Assignment01

Giacomo Foschi
Matricola: 0001179137
Email: giacomo.foschi3@studio.unibo.it

Giovanni Pisoni
Matricola: 0001189814
Email: giovanni.pisoni@studio.unibo.it

Giovanni Rinchiuso
Matricola: 0001195145
Email: giovanni.rinchiuso@studio.unibo.it

Gioele Santi
Matricola: 0001189403
Email: gioele.santi2@studio.unibo.it

Capitolo 1 - Analisi del problema

Il progetto si basa sulla simulazione dei boids, ideata da Craig Reynolds nel 1986. Ogni boid (agente) si muove seguendo tre regole principali: separazione, allineamento e coesione, interagendo con gli altri boid in modo da emulare comportamenti di gruppi di uccelli o pesci. L'obiettivo del progetto consisteva nel passare da una simulazione in versione sequenziale a una versione concorrente. Questo ha comportato diverse sfide:

- **Comunicazione tra i boid e l'ambiente**: un problema fondamentale è stato far comunicare i boid con l'ambiente, senza che modificassero lo stato durante la raccolta delle informazioni. La sincronizzazione tra i boid è stata necessaria per garantire che i dati non venissero modificati mentre i boid li leggevano, usando meccanismi di sincronizzazione come monitor e barriere cicliche.
- **Il numero dei thread**: è stato cruciale capire come suddividere l'intero gruppo di boids nei diversi modelli implementativi. Abbiamo quindi optato per suddividere i boids in gruppi (o *pool*) nelle versioni multithreading e task-based, per evitare overhead e appesantimento del sistema; al contrario, nella versione virtual-thread viene creato un nuovo thread per ogni boid, grazie alla leggerezza e scalabilità di questi, che ne permette l'utilizzo in un numero elevato.
- **La GUI**: La GUI, che consente di avviare, fermare e mettere in pausa la simulazione, ha comportato sfide nella concorrenza, poiché diversi thread possono accedere ai dati condivisi. È stato necessario implementare un sistema di sincronizzazione per evitare *race condition* e per garantire la possibilità di mettere in pausa il programma e controllare correttamente con slider e pulsanti i parametri di esecuzione.
- **Sincronizzazione e gestione dei thread**: Per garantire la coerenza dei dati, è stato necessario implementare un sistema di sincronizzazione robusto. L'accesso concorrente alle risorse è stato gestito usando meccanismi come barriere e futures, che hanno assicurato che i thread non sovrascrivessero i dati contemporaneamente.

Il progetto è stato sviluppato in tre versioni concorrenti:

1. **Multithreading tradizionale**: in cui ogni thread gestisce un gruppo di boid, utilizzando sincronizzazione per evitare conflitti attraverso costrutti basici come *monitor* e *barrier*.
2. **Task-executor**: in cui i thread vengono gestiti come task, ottimizzando la gestione della concorrenza, sfruttando le potenzialità del Java Executor Framework, che coordina l'utilizzo e la sospensione dei thread disponibili.
3. **Virtual Threads**: in cui si utilizzano i Virtual Threads di Java, che riducono significativamente l'overhead grazie alla loro leggerezza e scalabilità anche in numero elevato.

In sintesi, la transizione dalla simulazione sequenziale a una versione concorrente ha richiesto l'uso di tecniche avanzate di sincronizzazione per gestire i boid e la GUI in modo sicuro, ottenendo una simulazione reattiva ed efficiente.

Capitolo 2 - Strategia risolutiva e architettura

Il progetto è stato implementato secondo il pattern MVC (Model-View-Controller) in modo tale da separare la logica di simulazione, la gestione dell'interfaccia utente e il controllo dell'applicazione. Si è scelta questa struttura in modo tale da avere un'architettura modulare e facilmente estendibile. I principali componenti sono:

- **Boid**: rappresenta un singolo boid, il quale ha una posizione e una velocità, che vengono aggiornate ad ogni iterazione della simulazione secondo tre regole fondamentali: separazione, allineamento e coesione.
- **BoidsModel**: ha il compito di gestire l'ambiente e le regole della simulazione, mantenendo la lista dei boid e regolando il loro comportamento collettivo attraverso i parametri di separazione, allineamento e coesione (modificabili da interfaccia). Inoltre, si occupa dell'inizializzazione dei boid, assegnando loro posizioni e velocità casuali. Il suo ruolo all'interno della nostra architettura è quello di fornire il contesto e le regole affinché il comportamento emergente dello stormo si sviluppi in modo realistico.
- **BoidsController**: funge da controller della simulazione, gestendo l'interazione tra il modello (BoidsModel) e la vista (BoidsView). È responsabile dell'avvio, della pausa e dell'arresto della simulazione, oltre a coordinare l'aggiornamento della vista in base al framerate. Utilizza un numero di thread proporzionale ai core disponibili per garantire prestazioni ottimali. Il suo ruolo principale è quello di coordinare l'esecuzione della simulazione, mantenendo il controllo dello stato e sincronizzando gli aggiornamenti del sistema.
- **BoidsView**: gestisce l'interfaccia grafica della simulazione, permettendo agli utenti di visualizzare e controllare il comportamento dello stormo. Fornisce una finestra con un pannello di simulazione, pulsanti per avviare, mettere in pausa e interrompere la simulazione, e slider per modificare in tempo reale i parametri di separazione, allineamento e coesione. Inoltre, aggiorna dinamicamente la visualizzazione in base al framerate corrente. Il suo ruolo principale è rendere interattiva la simulazione, permettendo agli utenti di osservare e influenzare l'evoluzione del sistema.

Gestione Multi-Thread

Il primo approccio risolutivo proposto è quello multithreading, tecnica che sfrutta il parallelismo per migliorare le performance di un'applicazione, particolarmente utile in contesti computazionalmente intensivi come le simulazioni. In questo approccio i compiti vengono suddivisi in thread (fisici) separati che vengono eseguiti simultaneamente, consentendo così un'elaborazione parallela dei dati.
Nell'ambito della simulazione dei boids, il multithreading viene impiegato per suddividere il calcolo del movimento e delle interazioni tra i boids tra diversi thread, ognuno dei quali gestisce un sottoinsieme di boids (*pool*). Una gestione multithread riduce i tempi di calcolo e aumenta l'efficienza del sistema, soprattutto quando il numero di boids cresce significativamente. La sincronizzazione tra i thread è gestita tramite barriere cicliche, che assicurano che tutti i thread completino un passo di calcolo prima di procedere al successivo, evitando inconsistenze nei dati.

I principali componenti e le loro interazioni sono descritti di seguito:

- **MultithreadingController**: estende BoidsController e gestisce la simulazione dei boids utilizzando il multithreading. Suddivide il gruppo di boids in pool, ognuno dei quali viene gestito da un thread separato, tramite l'utilizzo di BoidThread. La sincronizzazione tra i thread viene gestita tramite una barriera ciclica personalizzata (CustomCyclicBarrier) e un monitor

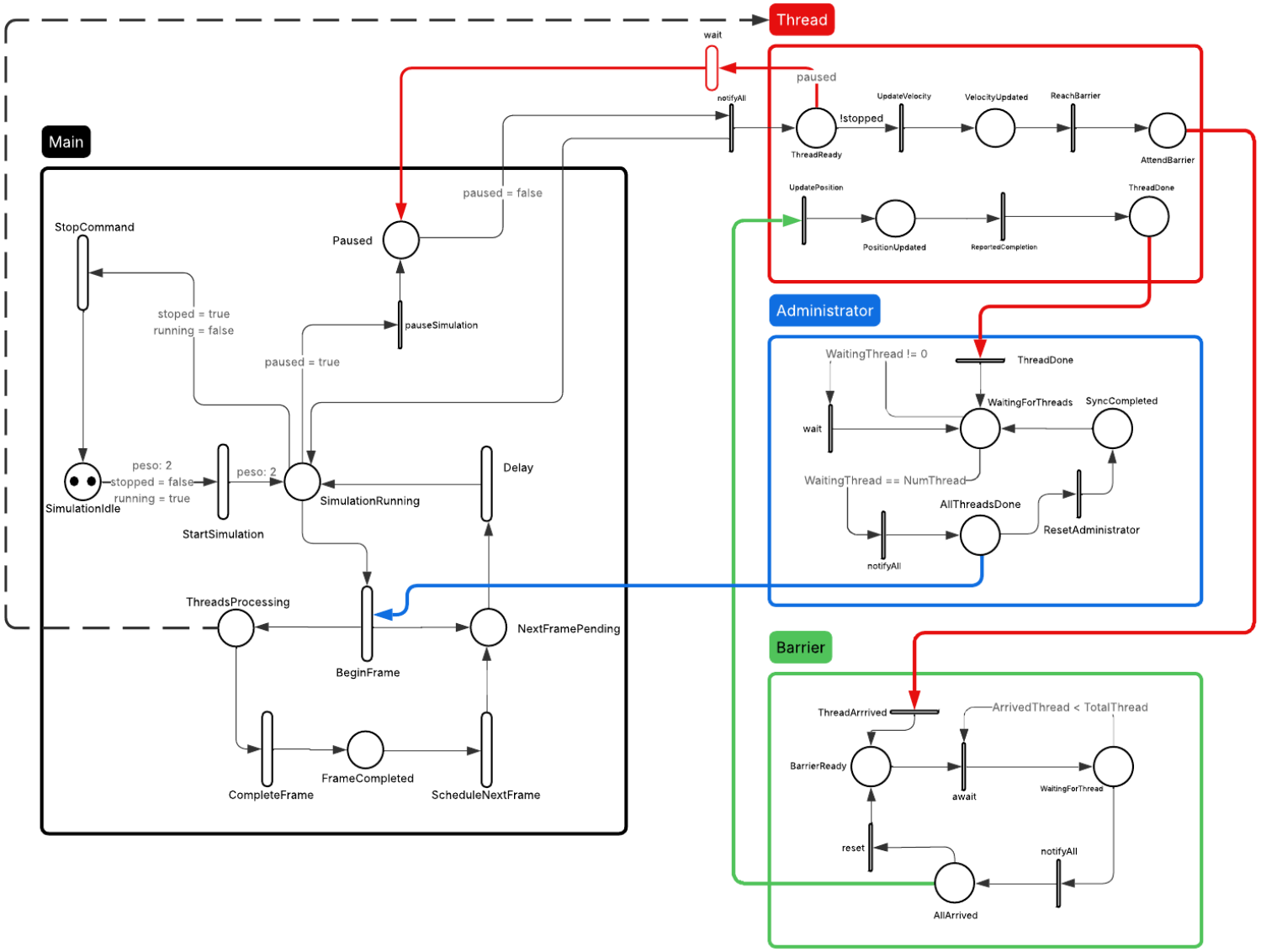
(MultiAdministrator), che coordina il completamento delle operazioni prima di procedere alla fase successiva. I thread calcolano separatamente l'aggiornamento delle velocità e delle posizioni dei boids, migliorando l'efficienza del calcolo parallelo. Inoltre, la classe offre funzionalità per avviare, fermare e riprendere la simulazione, gestendo la logica di pausa e ripresa in modo sincrono.

- **BoidThread**: gestisce l'aggiornamento parallelo del comportamento dei boid in una simulazione multithreading. Estende Thread rappresentando un thread di lavoro che aggiorna lo stato di un sottoinsieme di boid. Ogni istanza della classe è responsabile di un sottoinsieme di boid, aggiornandone la velocità e la posizione in modo sincronizzato grazie a una barriera ciclica (CustomCyclicBarrier). Interagisce con il MultithreadingController per rispettare pause e riprese della simulazione. Il suo scopo principale è migliorare l'efficienza della simulazione, distribuendo il carico computazionale su più thread per aggiornare i boid in parallelo.
- **Administrator**: interfaccia di un monitor che definisce i metodi necessari per gestire la sincronizzazione di più thread in una simulazione multithreading. Fornisce le operazioni per segnalare quando un thread ha completato il proprio compito, per attendere che tutti i thread abbiano finito, e per notificare che tutti i thread sono terminati e il ciclo può ripartire. Il suo scopo principale è facilitare la gestione del flusso di esecuzione parallela, garantendo che i thread siano sincronizzati e lavorino in modo coordinato.
- **MultiAdministrator:** classe, che implementando l'interfaccia Administrator, gestisce la sincronizzazione dei thread fungendo da monitor, coordinando il loro avanzamento. Tiene traccia di quanti thread hanno completato il loro compito e li mette in attesa finché tutti non hanno terminato, evitando corse critiche. Una volta che tutti i thread hanno completato il ciclo di aggiornamento, li sblocca per procedere al passo successivo. Il suo ruolo principale è garantire un'esecuzione ordinata e sincronizzata della simulazione, migliorando la stabilità e l'efficienza del sistema multithread.
- **CustomCycleBarrier:** interfaccia che definisce i metodi per implementare una barriera ciclica, utilizzata per sincronizzare l'esecuzione di più thread. Il suo scopo principale è gestire la sincronizzazione tra i thread, permettendogli di lavorare in fasi coordinate e di ripartire ciclicamente.
- **CustomCyclicBarrierImpl**: classe, che implementando l'interfaccia CustomCycleBarrier**,** sincronizza l'esecuzione di più thread. Utilizzando un ReentrantLock e una Condition, gestisce l'attesa dei thread fino a quando tutti non raggiungono il punto di sincronizzazione, per poi sbloccarli simultaneamente. Questa classe consente un coordinamento preciso tra thread, utile per gestire operazioni in parallelo in modo sicuro e sincrono.

L'implementazione di questa versione di concorrenza è stata testata utilizzando Java Path Finder (JPF), un framework di verifica formale per programmi Java, utilizzato per l'esplorazione degli stati del programma e la verifica della correttezza del codice, con l'obiettivo di individuare eventuali errori di concorrenza, deadlock o violazioni di proprietà. Per eseguire la verifica, è stato utilizzato un modello semplificato del progetto, al fine di ottimizzare i tempi di esecuzione. In particolare, è stata rimossa la parte relativa all'interfaccia grafica e il numero di boids è stato ridotto per evitare un numero elevato di stati da esplorare. Questo ci ha consentito di verificare la corretta sincronizzazione e gestione degli accessi alle risorse, evitando deadlock e race-condition.

Petri Net:

Assumiamo di avere una CPU, avendo un numero di thread = numero CPU + 1, nella petri net simuliamo il funzionamento del sistema con 2 thread.



Gestione Task-based basata su Java Executor Framework

Nel contesto della simulazione dei boid, un approccio task-oriented basato su ExecutorService offre una soluzione efficace per gestire la concorrenza, suddividendo il carico di lavoro tra più thread in modo dinamico e strutturato. Questo modello si basa sulla creazione di task indipendenti, ciascuno dei quali esegue un'operazione specifica, come l'aggiornamento della velocità o della posizione di un pool di boid.

In particolare, viene utilizzato un executor di tipo CachedThreadPool, che permette di scalare dinamicamente il numero di thread in base al carico effettivo, riducendo al minimo l'overhead di creazione e gestione dei thread quando non necessario. L'esecuzione dei task avviene in modo asincrono e restituisce oggetti Future, i quali vengono utilizzati per attendere il completamento di tutti i task prima di procedere all'aggiornamento della visualizzazione. In questo modo si garantisce la sincronizzazione tra le fasi della simulazione, senza dover ricorrere a meccanismi di sincronizzazione manuale come latch o barriere.

Nel complesso, questo modello consente di ottimizzare le prestazioni della simulazione, sfruttando al meglio le capacità di calcolo disponibili senza incorrere nei problemi tipici della gestione manuale dei thread, come le race condition o i deadlock.

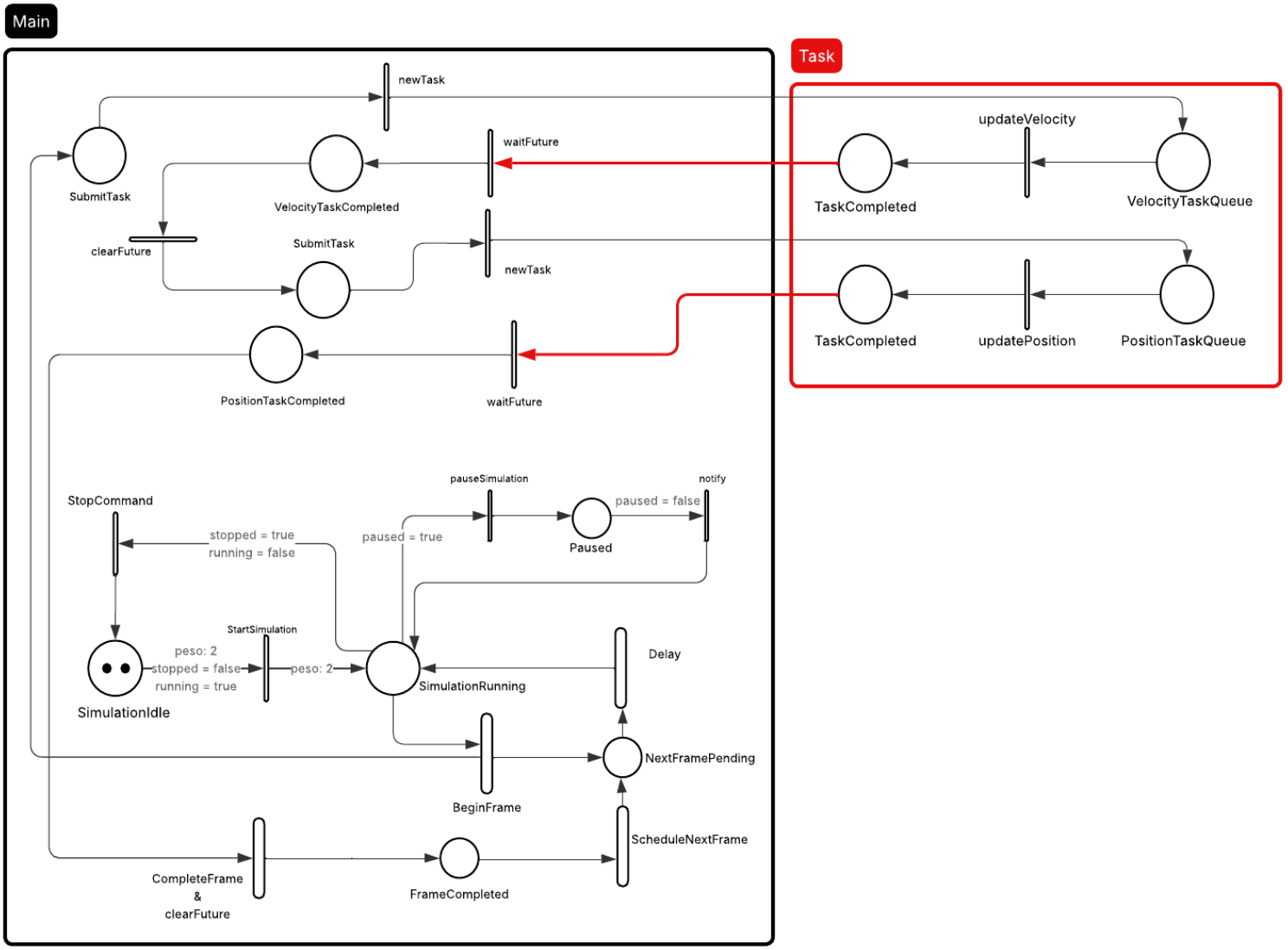
I principali componenti e le loro interazioni sono descritti di seguito:

- **TaskController**: estende BoidsController gestendo la simulazione dei boid sfruttando un pool di thread tramite ExecutorService e l'esecuzione asincrona di task con Future. La simulazione è suddivisa in due fasi distinte per ogni iterazione: aggiornamento delle velocità e aggiornamento delle posizioni. Utilizza un pool di thread (CachedThreadPool), creato con un ExecutorService, per eseguire in parallelo i calcoli di velocità e posizione dei boid, garantendo un aggiornamento efficiente del sistema. L'executor permette di gestire dinamicamente i thread senza doverli creare e distruggere manualmente, migliorando la gestione delle risorse. Il controller attende la conclusione di tutti i task tramite i future prima di passare alla fase successiva, garantendo così la sincronizzazione. Questo approccio migliora le prestazioni su macchine multicore mantenendo il controllo sull'esecuzione tramite metodi di pausa, ripresa e arresto. Inoltre, TaskController controlla lo stato della simulazione con meccanismi di sincronizzazione, permettendo di avviare, sospendere e fermare l'esecuzione in modo ordinato. Garantisce, inoltre, la corretta chiusura dell'executor per evitare sprechi di risorse.
- **UpdateTask**: classe che rappresenta un'astrazione per i task che aggiornano lo stato dei boid nella simulazione. Estende Callable<Void>, permettendo l'esecuzione asincrona dei task tramite un ExecutorService. È pensata per essere implementata da classi che definiscono operazioni specifiche, come l'aggiornamento delle velocità o delle posizioni dei boid. Con questo design si vuole consentire una gestione flessibile e modulare dei task concorrenti.
- **UpdatePositionTask**: classe che, implementando l'interfaccia UpdateTask, rappresenta un task eseguibile il cui scopo è aggiornare la posizione di un sottoinsieme di boid nella simulazione. Ogni istanza riceve una lista di boid e un riferimento al modello, e durante l'esecuzione itera sui boid assegnati, aggiornandone la posizione in base allo stato corrente del modello. Questo task viene eseguito all'interno di un ExecutorService e restituito come Future, consentendo al controller di sincronizzare l'esecuzione aspettando il completamento di tutti i task prima di procedere alla fase successiva della simulazione.

- **UpdateVelocityTask**: classe che rappresenta un task eseguibile con il compito di aggiornare la velocità di un sottoinsieme di boid nella simulazione. Il funzionamento è uguale a UpdatePositionTask.

Petri net:

Assumiamo di avere una CPU, avendo un numero di thread = numero CPU + 1, nella petri net simuliamo il funzionamento del sistema con 2 thread.



Gestione tramite Virtual-Thread

L'approccio basato sui Virtual Threads rappresenta una delle soluzioni più recenti ed efficienti per gestire l'esecuzione concorrente in Java, sfruttando una gestione leggera e altamente scalabile dei thread. Questo modello si distingue dai tradizionali thread fisici per la sua capacità di gestire un numero elevato di thread con un consumo di risorse ridotto, grazie alla gestione automatica da parte del runtime di Java.

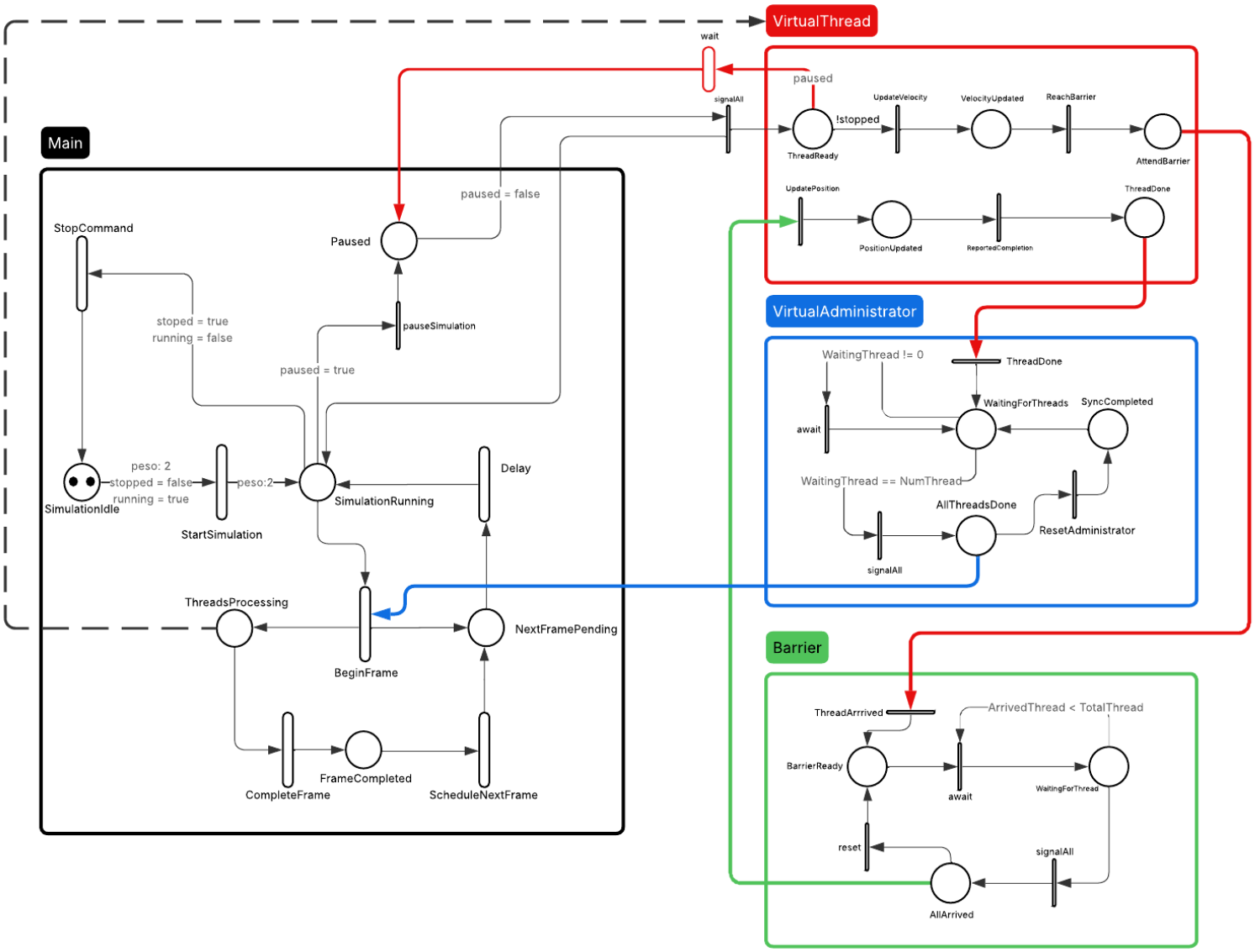
Nel contesto della simulazione dei boid, l'uso dei Virtual Threads permette di associare ogni boid a un thread virtuale dedicato, eseguendo i calcoli di aggiornamento (velocità e posizione) in parallelo. Questo approccio ottimizza l'utilizzo delle risorse di sistema e migliora la scalabilità, gestendo simulazioni con migliaia di boid in modo fluido ed efficiente. I Virtual Threads si rivelano vantaggiosi poiché riducono l'overhead dei tradizionali thread fisici, consentendo una gestione più efficiente dei thread con un impatto minimo sulle risorse, mantenendo alte prestazioni e riducendo la complessità della concorrenza grazie alla gestione automatica del runtime di Java.

Concludendo, l'uso dei Virtual Threads offre un approccio scalabile, leggero ed efficiente per la simulazione dei boid, rendendo possibile gestire un numero elevato di agenti (boid) senza compromettere le performance, migliorando al contempo la reattività e la sincronizzazione tra i vari task eseguiti in parallelo.

- **VirtualController**: classe, che estende BoidsController, gestisce la simulazione dei boid utilizzando virtual threads per ottimizzare l'esecuzione concorrente. Ogni boid è associato a un thread virtuale che viene gestito tramite una barriera ciclica per sincronizzare i calcoli. La classe utilizza un ReentrantLock e una Condition per controllare la pausa, il riavvio e l'arresto della simulazione in modo sicuro. I thread virtuali, più leggeri dei tradizionali, permettono una gestione efficiente di un gran numero di boid. La simulazione è coordinata dal VirtualAdministrator, che gestisce il ciclo di vita dei thread.
- **VirtualAdministrator**: classe che ha il compito di gestire la sincronizzazione dei thread virtuali fungendo da monitor nella simulazione. Utilizza un ReentrantLock e una Condition per coordinare l'esecuzione dei thread, assicurandosi che tutti completino il loro lavoro prima di procedere. La classe tiene traccia del numero di thread attivi e dei thread che hanno terminato il loro lavoro. La classe fornisce metodi per incrementare il contatore dei thread terminati, e per mettere in attesa i thread che hanno completato la propria esecuzione finché non sono tutti pronti. Una volta che tutti i thread sono terminati si resetta il contatore e segnala che la fase è completata.
- **BoidVirtualThread**: classe che rappresenta un thread virtuale associato a un boid nella simulazione. Ogni thread esegue in parallelo il calcolo della velocità e della posizione del proprio boid, utilizzando una barriera ciclica (CustomCyclicBarrier) per sincronizzarsi con gli altri thread. Durante l'esecuzione, il thread controlla se la simulazione è in pausa, e in tal caso, si mette in attesa fino a che non venga notificato di riprendere. Una volta che il thread ha aggiornato la velocità e la posizione del boid, segnala al Virtual Administrator che il suo lavoro è terminato. Il ciclo continua finché il thread non viene interrotto tramite il flag stopped.

Petri Net:

Assumiamo di avere una CPU, avendo un numero di thread = numero CPU + 1, nella petri net simuliamo il funzionamento del sistema con 2 thread.



Capitolo 3 - Analisi delle prestazioni

Per quanto riguarda l'analisi delle prestazioni, abbiamo effettuato dei log per registrare il tempo impiegato dai vari programmi nel completare 1000 iterazioni. Per iterazione intendiamo un ciclo completo di aggiornamento della velocità e della posizione dei Boid, seguito dal relativo aggiornamento grafico.
L'analisi è stata svolta su ogni versione del programma, variando il numero di Boid (1500, 3000, 4500). Infine, sono stati calcolati i relativi **Speed-Up** e l'**efficienza**, al fine di valutare il miglioramento delle prestazioni in funzione del parallelismo o dell'ottimizzazione applicata. Avendo a disposizione un computer con 12 core, considereremo tale dato per calcolare l'efficienza. I dati ottenuti sono frutto di molteplici run, di cui sono state prese in considerazione le migliori.

Per quanto riguarda il programma che implementata una soluzione sequenziale si è calcolato che impiega 4,1405673 * 10^7 nanosecondi (ns) con 1500 boids, 14,4988529 * 10^7 ns con 3000 boids, e 27,6871810 * 10^7 ns con 4500 boids.

La prima versione da noi implementata, quella che prevede una gestione dei boids multithread, impiega:

- 1,6762398 * 10^7 ns con 1500 boids, avendo quindi uno Speed-Up = $4,1405673 * 10^7\ ns\ /\ 1,6762398 * 10^7\ ns$ = **2,4702** e un'efficienza = 2,4702 / 12 = **0,2058**.
- Con 3000 boids impiega 4.0640799699 * 10^7 ns avendo quindi uno Speed-up = $14,4988529 * 10^7\ ns\ /\ 4.0640799699 * 10^7\ ns$ = **3,5676** e un'efficienza = 3,5676 / 12 = **0,2973**.
- Con 4500 boids impiega invece 9.0058076101 * 10^7 ns avendo quindi uno Speed-up = 27,6871810 * 10^7 ns / 9.0058076101 * 10^7 ns = **3,0744** e un'efficienza = 3,0743 / 12 = **0,2562**.

La seconda versione, che gestisce i boids con una struttura task based, impiega:

- 1,7446744 * 10^7 ns con 1500 boids, avendo quindi uno Speed-Up = $4,1405673 * 10^7\ ns\ /\ 1,7446744 * 10^7$ ns = **2,3733** e un'efficienza = 2,3733 / 12 = **0,1978**.
- Con 3000 boids impiega 4,0159035 * 10^7 ns avendo quindi uno Speed-up = $14,4988529 * 10^7\ ns\ /$ 4,0159035 * 10^7 ns = **3,6104** e un'efficienza = 3,6104 / 12 = **0,3009**.
- Con 4500 boids impiega invece 8.5213041599 * 10^7 ns avendo quindi uno Speed-up = 27,6871810 * 10^7 ns / 8.5213041599 * 10^7 ns = **3,2492** e un'efficienza = 3,2492 / 12 = **0,2708**.

La terza versione, la quale prevede una gestione dei boids tramite i virtual thread, impiega:

- 1,71831008 * 10^7 ns con 1500 boids, avendo quindi uno Speed-Up = $4,1405673 * 10^7\ ns\ /\ 1,71831008 * 10^7$ ns = **2,4097** e un'efficienza = 2,4097 / 12 = **0,2008**.
- Con 3000 boids impiega 4,24942839 * 10^7 ns avendo quindi uno Speed-up = $14,4988529 * 10^7\ ns\ /$ 4,24942839 * 10^7 ns = **3,4120** e un'efficienza = 3,4120 / 12 = **0,2843**.
- Con 4500 boids impiega invece 9.1806675 * 10^7 ns avendo quindi uno Speed-up = 27,6871810 * 10^7 ns / 9.1806675 * 10^7 ns = **3,0158** e un'efficienza = 3,0158 / 12 = **0,2513**.

Dall’analisi delle prestazioni emerge che tutte le versioni parallele del programma offrono un miglioramento significativo rispetto alla versione sequenziale, con risultati particolarmente positivi nel caso di 3000 e 4500 Boid, dove si ottengono gli Speed-Up e le efficienze più elevate. Questo dimostra che il parallelismo è efficace soprattutto con carichi di lavoro medio-alti, mentre tende a essere penalizzato dall’overhead nei casi con pochi Boid.

Le tre versioni parallele si comportano in modo comparabile, con la versione task-based leggermente in vantaggio in termini di scalabilità ed efficienza. La versione con virtual thread mostra prestazioni solide, anche se leggermente inferiori nei test più pesanti, probabilmente per limiti legati alla gestione delle operazioni CPU-intensive. La versione multithread tradizionale si mantiene competitiva, pur con una leggera perdita di efficienza all’aumentare del carico.

In conclusione, l'ottimizzazione tramite parallelismo apporta benefici concreti in termini di prestazioni, ma la scelta della strategia più adatta dipende fortemente dal contesto d’uso e dal tipo di carico computazionale da gestire. È essenziale quindi considerare le specifiche caratteristiche e limitazioni di ciascun approccio per ottenere i migliori risultati.