Programmazione Concorrente e Distribuita
Assignment-01

Lorenzo Tosi - lorenzo.tosi10@studio.unibo.it
Matteo Susca - matteo.susca@studio.unibo.it
Andrea Zavatta - andrea.zavatta3@studio.unibo.it
Alessandro Stefani - alessandro.stefani10@studio.unibo.it

10 aprile 2025

Indice

Capitolo 1

Introduzione

Lo scopo dell'assignment è quello di sviluppare una versione concorrente della "Simulazione dei boid", proposta nel 1986 da Craig Reynolds. L'obiettivo primario è quello di sviluppare tre diverse implementazioni concorrenti e valutarne le prestazioni. I tre approcci affrontati sono:

1. Programmazione multithreaded Java.
2. Task-based basato sul Java Executor Framework.
3. Virtual Threads di Java.

La simulazione richiede anche l'implementazione delle seguenti funzionalità grafiche:

- Specificare il numero di boid all'interno della simulazione.
- Far partire, sospendere e ricominciare la simulazione.
- Impostare tramite l'utilizzo di slider i parametri di separazione, allineamento e coesione dei boid.

Capitolo 2

Analisi del Dominio

"Boids is an artificial life program, developed by Craig Reynolds in 1986, which simulates the flocking behaviour of birds, and related group motion. [...] The name "boid" corresponds to a shortened version of "bird-oid object", which refers to a bird-like object. Reynolds' boid model is one example of a larger general concept, for which many other variations have been developed since." [1]

La complessità computazionale dell'implementazione sequenziale di questo modello è esponenziale: con elevati valori di boids, diviene estremamente dispendioso calcolare e aggiornare le loro caratteristiche ad ogni ciclo. La soluzione implementativa concorrente risulta particolarmente utile nella gestione dei parametri di ogni singolo boid, tra cui la sua posizione e la sua velocità. Il vantaggio dell'utilizzare un approccio concorrente risiede proprio in un sensibile aumento delle prestazioni dovuto al fatto che, nello stesso istante, un numero maggiore di boids viene trattato.

[1] `https://en.wikipedia.org/wiki/Boids`

Capitolo 3

Design ed Implementazione Generale

L'approccio sequenziale inizialmente fornito segue la seguente struttura: per ciascun boid presente nella simulazione, viene calcolata iterativamente la sua velocità e la sua posizione nello spazio, e poi questi due valori vengono aggiornati. Dunque, sia le operazioni di calcolo che quelle di aggiornamento avvengono in modo atomico nello stesso ciclo, per ciascun boid; tuttavia, finchè la simulazione non viene fermata, il programma ripete le istruzioni all'infinito per tutto lo stormo.
L'approccio concorrente sfrutta l'utilizzo di `Threads` per gestire in modo più efficiente le operazioni chiave per la gestione dello stormo. Ogni `Thread` calcola la velocità dei boids a lui assegnati, e attende che gli altri `Threads` facciano lo stesso. Solo allora, tutti i `Threads` procedono all'aggiornamento di tali valori. Una volta che i valori di ogni boid sono stati aggiornati, il processo inizia da capo. Per far sì che le operazioni di calcolo e di aggiornamento dei valori non si sovrappongano è stata usata una `Barriera`, ovvero un meccanismo di sincronizzazione che blocca i `Thread` finchè tutti non raggiungono un determinato punto comune di destinazione.
La simulazione presenta un aspetto interattivo: al momento dell'avvio, l'utente può specificare il numero di boids desiderati. Successivamente ha a disposizione 3 pulsanti: uno per fare partire la simulazione, uno che la ferma ed un ultimo che la interrompe e la riporta allo stato iniziale. La gestione dello *start* e dello *stop* della simulazione è stata implementata e gestita attraverso un monitor condiviso a tutti i `Thread`. Il compito di questo monitor, chiamato `SimulationMonitor` è quello di fermare l'esecuzione dei `Thread` ogni volta che l'utente clicca il pulsante "Suspend", e di fare riprendere l'esecuzione quando l'utente clicca il pulsante "Start". Questo comportamento è ottenuto attraverso il metodo `waitIfSimulationIsStopped()` presente nel

`SimulationMonitor`. Tale metodo viene invocato da tutti i `Thread` che condividono il monitor, al fine di verificare se la simulazione sia attualmente in esecuzione. Il metodo è sincronizzato per garantire l'accesso esclusivo al monitor. Al suo interno, viene utilizzato un ciclo while che controlla la variabile booleana `simulationIsRunning`: se questa è `false`, la simulazione non è attiva e i `Thread` vengono messi in attesa chiamando il metodo `wait()`.

```
public synchronized void waitIfSimulationIsStopped() {
        while (!this.simulationIsRunning) {
            try {
                wait();
            } catch (InterruptedException e) {
                ...
            }
        }
    }
```

Quando l'utente clicca sul pulsante di Start, tutti i `Thread` fermi in `wait()` verranno risvegliati, attraverso l'invocazione del metodo `notifyAll()` e all'assegnazione al valore `true` del booleano.

```
    public synchronized void startSimulation(){
        simulationIsRunning = true;
        notifyAll();
    }
```

L'ultimo pulsante, "Stop", ha il compito di interrompere la simulazione, riportandola allo stato iniziale. Questo è ottenuto mediante la distruzione dei `Threads`, e in un secondo momento dopo aver "ripulito" tutte le strutture dati presenti nel modello in modo tale da garantire una corretta inizializzazione in caso di un nuovo avvio.

Nel modello originale, il calcolo dei vicini per ciascun boid avviene tramite un confronto esaustivo con tutti gli altri boid presenti nella simulazione. Questo approccio, seppur semplice, ha una complessità computazionale quadratica ($O(n^2)$). Per ottimizzare questi calcoli, è stata introdotta una struttura dati denominata `SpatialHashGrid`. Lo spazio in cui si muovono i boids è suddiviso in una griglia di celle, ciascuna delle quali contiene un sottoinsieme di boid, raggruppati in base alla loro posizione. In fase di calcolo dei vicini, ogni boid consulta solo quelli presenti nella propria cella, riducendo la complessità delle operazioni da $O(n^2)$ a $O(n)$ nel caso più generale.

Capitolo 4

Tre Approcci

4.1 Java Multithreaded Programming

4.1.1 Analisi

La prima evoluzione del progetto ha riguardato la trasformazione dell'architettura da sequenziale a concorrente, con l'obiettivo di migliorare l'efficienza del sistema. Nella versione iniziale, tutta la logica è eseguita in un unico thread, il che rendeva la simulazione poco efficiente.

Nella versione concorrente, è stata introdotta una gestione multi-threaded, assegnando ad un thread dedicato un gruppo di boids. Questo ha permesso di parallelizzare il calcolo dei movimenti, ottenendo un miglioramento significativo nelle prestazioni, soprattutto all'aumentare del numero di agenti presenti nella simulazione.

4.1.2 Implementazione

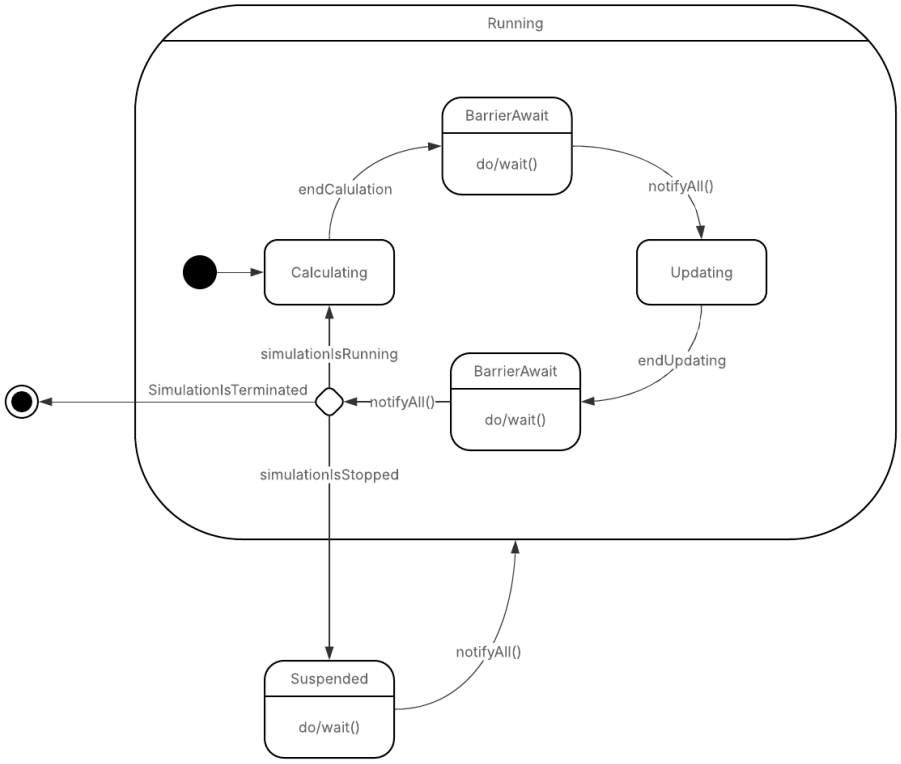

Figura 4.1: Comportamento del `MultiWorker`

I boids vengono suddivisi equalmente tra un numero di thread pari a `Runtime.availableProcessors() + 1`. Questi thread, chiamati `MultiWorker`, internamente gestiscono un gruppo di boids ed eseguono nel loop interno al metodo `run()`, per ogni boid, il calcolo della velocità vettoriale e l'update della stessa, seguito dall'aggiornamento della posizione. Per evitare l'aggiornamento prematuro della posizione di un gruppo di boids si è scelto di inserire due barriere: la prima attende che tutti i `MultiWorker` abbiano effettuato il calcolo della velocità vettoriale basandosi sui dati aggiornati al ciclo precedente; La seconda, similmente, attende che tutti i `MultiWorker` abbiano aggiornato i valori di velocità vettoriale e posizione prima di consentire l'inizio di un nuovo ciclo, evitando che un thread possa effettuare il calcolo della velocità su dati potenzialmente non ancora aggiornati.

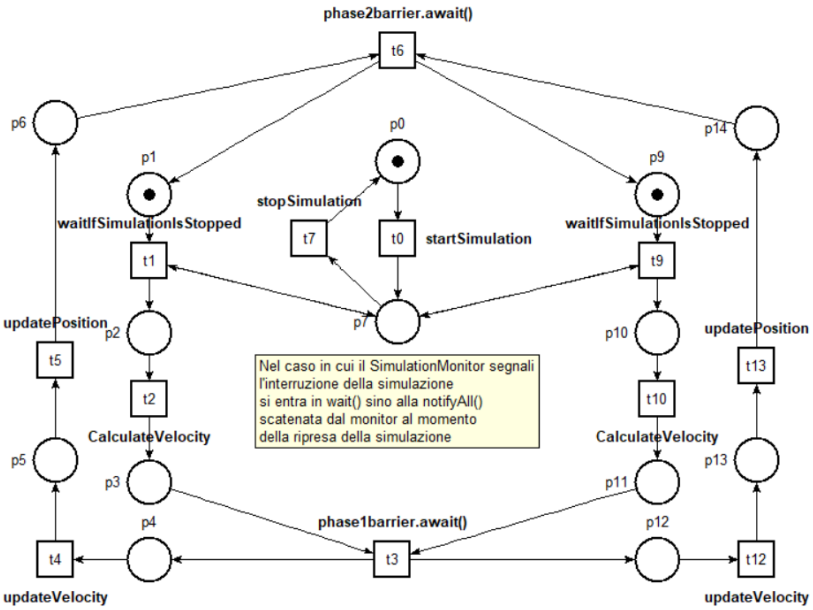

Figura 4.2: Esempio a 2 thread della sincronizzazione tra `MultiWorker` e `SimulationMonitor`

I `MultiWorker` sono inoltre interfacciati con il `SimulationMonitor` che consente loro di attendere la terminazione della `wait()` presente all'interno del monitor (`waitIfSimulationIsStopped()`) nel caso in cui arrivi il segnale di pausa dall'interfaccia grafica.

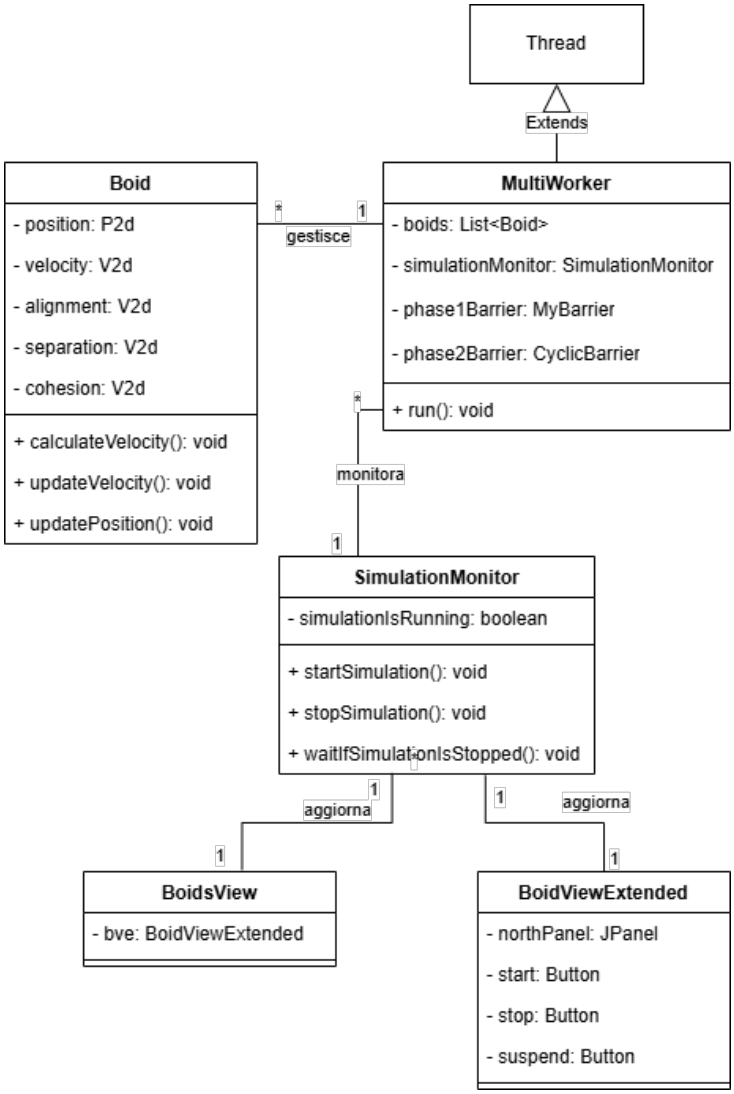

Figura 4.3: Diagramma delle classi

4.1.3 Java Pathfinder

La presenza di errori in questa versione è stata controllata con l'utilizzo di Java Pathfinder, un software di model checking per programmi Java. Il sistema testato è un modello semplificato della simulazione con un numero inferiore di boids ed i `MultiWorker` eseguono un numero limitato di cicli. Il programma fa partire la simulazione, crea i boids ed i Threads ed infine termina l'intero processo. Al momento, a progetto finito, Java pathfinder non rileva errori nel presente assignment.

4.2 Soluzione Task Based

4.2.1 Analisi

La seconda evoluzione del progetto ha riguardato il passaggio da una gestione esplicita dei `Thread` a un'architettura basata su `ExecutorService` e Task.

4.2.2 Implementazione

Sono state create due differenti classi di task: `CalculateBoidVelocityTask` e `UpdateBoidTask`. Queste implementano l'interfaccia `Runnable` e gestiscono una parte dello stormo, il quale è stato equamente suddiviso tra tutti i task. I compiti di questi task sono:

- Calcolare la nuova velocità vettoriale del gruppo di `Boid`.
- Aggiornare la velocità vettoriale e la posizione dei `Boid`.

All'interno di `BoidsModel` vengono creati `Runtime.availableProcessors() + 1` task per entrambe le tipologie sopra descritte ed ad ognuno di questi vengono distribuiti equamente i `Boids` selezionati al lancio della simulazione. L'esecuzione di questi task è avviata da `BoidsSimulator` chiamando i metodi `executeCalculateTask` e `executeUpdateTask` di `BoidsModel`, ed eseguita tramite un executor del tipo `Executors.newCachedThreadPool()`, usando un `CountDownLatch` per la sincronizzazione.

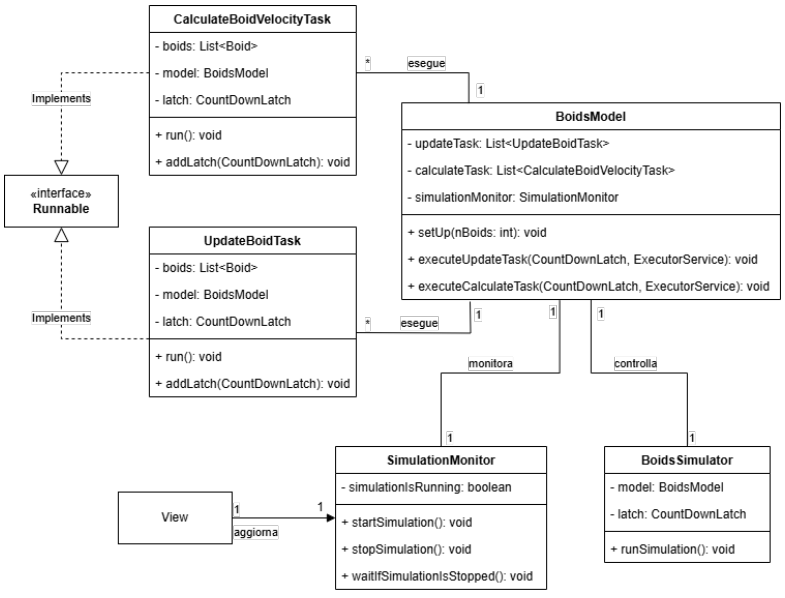

Figura 4.4: Diagramma delle classi

Ad ogni ciclo `BoidsSimulator` controlla, chiamando il `SimulationMonitor`, che la simulazione non sia sospesa. Poi, assegna all'`ExecutorService` un pool di cached threads attraverso i quali i task vengono eseguiti e crea un `CountDownLatch` per la sincronizzazione dell'esecuzione. Dunque, vengono lanciate le esecuzioni dei task di calcolo della velocità vettoriale e ad ognuno viene passato il latch precedentemente generato e l'`ExecutorService`. La simulazione attende la terminazione dell'esecuzione di tutti i `CalculateBoidVelocityTask` tramite la funzione `latch.countDown()` chiamata al termine della funzione `run()` del task. L'esecuzione poi continua, in modo analogo a quanto descritto in precedenza, creando un nuovo `CountDownLatch` ed eseguendo gli `UpdateBoidTask`. Infine, il ciclo termina eseguendo lo shutdown dell'executor.

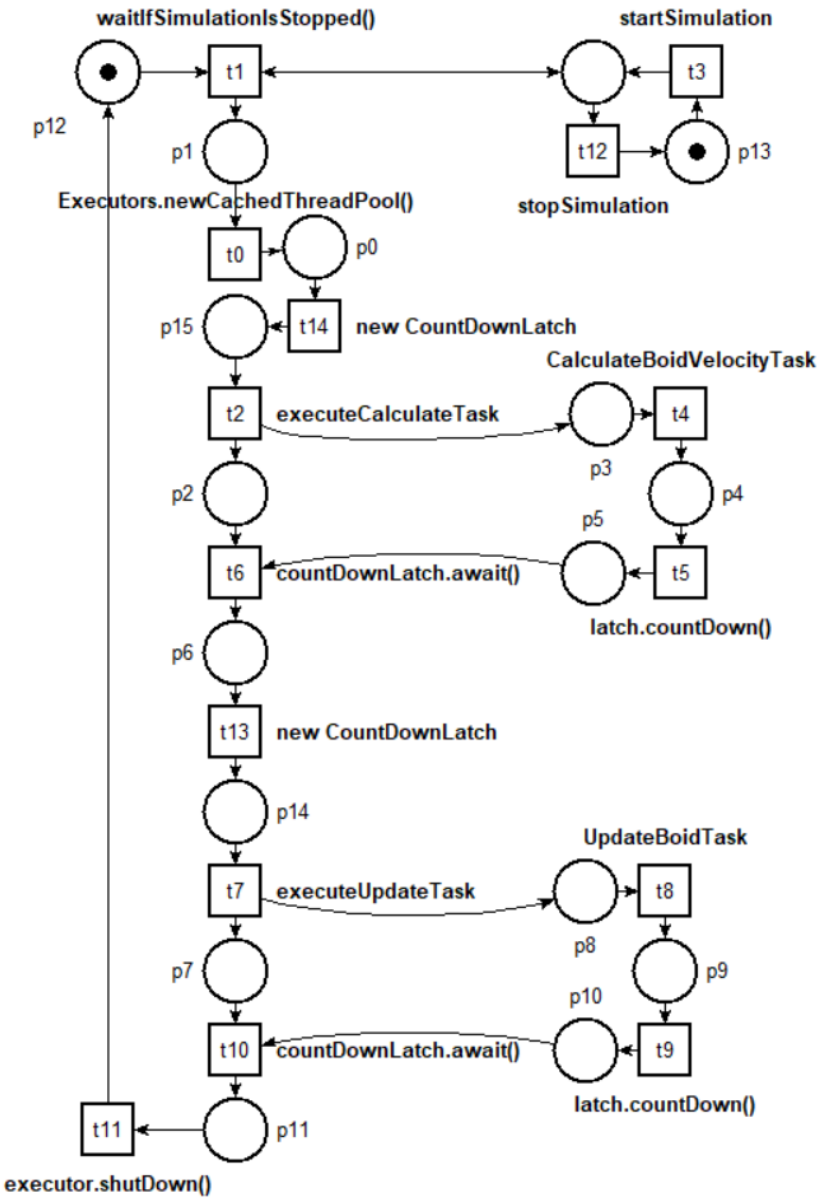

Figura 4.5: Esempio di esecuzione e sincronizzazione di `BoidsSimulator` con `CalculateBoidVelocityTask` e `UpdateBoidTask`

4.3 Virtual Threads

4.3.1 Analisi

L'ultimo approccio, basato sui Virtual Threads di Java, riprende la prima versione del progetto, sostituendo i `Thread` con `VirtualThread`.

4.3.2 Implementazione

La struttura è rimasta del tutto simile alla prima versione dell'assignment tranne che per la modifica di `MultiWorker` in `VirtualWorker`. Mentre il `MultiWorker` gestisce una lista di boids, il `VirtualWorker` gestisce un singolo oggetto boid. Questo è reso possibile grazie ai Thread Virtuali che, essendo molto più leggeri rispetto ai thread tradizionali, consentono di associare un thread per ogni boid senza peggiorare le prestazioni della simulazione.

Capitolo 5

Valutazione delle Performance

Per analizzare e comparare le prestazioni delle varie versioni nei vari scenari sono stati ottenuti e classificati i dati in due modalità:

- Calcolo del tempo impiegato per l'esecuzione di 1000 cicli
- Calcolo del tempo impiegato per l'esecuzione di un singolo ciclo

La seguente tabella mostra il tempo di esecuzione di 1000 cicli (in cui un ciclo termina con l'aggiornamento della posizione di tutti i Boid) nelle varie versioni e, dove possibile, con un numero differente di Threads.

Versione	**Threads**	**Tempo (ms)**
Sequential	1	57 735
Multithread	8	3650
Multithread	2	6306
Multithread	1	12 537
Task Based	8	4226
Task Based	2	6938
Task Based	1	13 916
Virtual Threads	1500	6309

Tabella 5.1: Tempo di esecuzione di 1000 cicli con 1500 boids

Qui viene illustrato il valore dello Speedup nelle diverse versioni al variare del numero di Thread.

Thread	Tempo (ms)	Speedup vs Seq[1]	Speedup vs v1-1T	Efficienza
1	12537	4.60×	–	1.00
2	6306	9.15×	1.99×	0.995
8	3650	15.82×	3.43×	0.429

Tabella 5.2: Prestazioni della versione v1

Thread	Tempo (ms)	Speedup vs Seq[1]	Speedup vs v2-1T	Efficienza
1	13916	4.15×	–	1.00
2	6938	8.32×	2.01×	1.005
8	4226	13.65×	3.29×	0.411

Tabella 5.3: Prestazioni della versione v2

Thread	Tempo (ms)	Speedup vs Seq[1]	Speedup vs v2-1T	Efficienza
1500 (virt)	6309	9.15×	2.20×	0.0061

Tabella 5.4: Prestazioni con virtual thread

L'efficienza risulta molto bassa nella versione che utilizza i virtual thread poiché essi sono progettati per ottimizzare la concorrenza leggera e operazioni I/O-bound. La bassa efficienza non riflette un difetto dei virtual thread, ma un aspetto caratteristico in contesti di calcolo intensivo.
Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency).[2]

[1]Nota: la differenza tra la versione sequenziale e le altre non risiede unicamente nell'impiego dei thread. Queste differenze potrebbero aver contribuito ad un ulteriore aumento di prestazioni.

[2]`https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-BEC799E0-00E9-4386-B220-8839EA6B4F5C`

Per i seguenti grafici[3] il codice è stato modificato per estrarre i dati relativi al tempo di esecuzione di una singola iterazione che inizia con il calcolo della velocità e termina con l'aggiornamento della posizione. Al fine di ridurre il rumore e rendere i grafici più comprensibili è stata applicata una media aritmetica ad intervalli di 100 misurazioni.

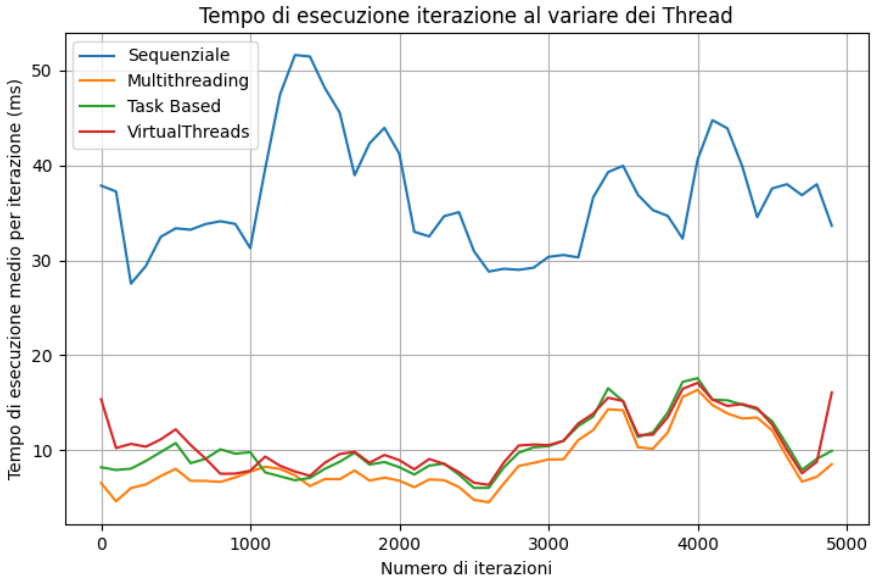

Figura 5.1: Grafico delle performance con 5 thread (4 + 1) e 1500 boids

Per visualizzare la differenza di performance di una stessa versione al variare del numero di Thread sono stati generati i seguenti grafici

[3]Nota: I seguenti test sono stati eseguiti su una macchina differente da quella usata per le misurazioni precedenti.

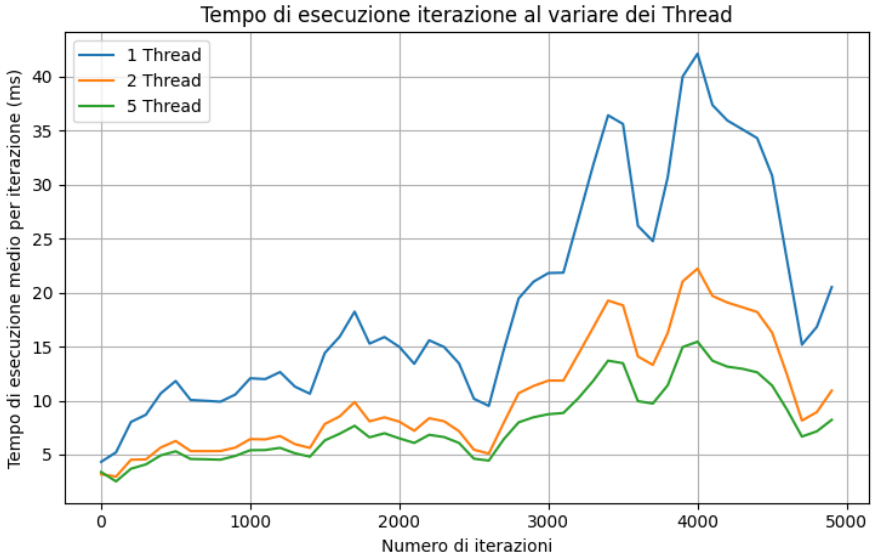

Figura 5.2: Grafico delle performance della versione MultiThreaded con 1,2 e 5 Threads, 1500 boids

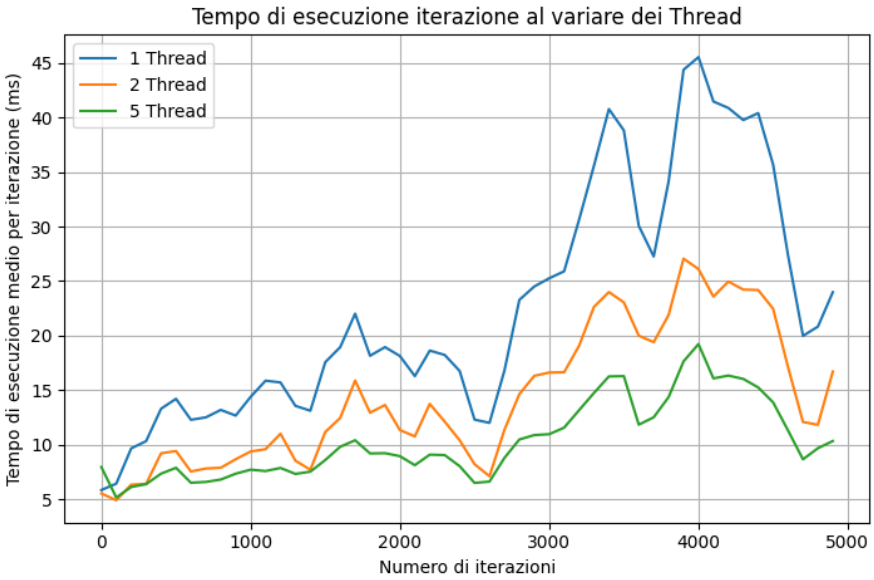

Figura 5.3: Grafico delle performance della versione Task Based con 1,2 e 5 Threads, 1500 boids

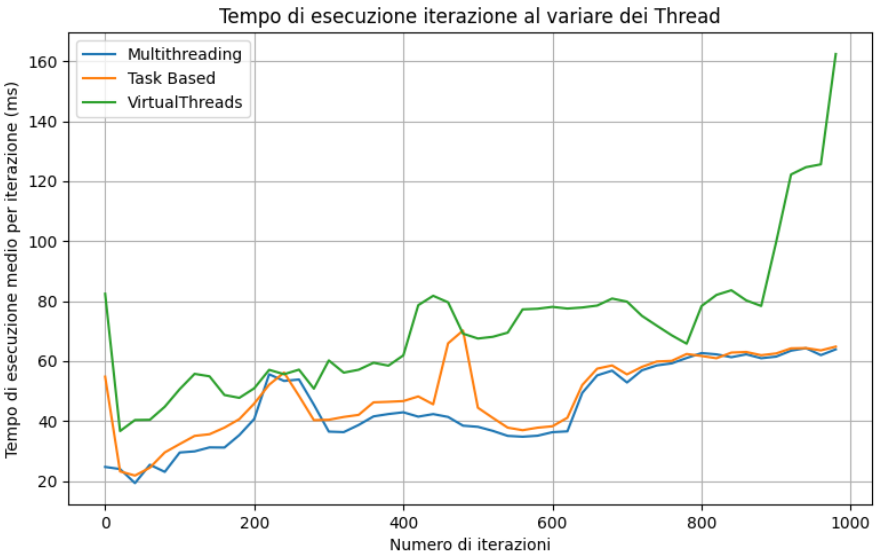

Figura 5.4: Grafico delle performance con 5000 boids ad intervalli di 20 misurazioni

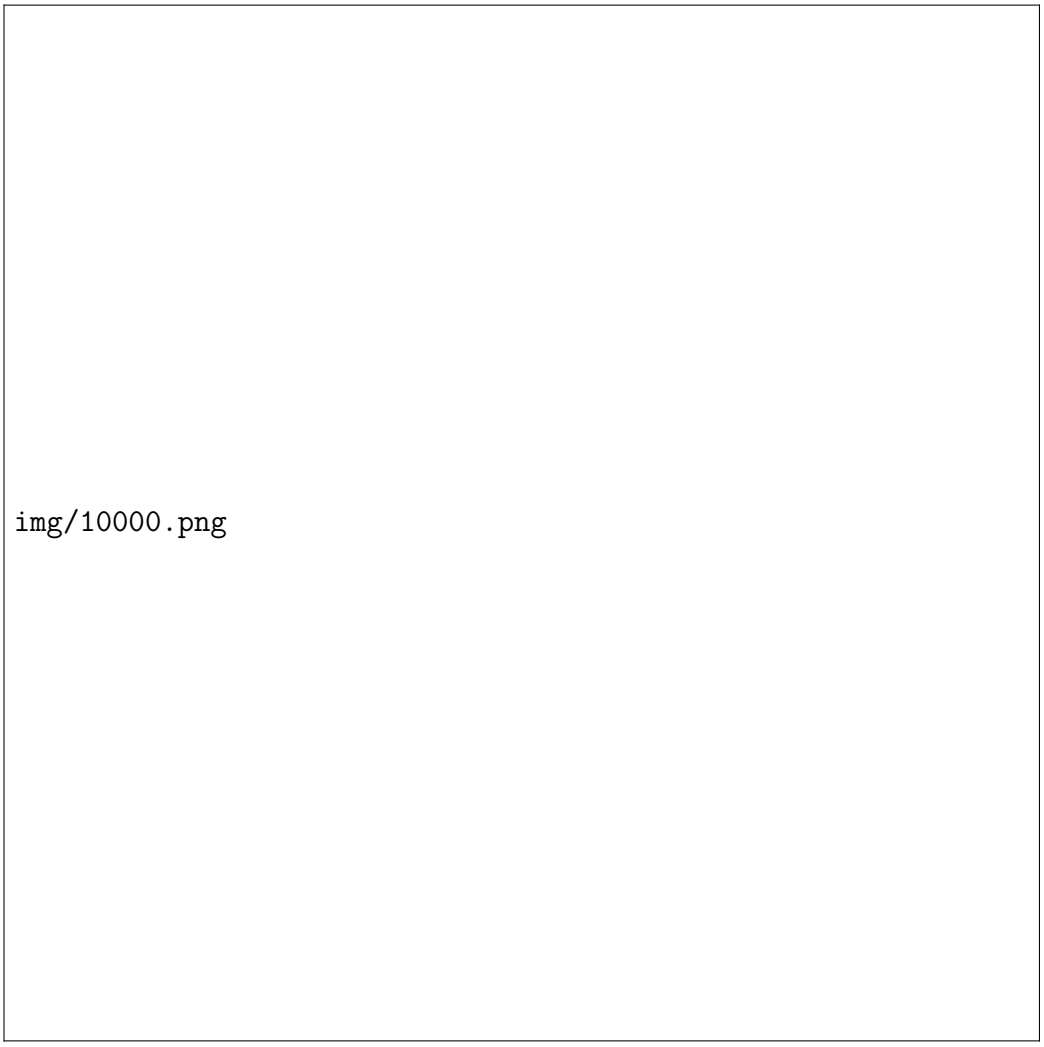

Figura 5.5: Grafico delle performance con 10000 boids ad intervalli di 20 misurazioni

È possibile notare che le ottimizzazioni introdotte hanno portato a un significativo miglioramento delle prestazioni rispetto alla versione sequenziale del sistema. Grazie all'adozione di soluzioni concorrenti e strutture dati più efficienti, si è riscontrata una riduzione del carico computazionale e un aumento alla scalabilità del modello.