Programmazione concorrente e distribuita

Assignment 01 : Boids

Ines Fraccalvieri - ines.fraccalvieri@studio.unibo.it

April 13, 2025

Contents

1 Analisi del problema

Il progetto si è concentrato sulla parallelizzazione dell'algoritmo Boids, un modello computazionale che simula il comportamento collettivo di entità autonome (i boid). In particolare, l'attenzione è stata rivolta alla parallelizzazione delle fasi di aggiornamento delle velocità e delle posizioni dei boid.

Queste due operazioni sono strettamente interdipendenti: il calcolo della nuova posizione di ciascun boid richiede l'accesso alle sue velocità attuali, pertanto si rende necessaria una forma di sincronizzazione per evitare race conditions tra i thread che leggono e scrivono questi dati.

In parallelo, vi è anche l'aggiornamento della vista grafica, che necessita esclusivamente della lettura delle posizioni dei boid. Questo permette di eseguire l'aggiornamento della view contemporaneamente al calcolo delle nuove velocità, ma non in concomitanza con la modifica delle posizioni, per non incorrere in letture incoerenti.

Nella versione sequenziale dell'algoritmo, i boid vengono aggiornati uno alla volta e in ordine, senza effettuare una copia consistente dello stato globale del sistema all'inizio di ogni iterazione. Questo significa che ogni boid può basarsi, per il proprio aggiornamento, su dati parzialmente aggiornati di altri boid. Tale comportamento non compromette significativamente la correttezza nel contesto sequenziale, ma diventa problematico in uno scenario concorrenziale.

Per confrontare diversi approcci alla programmazione concorrente, sono state sviluppate tre implementazioni distinte dell'algoritmo parallelo:

- **Platform Threads**: utilizza i thread nativi del sistema operativo per la parallelizzazione.
- **Task-based**: adotta un approccio basato su task sfruttando le API del Java Executor Framework.
- **Virtual Threads**: impiega i nuovi virtual thread introdotti in Java, che offrono un'alternativa leggera ai thread tradizionali.

2 Architettura

Ogni versione implementata dell'algoritmo si presta a differenti strategie di parallelizzazione, tuttavia la logica di sincronizzazione rimane sostanzialmente analoga. È stato adottato un modello MVC (Model-View-Controller).

Quando l'utente interagisce con l'interfaccia grafica, ad esempio premendo uno dei pulsanti, la View invia una notifica al Controller, quale aggiorna i comandi di controllo associati. In particolare, sono state definiti comandi distinti per gestire le operazioni di pause/resume della simulazione e per lo stop.

Questi valori vengono interrogati ciclicamente sia dai thread worker – per decidere se continuare l'elaborazione – sia dal thread principale, che ne

controlla lo stato per gestire eventuali richieste di stop ed eventuamente un nuovo start della simulazione.

Il comportamento dei pulsanti è il seguente:

- **Pause/Resume**: quando la simulazione è sospesa, i worker non possono iniziare una nuova iterazione. Il pulsante di reset viene attivato e il thread principale si occupa di controllare se l'utente decide di ripristinare la simulazione.

- **Stop**: In questo stato, i worker vengono terminati, e svuotata la lista dei boid. Nel caso in cui si volesse far ripartire la simulazione bisognerà inserire il nuovo numero di boids che vogliamo visualizzare e cliccare sul tasto di start. In questo caso verranno creati i nuovi boids e successivamente divisi nelle varie sottoliste dei worker.

In tutte le implementazioni è stata posta particolare attenzione alla gestione delle operazioni concorrenti. Per evitare conflitti tra letture e scritture concorrenti, le fasi di lettura e scrittura sono state separate e serializzate. In particolare, l'inizio di una nuova fase è subordinato al completamento della fase precedente da parte di tutti i thread. Questo approccio garantisce la consistenza dello stato condiviso e semplifica il coordinamento tra i thread.

Ogni iterazione del comportamento di un boid è suddivisa nelle seguenti fasi:

1. Calcolo della nuova velocità.

2. Scrittura della velocità calcolata.

3. Attesa che tutti i boids abbiano calcolato la propria velocità.

4. Calcolo e aggiornamento della nuova posizione

5. Aggiornamento della GUI

Questa struttura garantisce che le operazioni siano eseguite in maniera ordinata e coerente, riducendo il rischio di race condition e migliorando la stabilità complessiva della simulazione concorrente.

2.1 Multi-Thread

In questa versione dell'algoritmo sono stati creati N thread, corrispondenti al numero di core disponibili sul sistema. Ogni thread è responsabile dell'elaborazione di un sottoinsieme di boid, che vengono aggiornati in parallelo durante ciascuna iterazione della simulazione.

Per garantire una corretta sincronizzazione tra le varie fasi computazionali, è stato adottato l'uso di BoidsLatch. Ogni fase è associata a un latch specifico, che consente di coordinare l'avanzamento dei thread: ogni worker decrementa il latch al termine della propria elaborazione, e il thread principale (o gli altri worker) attende il completamento della fase attraverso il metodo await(), bloccandosi finché il conteggio non arriva a zero.

Le fasi sincronizzate sono le seguenti:

Latch per il calcolo e aggiornamento delle velocità: garantisce che tutti i boid abbiano completato il calcolo delle nuove velocità prima di procedere ed assicura che la scrittura delle nuove velocità sia completata da tutti i thread prima di iniziare a calcolare le nuove posizioni. Latch per l'aggiornamento delle posizioni: sincronizza l'ultima fase computazionale, impedendo l'aggiornamento della GUI fino al completamento della scrittura delle nuove posizioni da parte di tutti i thread.

Una volta che tutte le velocità e posizioni sono state aggiornate, viene avviato l'aggiornamento della GUI. Durante questa fase, i thread sono bloccati e non possono iniziare una nuova iterazione. Solo al termine dell'aggiornamento grafico possono accedere di nuovo allo stato condiviso per avviare il passo successivo della simulazione.

Il controllo dell'esecuzione viene gestito tramite una variabile di stato, che consente di sospendere o riprendere il ciclo di aggiornamento. Quando viene premuto il pulsante Pause, i thread si arrestano prima di iniziare una nuova iterazione, attendendo che venga ripristinata su Resume. Nel caso venga premuto il pulsante Stop, i thread vengono terminati e viene pulita la lista dei boid. Nel caso in cui si voglia far ripartire la simulazione, viene creata una nuova lista di boid con dimensione aggiornata. Di conseguenza, anche i thread vengono ricreati e riassegnati ai nuovi dati.

L'implementazione del meccanismo di sincronizzazione si basa su un latch riutilizzabile, progettato per coordinare l'avanzamento dei thread in modo

semplice ed efficace. A differenza del classico CountDownLatch fornito da Java, che non può essere riutilizzato, questo latch personalizzato si resetta automaticamente a ogni iterazione, rendendolo adatto a scenari ciclici come quelli di una simulazione. Questa scelta consente di evitare i rischi e la complessità legati alla gestione manuale di barriere cicliche, garantendo al tempo stesso chiarezza, affidabilità e riutilizzabilità nella logica concorrente. Pur introducendo una leggera complessità nell'implementazione rispetto a un latch monouso, offre un migliore bilanciamento tra prestazioni ed eleganza del codice in contesti iterativi.

2.2 Task based

Questa implementazione ha come obiettivo quello di ottimizzare l'elaborazione delle due fasi principali della simulazione tramite l'utilizzo di ExecutorService:

- L'aggiornamento della velocità di ciascun boid.
- L'aggiornamento della sua posizione.

Per farlo, la simulazione divide i boid in sottoinsiemi (batch) e li assegna a thread separati, gestiti tramite due `ExecutorService` distinti: uno per il calcolo delle velocità e uno per l'aggiornamento delle posizioni.

L'utilizzo degli executor consente di astrarre e semplificare la creazione, l'assegnazione e la terminazione dei thread. Il metodo `Executors.newFixedThreadPool(n)`

crea un pool di n thread, che rimangono attivi in attesa di nuovi compiti (task) da eseguire. In questo caso, il numero di thread è calcolato dinamicamente in base al numero di core disponibili (Runtime.getRuntime().availableProcessors() - 1), per sfruttare al massimo la parallelizzazione, lasciando però un core libero per evitare il sovraccarico del sistema.

L'utilizzo dei Task Executor offre diversi vantaggi rispetto alla gestione manuale dei thread:

- **Semplicità e leggibilità**: il codice è più compatto e facilmente gestibile.
- **Efficienza**: i thread vengono riutilizzati dal pool, riducendo l'overhead di creazione/distruzione.
- **Controllo**: è possibile gestire facilmente la pausa, la ripresa o l'arresto della simulazione tramite metodi sincroni e meccanismi di interruzione.

Anche in questa implementazione viene data la possibilità di mettere in pausa la simulazione, grazie all'utilizzo di meccanismi sopra descritti. Inoltre, è possibile interrompere completamente la simulazione, che arresta i thread attivi e rimuove i boid dal modello. Una volta fermata, la simulazione può essere riavviata con un nuovo insieme di boid, anche con un numero diverso rispetto all'iniziale, rendendo il sistema flessibile e facilmente riutilizzabile per test e scenari differenti.

2.3 Virtual Thread

Anche nell'implementazione BoidsVirtualThreadSimulator, la simulazione del comportamento dei boid viene gestita in modo concorrente, ma in questo caso si sfrutta il supporto dei virtual thread. I virtual thread sono una forma leggera di thread gestita direttamente dalla JVM, questo li rende particolarmente adatti a simulazioni agent-based, come quella dei boid, in cui ogni unità può essere elaborata in modo indipendente.

Alla base della simulazione troviamo, come nella versione precedente, due fasi principali: il calcolo della nuova velocità per ogni boid e il successivo aggiornamento della sua posizione. Per gestire in parallelo queste operazioni, la lista dei boid viene suddivisa in gruppi (batch), calcolando dinamicamente il numero di elementi per ogni gruppo in base al numero di core della macchina (boidsWorkers).

Per ogni batch viene avviato un virtual thread tramite `Thread.ofVirtual().start(...)`, che entra in un ciclo continuo nel quale esegue l'aggiornamento dei boid del proprio gruppo. All'interno del ciclo, i virtual thread attendono eventuali pause tramite il metodo isPaused(), che sfrutta wait() per sospendere il thread fino a quando la simulazione non viene ripresa con notifyAll().

L'aggiornamento della velocità e della posizione viene coordinato tramite un oggetto BoidsLatch, permettendo ai thread di sincronizzarsi dopo aver terminato ciascuna fase. In pratica, tutti i thread devono completare l'aggiornamento

della velocità prima di poter iniziare a modificare la posizione, garantendo la coerenza temporale della simulazione.

Dal punto di vista del controllo dell'esecuzione, anche questa implementazione fornisce strumenti per mettere in pausa, interrompere e riavviare la simulazione in modo dinamico.

In conclusione, questa versione della simulazione evidenzia l'efficacia dei virtual thread nella gestione di applicazioni fortemente parallele, mantenendo al contempo un alto livello di controllo sull'esecuzione. Il risultato è un sistema reattivo, scalabile e adattabile, in cui è possibile sperimentare con semplicità diverse configurazioni e strategie comportamentali.

3 Rete di Petri

La rete di Petri illustrata rappresenta il flusso logico della simulazione dei Boids implementata in un ambiente concorrente. La simulazione è strutturata in due fasi principali: aggiornamento della velocità e aggiornamento della posizione di ciascun boid. A queste si aggiungono meccanismi di controllo che permettono di mettere in pausa, riprendere o arrestare la simulazione.

Ogni thread inizia dalla transizione updateVelocity, che rappresenta il calcolo della nuova direzione/movimento del boid in base al comportamento degli altri (regole di allineamento, coesione e separazione). Dopo aver aggior-

nato la velocità, ogni thread deposita un token nel posto `Update Velocity`, dopo la sincronizzazione, i boid possono passare alla fase `Update Position`, che aggiorna le coordinate spaziali di ciascun boid nel modello.

La transizione `Pause Simulation` sposta un token da uno stato attivo a Paused, bloccando l'esecuzione e i thread restano fermi in attesa.

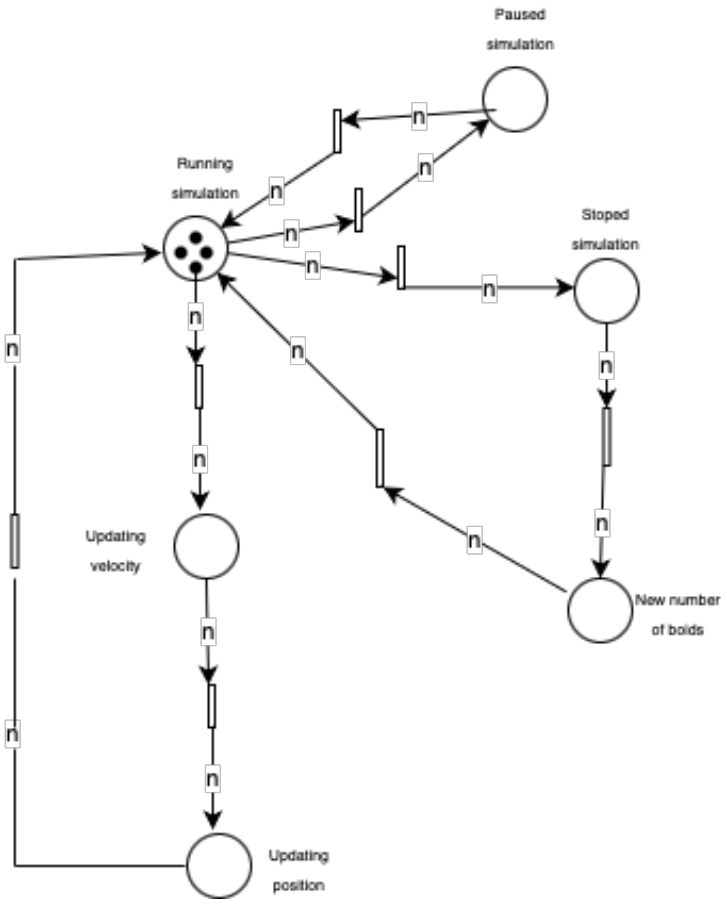

Figure 1: Rete di Petri del modello multi-thread

4 Prestazioni

L'obiettivo è quello di confronatr le prestazioni delle varie implementazioni del simulatore di boids in base al numero di frame generati al secondo (FPS) e al tempo medio di calcolo per frame (ms/frame).

Table 1: Confronto delle prestazioni tra diverse strategie di simulazione

Modalità	**#Boids**	**Avg. Frame Time (ms)**	**Avg. FPS**	**Total Frames**
Sequenziale	100	0.417	2397.93	3242
	2000	35.654	28.05	2897
	5000	127.469	7.85	974
Multi-thread	100	0.032	31412.16	4649
	2000	0.051	19692.31	3328
	5000	0.059	16948.98	3322
Executor (Task)	100	0.537	1860.70	3286
	2000	6.836	146.29	3368
	5000	36.723	27.23	1690
Virtual Threads	100	0.028	36032.97	3279
	2000	0.058	17295.70	3217
	5000	0.058	17230.37	3291

- **Sequenziale**: Le prestazioni calano drasticamente all'aumentare del numero di boid. Oltre i 2000 boids il frame rate diventa troppo basso

per una simulazione fluida.

- **Multi-thread**: Mostra prestazioni eccellenti e molto stabili, con un incremento lineare del carico ben distribuito tra i thread. È tra le soluzioni migliori in termini di efficienza.

- **ExecutorService**: Risulta meno efficiente del multi-thread puro, probabilmente a causa del costo di scheduling dei task e della gestione dei thread da parte del pool. È comunque una soluzione valida fino a carichi medi (es. 2000 boids).

- **Virtual Threads**: Ottimi risultati, simili (e in alcuni casi migliori) del multi-thread, con una gestione del carico molto leggera e scalabile. È la soluzione con il miglior bilanciamento tra semplicità del codice e prestazioni, soprattutto con carichi elevati.

Per concludere, quindi, per le simulazioni leggere, tutte le implementazioni sono accettabili, ma virtual threads e multi-threading offrono prestazioni estremamente superiori. Per carichi pesanti (2000+ boids), l'approccio sequenziale non è sostenibile. Virtual threads emergono come la scelta migliore in termini di prestazioni e scalabilità. L'ExecutorService è un buon compromesso per codice ordinato, ma meno performante rispetto al multi-thread diretto o virtual threads.

5 Verifica con l'utilizzo di JPF

Per assicurarci che non fossero presenti errori legati alla concorrenza, come deadlock o starvation, nella prima parte del progetto, abbiamo utilizzato il template di Java PathFinder fornito dal professor Aguzzi. Per rendere il codice compatibile con JPF, è stato necessario astrarre i concetti fondamentali, mantenendo esclusivamente la logica di simulazione, ed eliminando la componente grafica, non rilevante ai fini dell'analisi formale.

Il sistema è stato verificato in termini di correttezza, e non sono stati rilevati errori di sincronizzazione. La gestione della concorrenza risulta implementata correttamente, senza la presenza di condizioni di race o altre anomalie legate all'accesso concorrente alle risorse condivise.

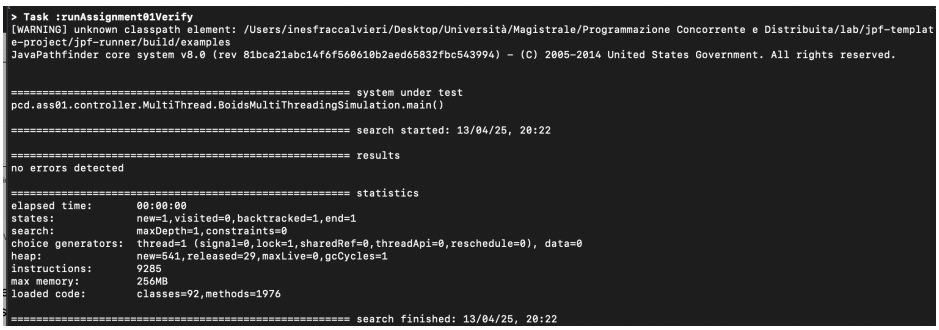

Figure 2: Risultati verifica jpf