Pcd-Assigment01 Report

Concurrent Boids

Bedeschi Federica
0001175655 federica.bedeschi4@studio.unibo.it

Pracucci Filippo
0001183555 filippo.pracucci@studio.unibo.it

Anno accademico 2024-2025

Indice

Analisi

1.1 Descrizione del problema

Il problema dei Boids consiste in una simulazione del comportamento degli stormi di uccelli. Ogni boid aggiorna la propria posizione in maniera continuativa, solo dopo aver aggiornato la propria velocità in base a quelle degli altri boid. In questo modo si simula il comportamento di gruppo, che può essere condizionato dalle seguenti regole:

- separazione: il boid vira al fine di evitare il sovraffollamento locale (dunque si allontana dai boid vicini);
- allineamento: il boid vira al fine di allinearsi alle traiettorie di volo dei boid vicini;
- coesione: il boid vira al fine di muoversi verso la posizione media (baricentro) dei boid vicini.

1.2 Visione concorrente

Dal punto di vista concorrente gli aggiornamenti delle velocità e delle posizioni per ogni boid possono essere fatti concorrentemente. Tuttavia, bisogna tenere conto che tutte le posizioni possono essere aggiornate solo dopo aver aggiornato tutte le velocità, e viceversa. Analizzando la soluzione sequenziale fornita, è emerso come punto critico, a livello di singolo boid, l'aggiornamento della velocità in quanto nel calcolo dell'allineamento si accede alle velocità dei boid "vicini". Per questo motivo l'accesso a queste ultime viene fatto in mutua esclusione.
A livello di interfaccia grafica, l'aggiornamento della vista deve essere eseguito a seguito dell'aggiornamento di tutte le posizioni dei boid.

Design e architettura

Abbiamo optato per un'architettura che seguisse il pattern *MVC*, quindi con un *Controller* che si occupasse di mettere in comunicazione il *Model* con la *View*. Ogni versione utilizza un Controller equivalente, modificando esclusivamente il metodo di avvio della simulazione. I Controller sfruttano due monitor per notificare i componenti attivi degli eventi di sospensione ed interruzione scaturiti dalla View.

2.1 Multithreaded

Abbiamo creato due tipi di worker, un *ComputeWorker* che si occupa di tutta la computazione riguardante i boid, e un *GUIWorker* che si occupa di aggiornare l'interfaccia grafica. Viene istanziato un numero di ComputeWorker uguale al numero di core presenti nella macchina sottostante, consentendo di sfruttare appieno tutte le risorse dell'architettura. Al contrario viene istanziato un singolo GUIWorker.
Al fine di bilanciare il lavoro dei ComputeWorker, ad ognuno viene affidata una porzione dei boid totali. Per effettuare la sincronizzazione tra i worker abbiamo creato una barriera ciclica, della quale utilizziamo due istanze, una per attendere il completamento degli aggiornamenti delle velocità, ed una seconda per aspettare la terminazione degli aggiornamenti delle posizioni. Entrambe le barriere vengono utilizzate sia dai ComputeWorker che dal GUIWorker.

2.1.1 Rete di Petri

La rete di Petri in Figura 2.1 descrive il comportamento del sistema, ovvero si compone di due attività, una per i ComputeWorker ed una per il GUIWorker. La prima gestisce un numero di token pari al numero di istanze di ComputeWorker create, cioè il numero di core disponibili (N); mentre la

seconda gestisce un unico token. L'attività del GUIWorker si sincronizza con quella dei ComputeWorker in corrispondenza delle due barriere posizionate in seguito all'aggiornamento delle velocità e delle posizioni.

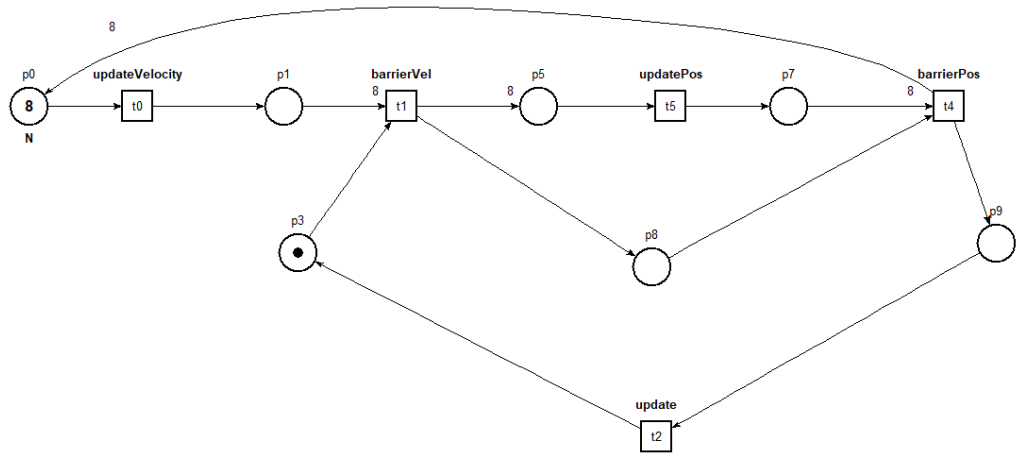

Figura 2.1: Rete di Petri - multithreaded
Ogni 8 presente in figura rappresenta in realtà gli N ComputeWorker utilizzati.

2.2 Task-based

Il comportamento complessivo viene gestito da un *MasterWorker*, il quale si compone di un *Executor*, che manda in esecuzione nei momenti opportuni tre tipi di task diversi:

- *UpdateVelocityTask*: si occupa di aggiornare la velocità di un singolo boid;
- *UpdatePositionTask*: si occupa di aggiornare la posizione di un singolo boid;
- *UpdateGUITask*: si occupa di aggiornare l'interfaccia grafica, dopo aver calcolato il frame rate attuale.

Ad ogni iterazione viene eseguito un numero di UpdateVelocityTask e UpdatePositionTask pari al numero di boids ed un singolo UpdateGUITask.
La sincronizzazione viene gestita tramite il meccanismo dei *Future*, ovvero fornendo all'Executor i task delle posizioni solo dopo aver ottenuto tutti i

risultati relativi ai task delle velocità, e viceversa; mentre l'UpdateGUITask viene eseguito concorrentemente a quelli delle velocità ed il risultato che fornisce corrisponde al tempo che viene utilizzato per calcolare il frame rate.

2.2.1 Rete di Petri

La rete di Petri in Figura 2.2 descrive il comportamento del sistema, evidenziando i punti di sincronizzazione, come descritto sopra, e l'esecuzione concorrente dell'UpdateGUITask con i task delle velocità. I token corrispondono al numero di task eseguiti ad ogni iterazione. Differentemente da quanto descritto nella sezione 2.1.1 gli UpdatePositionTask hanno un flusso di controllo separato rispetto agli UpdateVelocityTask.

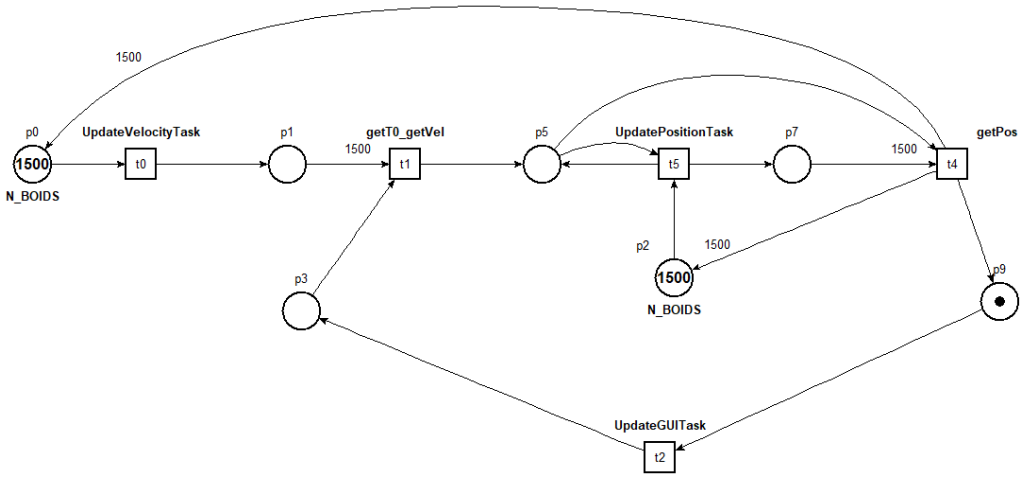

Figura 2.2: Rete di Petri - task-based
Ogni 1500 presente in figura rappresenta in realtà gli N_BOIDS gestiti.

2.3 Virtual threads

Abbiamo creato due tipi di worker, un *ComputeWorker* che si occupa di tutta la computazione riguardante un singolo boid, e un *GUIWorker* che si occupa di aggiornare l'interfaccia grafica. Ogni worker è gestito da un singolo *Virtual Thread*, quindi complessivamente sono presenti un numero di Virtual Thread pari al numero di boids ed in aggiunta uno per il GUIWorker. La sincronizzazione viene gestita analogamente al caso *Multithreaded*, come descritto nella sezione 2.1.

2.3.1 Rete di Petri

La rete di Petri in Figura 2.3 descrive il comportamento del sistema, ovvero si compone di due attività, una per i ComputeWorker ed una per il GUIWorker. La prima gestisce un numero di token pari al numero di istanze di ComputeWorker create, cioè il numero di boids (N_BOIDS); mentre la seconda gestisce un unico token. L'attività del GUIWorker si sincronizza con quella dei ComputeWorker in corrispondenza delle due barriere posizionate in seguito all'aggiornamento delle velocità e delle posizioni.

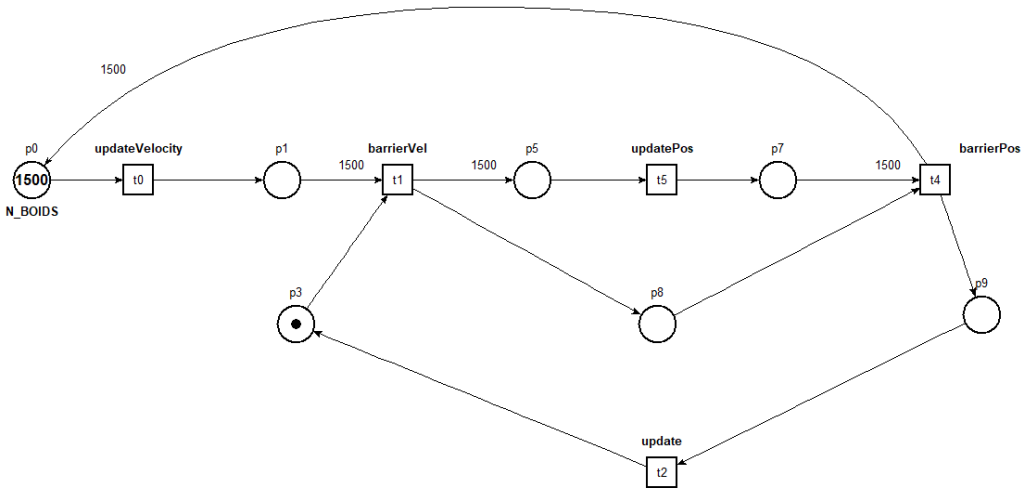

Figura 2.3: Rete di Petri - virtual threads
Ogni 1500 presente in figura rappresenta in realtà il numero di ComputeWorker (equivalente agli N_BOIDS utilizzati).

Test performance

Per valutare le performance delle varie versioni in relazione a quella sequenziale, abbiamo misurato il throughput, ovvero il tempo impiegato per effettuare un'iterazione. Per aver valori più consistenti abbiamo preso in considerazione 500 iterazioni e abbiamo calcolato la media.
Per ogni versione abbiamo ripetuto le misurazioni al variare del numero di boid e del numero di thread. Nello specifico, la versione sequenziale è rappresentata in ogni grafico dalla curva indicata con un thread, mentre le altre rappresentano la versione concorrente al variare del numero di thread.
I valori ottenuti con 9 thread sono evidenziati in quanto rappresentano il valore teoricamente ottimale, ovvero $N\_core + 1$, considerando che l'architettura utilizzata per effettuare i test presenta un processore ad 8 core.
Per ogni versione riportiamo i calcoli dello speedup e dell'efficienza, secondo le formule:

- Speedup: $S = \frac{T_1}{T_N}$, dove T_1 indica il throughput del caso sequenziale e T_N il throughput del caso concorrente con N thread;
- Efficienza: $E = \frac{S}{N}$, dove S indica lo speedup e N il numero di thread.

3.1 Multithreaded

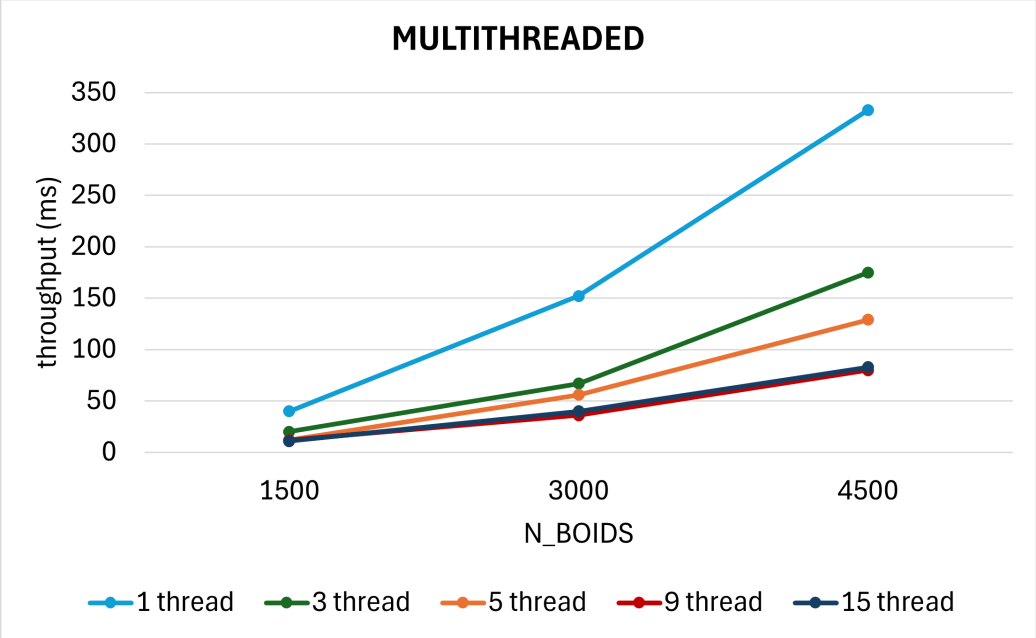

Figura 3.1: Throughput test - multithreaded

	N_THREADS	N_BOIDS		
		1500	3000	4500
Speedup	15	3.64	3.80	4.01
	9	**3.33**	**4.22**	**4.16**
	5	3.33	2.71	2.58
	3	2.00	2.27	1.90
Efficienza	15	0.24	0.25	0.27
	9	**0.37**	**0.47**	**0.46**
	5	0.67	0.54	0.52
	3	0.67	0.76	0.63

Tabella 3.1: Misure performance - multithreaded

3.2 Task-based

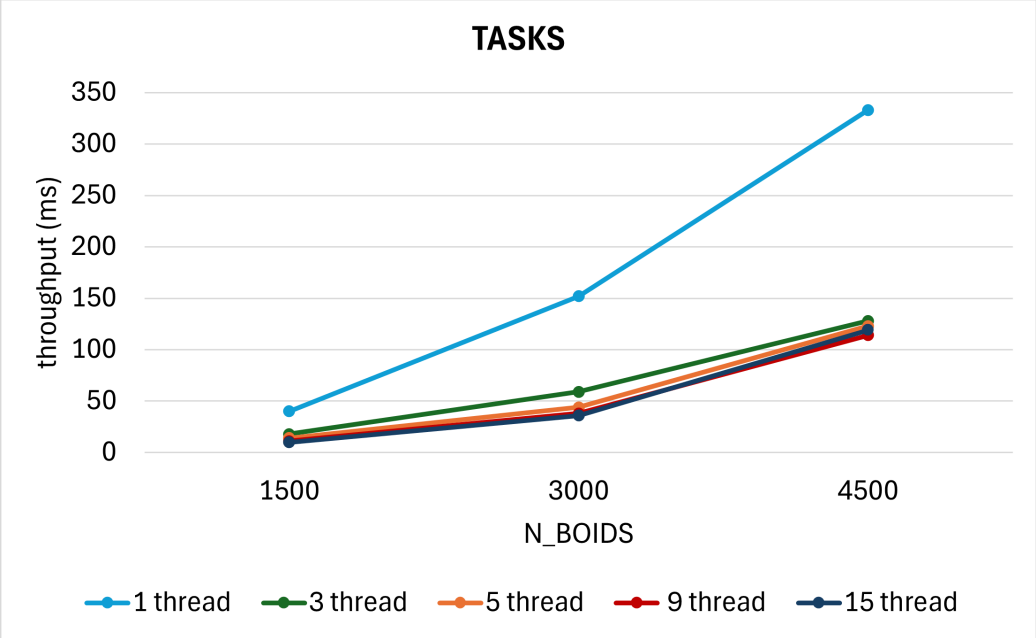

Figura 3.2: Throughput test - task-based

	N_THREADS	N_BOIDS		
		1500	3000	4500
Speedup	15	4.00	4.22	2.80
	9	**3.64**	**4.00**	**2.92**
	5	2.86	3.45	2.71
	3	2.22	2.58	2.60
Efficienza	15	0.27	0.28	0.19
	9	**0.24**	**0.25**	**0.27**
	5	0.57	0.69	0.54
	3	0.74	0.86	0.87

Tabella 3.2: Misure performance - task-based

3.3 Virtual threads

In questa versione abbiamo preso in considerazione solo un numero di thread pari a 9, in quanto non siamo riusciti a modificare il numero di platform thread.

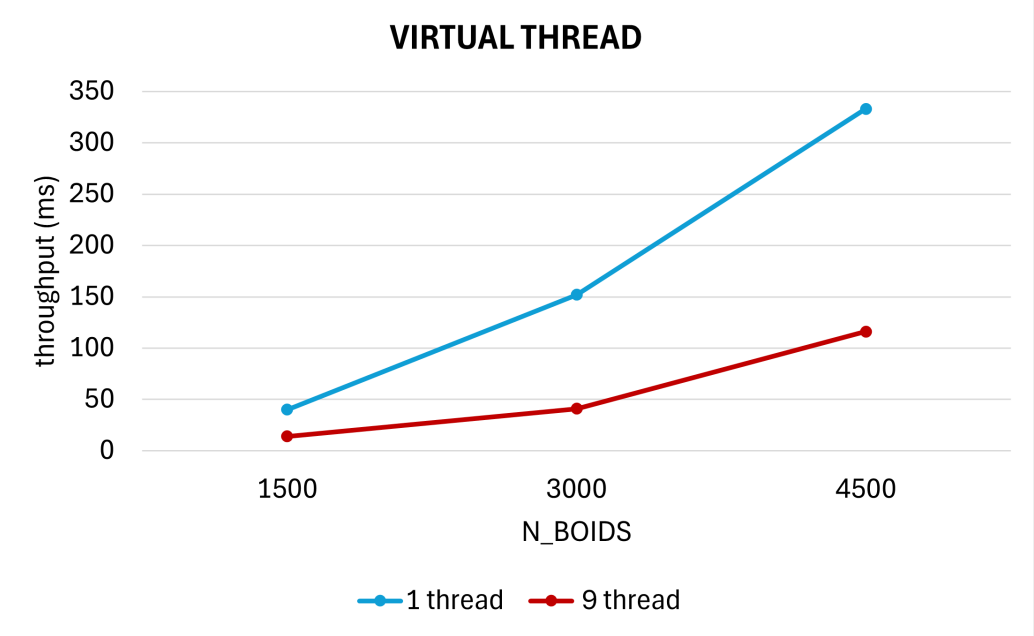

Figura 3.3: Throughput test - virtual threads

	N_THREADS	N_BOIDS		
		1500	3000	4500
Speedup	9	2.86	3.71	2.87
Efficienza	9	0.32	0.41	0.32

Tabella 3.3: Misure performance - virtual threads

Conclusioni

Per quanto riguarda le performance si può notare come le varie versioni concorrenti scalino correttamente all'aumentare del numero di boid e in relazione al numero di thread. Il miglioramento delle prestazioni, come ci si poteva aspettare, però ha un limite in corrispondenza di un numero di thread pari a $N\_core + 1$, quindi nel nostro caso 9 thread, che corrisponde al valore teoricamente ottimale; questo lo si nota nei grafici e nelle tabelle in corrispondenza dei valori ottenuti utilizzando 15 thread. Inoltre, come evidenziato dai grafici e dai valori dello speedup, c'è un miglioramento significativo nelle prestazioni delle versioni concorrenti rispetto a quelle della versione sequenziale.
Abbiamo controllato la correttezza del nostro modello multithreaded tramite il tool JPF utilizzando una versione semplificata con le sole barriere per la sincronizzazione ed utilizzando un numero di iterazioni predefinito. Da questo controllo non è emerso nessun errore.