efficiente.

In sintesi, l'uso dell'Executor Framework migliora la gestione della concorrenza e la scalabilità della simulazione, consentendo una gestione più fluida e sicura delle risorse rispetto all'approccio tradizionale con thread separati.

2.4 Caso Virtual Threads

In questa implementazione, i boid sono gestiti utilizzando i *virtual threads*, una funzionalità introdotta nelle versioni più recenti di Java per semplificare la gestione della concorrenza e migliorare le prestazioni in scenari con un alto numero di thread.

1. **Creazione dei Virtual Threads**: Per ogni boid, viene creato un `virtual thread` che esegue un ciclo continuo di aggiornamento della posizione e della velocità. A differenza dei thread tradizionali, i virtual threads sono leggeri in termini di risorse e possono essere gestiti in modo più efficiente.

2. **Sincronizzazione**: Anche se i virtual threads sono gestiti in modo più efficiente rispetto ai thread tradizionali, viene comunque utilizzato un monitor (`BoidMonitor`) per gestire lo stato di pausa e assicurare che tutti i thread vengano sospesi correttamente quando necessario.

3. **Gestione del Ciclo di Vita**: Ogni virtual thread esegue un loop continuo, aggiornando lo stato del boid e dormendo per 40 millisecondi per simulare un framerate di 25 FPS. I thread possono essere interrotti in qualsiasi momento, e il loro ciclo di vita è gestito tramite un monitor dedicato.

2.4.1 Il Comportamento dei Virtual Threads

I virtual threads vengono creati utilizzando il costruttore `Thread.ofVirtual().start()` per ogni boid. Il comportamento di ciascun thread è simile a quello dei thread tradizionali, con la differenza che i virtual threads consumano meno risorse e possono gestire un numero significativamente maggiore di boid contemporaneamente senza degradare le prestazioni. Ogni boid viene gestito da un `virtual thread`, che aggiorna la velocità e la posizione del boid in base alle regole del modello (separazione, allineamento e coesione). Inoltre come accennato in precedenza, ogni thread dorme per circa 40 millisecondi per simulare un framerate di circa 25 FPS.

2.4.2 Sincronizzazione e Gestione della Concorrenza

L'accesso alle risorse condivise, come il modello dei boid, è protetto da un monitor (`BoidMonitor`), che gestisce lo stato di pausa della simulazione. Quando la simulazione è in pausa, tutti i virtual threads sono sospesi utilizzando il metodo `waitIfPaused()` all'interno di un blocco `while` che attende il segnale di ripresa. La sincronizzazione viene realizzata con `ReentrantLock` e `Condition`, per garantire che i thread siano sospesi e ripresi in modo sicuro e coordinato.

2.4.3 Considerazioni sul Modello

Questo approccio offre numerosi vantaggi rispetto alla creazione di thread tradizionali, in particolare in termini di efficienza e scalabilità. I virtual threads permettono di gestire un numero significativamente maggiore di boid senza l'overhead associato ai thread tradizionali. Tuttavia, è importante considerare anche i seguenti limiti:

- **Semplicità nel controllo della concorrenza**: Il monitor semplifica notevolmente la gestione della concorrenza e lo stato di pausa della simulazione, ma può comunque esserci una piccola latenza nell'attesa dei thread, sebbene minimizzata rispetto ai thread tradizionali.
- **Limitazioni della piattaforma**: L'utilizzo di virtual threads è supportato solo nelle versioni più recenti di Java (io ad esempio uso lo versione 21 di JDK), quindi l'approccio non è compatibile con versioni precedenti della piattaforma.

In sintesi, l'uso dei virtual threads rappresenta un'ottima soluzione per simulazioni ad alta concorrenza, poiché consente di gestire un numero elevato di agenti senza sacrificare le prestazioni del sistema.

3 Rappresentazione tramite Reti di Petri

Per modellare il comportamento concorrente del sistema in maniera formale, è stata utilizzata una Rete di Petri. Questa rappresentazione consente di evidenziare i punti di sincronizzazione e i possibili conflitti.

3.1 Reti di Petri per le tre implementazioni

Rete di Petri per il caso *Multithread (Thread tradizionali)*

Descrizione:

Ogni boid è gestito da un thread dedicato che cicla autonomamente tra le fasi di calcolo e aggiornamento, con sincronizzazione diretta (`synchronized`) sull'accesso al modello condiviso.

Elementi:

- **Posti:**
  - P0 – Thread Boid pronto
  - P1 – Inizio calcolo movimento
  - P2 – Sezione critica (accesso sincronizzato al modello)
  - P3 – Aggiornamento posizione
  - P4 – Dormire (sleep per frame)
  - P5 – Attesa/Pausa
- **Transizioni:**
  - T0 – Avvio thread
  - T1 – Esecuzione regole boid
  - T2 – Accesso sincronizzato (synchronized block)
  - T3 – Calcolo e aggiornamento stato
  - T4 – Sleep
  - T5 – Ripresa da pausa

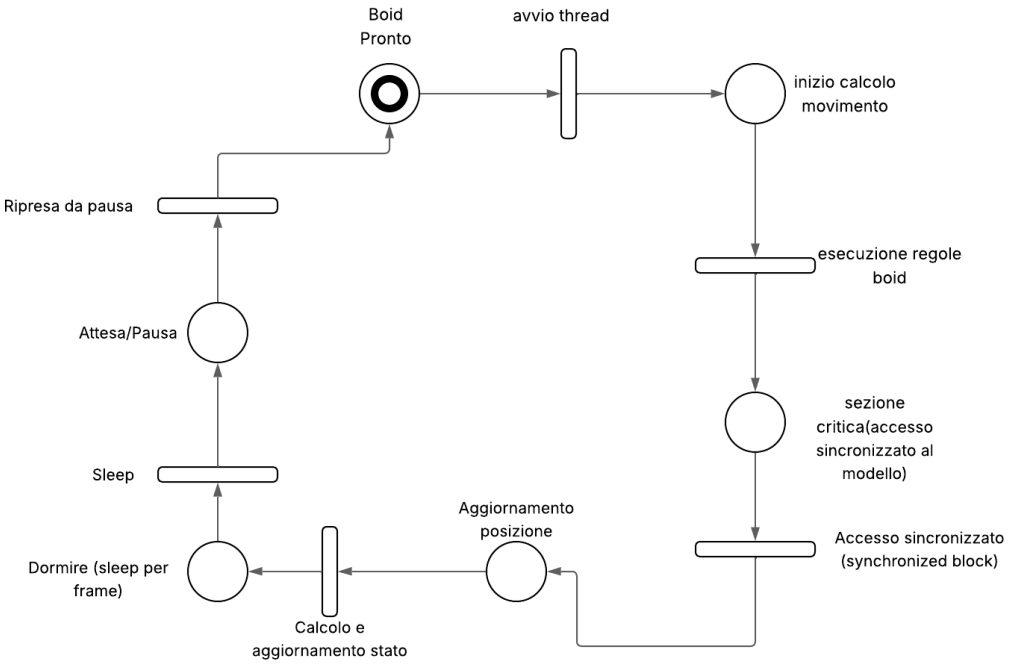

Figure 1: Modello di Rete di Petri per la simulazione dei boids

Rete di Petri per il caso *Executor Framework*

Descrizione:

Ogni boid è rappresentato da un `Runnable` (BoidTask) gestito da un `ExecutorService` con thread pool fisso. Il controllo della pausa è effettuato tramite `ReentrantLock` + `Condition`.

Elementi:

- **Posti:**
  - P0 – Task pronto
  - P1 – Pool disponibile
  - P2 – Lock acquisito
  - P3 – BoidTask in esecuzione
  - P4 – Task in pausa
  - P5 – Task completato (sleep)

- **Transizioni:**
  - T0 – Submit del task al pool
  - T1 – Lock.acquire()
  - T2 – WaitIfPaused
  - T3 – Calcolo stato boid
  - T4 – Sleep (simula 40 ms)
  - T5 – Lock.release()

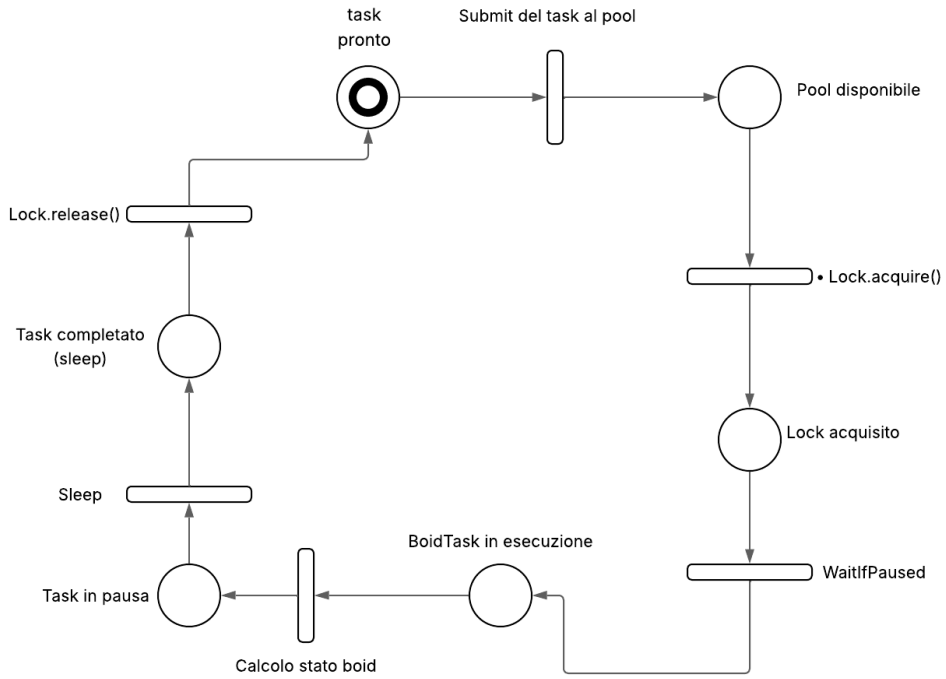

Figure 2: Modello di Rete di Petri per la simulazione dei boids

Rete di Petri per il caso *Virtual Threads (Project Loom)*

Descrizione:

Ogni boid è gestito da un *virtual thread* (`Thread.ofVirtual()`), molto leggero, con sincronizzazione tramite monitor e possibilità di sospensione/ripresa efficiente.

Elementi:

- **Posti:**

  - P0 – Virtual thread creato
  - P1 – Boid pronto al calcolo
  - P2 – Entrata nel monitor
  - P3 – Verifica pausa (`waitIfPaused`)
  - P4 – Aggiornamento stato boid
  - P5 – Dormire per framerate

- **Transizioni:**
  - T0 – Start virtual thread
  - T1 – Entrata in monitor
  - T2 – Verifica pausa
  - T3 – Calcolo e aggiornamento
  - T4 – Sleep (simulazione 25 FPS)
  - T5 – Ripresa

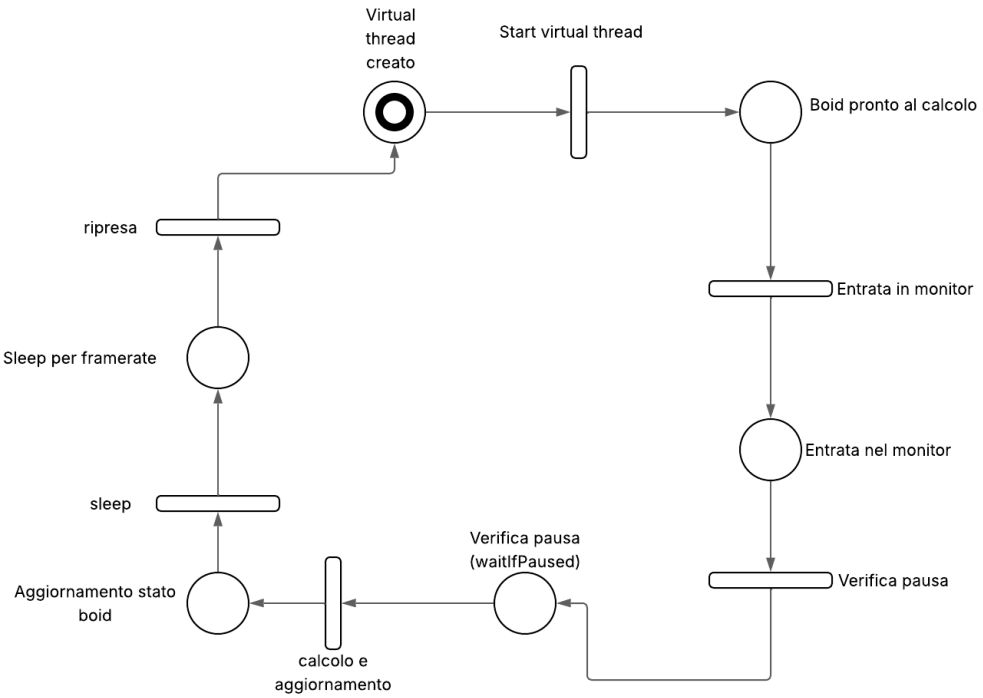

Figure 3: Modello di Rete di Petri per la simulazione dei boids

4 Test delle Prestazioni

L'analisi delle prestazioni si concentra sull'efficienza dell'implementazione concorrente nelle tre varianti (Multithread, ExecutorService e Virtual Threads), valutando principalmente il carico del sistema, la latenza e l'overhead associato alla creazione dei thread.

4.1 Casi di Test

Sono stati eseguiti diversi scenari di test per misurare l'impatto delle varie tecniche di concorrenza sulle prestazioni complessive della simulazione:

- **Test 1:** Simulazione con 100 boid, eseguita per 5 minuti, per valutare il consumo delle risorse di sistema (CPU, memoria) e la latenza nell'aggiornamento dei boid.
- **Test 2:** Simulazione con 1000 boid, eseguita per 10 minuti, per analizzare la scalabilità delle implementazioni.
- **Test 3:** Simulazione con 5000 boid, eseguita per 10 minuti, per testare l'efficienza della versione con thread virtuali.

4.2 Risultati dei Test

I risultati sono stati raccolti monitorando l'uso della CPU, la latenza media per aggiornare la posizione dei boid e il tempo di rendering grafico.

```
Prestazioni:

VT                                      MT                                      EXECUTOR
Simulazione avviata con 100 boids…      Simulazione avviata con 100 boids…      Simulazione avviata con 100 boids…
Simulazione completata in 56 ms         Simulazione completata in 36 ms         Simulazione completata in 35ms

Simulazione avviata con 500 boids…      Simulazione avviata con 500 boids…      Simulazione avviata con 500 boids…
Simulazione completata in 71 ms         Simulazione completata in 94 ms         Simulazione completata in 115 ms

Simulazione avviata con 1000 boids…     Simulazione avviata con 1000 boids…     Simulazione avviata con 1000 boids...
Simulazione completata in 77 ms         Simulazione completata in 146 ms        Simulazione completata in 188 ms
```

Figure 4: Risultati dei test delle prestazioni

4.2.1 Multithreading con Thread Tradizionali

In questa versione, l'utilizzo della CPU aumenta in modo lineare con il numero di boid. Ogni boid è gestito da un thread separato, quindi l'overhead di gestione dei thread cresce significativamente per grandi numeri di boid.

Risultati principali:

- Elevato carico sulla CPU per simulazioni con più di 100 boid.
- Maggiore latenza nell'aggiornamento della posizione dei boid quando il numero di agenti cresce.

4.2.2 ExecutorService

L'uso di un `ExecutorService` migliora significativamente la gestione dei thread, riducendo il carico sulla CPU. Con un pool di thread fisso, il sistema è in grado di gestire un numero maggiore di boid con un overhead inferiore rispetto alla versione con thread tradizionali.

Risultati principali:

- Migliore gestione delle risorse rispetto alla creazione di thread separati per ogni boid.
- Latenza inferiore rispetto alla versione con thread tradizionali.

4.2.3 Thread Virtuali

I thread virtuali, che sono molto più leggeri rispetto ai thread tradizionali, hanno permesso una gestione estremamente scalabile. Con 5000 boid, il sistema ha mantenuto una bassa latenza e un carico CPU significativamente inferiore rispetto agli altri approcci.

Risultati principali:

- Scalabilità eccellente: la latenza e il carico della CPU sono rimasti costanti anche con un numero molto elevato di boid.
- Maggiore efficienza nell'uso delle risorse, senza sacrificare le prestazioni.

5 Verifica con Java PathFinder (JPF)

La simulazione dei boid, in particolare nella versione multithread, è stata testata utilizzando **Java PathFinder (JPF)**, un framework per la verifica formale dei programmi Java.
Tutto cio' è stato reso possibile attraverso l'utilizzo di **Gradle** attraverso i suoi task in particolare a quello chiamato *Verification* dove al suo interno sono inclusi tutti check necessari per controllare il progetto se viene eseguito senza problemi.

5.1 Obiettivo della Verifica

L'obiettivo principale della verifica era testare la correttezza dell'implementazione concorrente, in particolare per quanto riguarda:

- La corretta sincronizzazione tra i thread.
- L'assenza di condizioni di race.
- Il corretto aggiornamento dello stato del modello.

5.2 Setup di JPF

La configurazione di JPF ha incluso la definizione di stati critici e transizioni all'interno del programma. I principali test sono stati effettuati per verificare le seguenti proprietà:

- **Mutua Esclusione**: Verifica che nessun boid acceda contemporaneamente a risorse condivise, come la lista dei boid.
- **Correttezza del Ciclo di Simulazione**: Assicurarsi che ogni boid venga aggiornato correttamente senza errori o conflitti di stato.

5.3 Risultati della Verifica

La simulazione è stata testata per verificare che non vi fossero deadlock, race condition o altre anomalie. I risultati hanno mostrato che la versione multithread era ben protetta contro le condizioni di race, grazie all'uso di sincronizzazione sui dati condivisi.

Conclusioni della verifica con JPF:

- Il sistema è stato verificato per la correttezza, e non sono stati trovati errori relativi alla sincronizzazione.
- La gestione della concorrenza è stata correttamente implementata, senza alcuna condizione di race.

6 Conclusioni e Sviluppi Futuri

In questo assignment sono stati esplorati tre approcci per la simulazione concorrente dei boid: multithreading con thread tradizionali, utilizzo di un thread pool tramite `ExecutorService` e l'adozione di thread virtuali(`Virtual Threads`).
I risultati hanno mostrato che, sebbene tutte le tecniche possano essere utilizzate per creare una simulazione efficace, l'approccio basato sui thread virtuali ha offerto il miglior compromesso tra prestazioni e scalabilità, consentendo la gestione di un numero molto elevato di boid senza un significativo aumento del carico di sistema.
Le principali conclusioni della ricerca sono:

- L'approccio multithread tradizionale è efficace per piccoli numeri di boid, ma non scala bene quando il numero di agenti aumenta.
- L'uso dell'ExecutorService migliora la gestione delle risorse e riduce l'overhead, ma presenta ancora dei limiti nella gestione di grandi quantità di boid.
- I thread virtuali offrono la migliore scalabilità e l'efficienza delle risorse, senza compromettere le prestazioni.

Per sviluppi futuri, si potrebbero esplorare altre tecniche di ottimizzazione, come l'uso di algoritmi di ricerca spaziale per ridurre il numero di interazioni necessarie tra boid, o l'integrazione di altre tecnologie di concorrenza avanzata.