PCD Assignment 01 - Boids Simulator

A cura di

Alessandra Versari - alessandra.versari2@studio.unibo.it

Lorenzo Rigoni - lorenzo.rigoni2@studio.unibo.it

Riccardo Moretti - riccardo.moretti6@studio.unibo.it

Analisi del problema

In questo assignment, viene richiesto di implementare una versione concorrente della "*simulazione dei boid*" proposta da Craig Reynolds nel 1986. Nella simulazione, vengono create ***n*** entità chiamate "*boid*". Ogni *boid*, in un ciclo infinito, deve svolgere due azioni:

1. modificare la propria velocità in base ai pesi di separazione, allineamento e coesione;
2. modificare la propria posizione in base alla velocità calcolata precedentemente.

Oltre a ciò, durante la simulazione, l'utente può modificare i parametri dei boid (separazione, allineamento e coesione) e può sospendere/riprendere la simulazione, oltre a poterla fermare e avviarne una nuova.
Dunque, quando si inizia ad avere un numero elevato di boid, diventa fondamentale gestire tutti gli aggiornamenti con approccio concorrente. Per farlo, si possono utilizzare tre diversi approcci:

- Programmazione multithreaded
- Programmazione task-based
- Programmazione con Virtual Thread

Design, strategie ed architetture usate

Versione multithreaded

Per la versione multithreaded, la soluzione è pensata in funzione dei thread, il cui numero è stato scelto uguale al numero di core del sistema più uno. Ogni thread, implementato come *BoidWorker*, è responsabile dell'aggiornamento delle velocità e delle posizioni di una sottolista dei boid totali.
Inoltre, per quanto riguarda i worker, abbiamo dovuto implementare altre due entità: la prima è una barriera (*UpdateBarrier*) la quale fa sì che le posizioni vengano calcolate solo quando tutti i worker hanno calcolato le velocità dei loro boid. La seconda, invece, è un coordinatore dei worker (*WorkersCoordinator*) il quale permette di aspettare che tutti i thread abbiano aggiornato le posizioni e di mettere in wait i worker finché la view non ha finito di disegnare il nuovo scenario.
Infine, per gestire la sospensione, la ripresa e lo stop della simulazione, abbiamo implementato un monitor (*SimulationState*) il quale gestisce i vari stati possibili. Quest'ultimo viene usato da tutti e tre i tipi di simulatori in quanto viene implementato nella classe astratta *AbstractBoidsSimulator*, la quale è estesa dai simulatori.

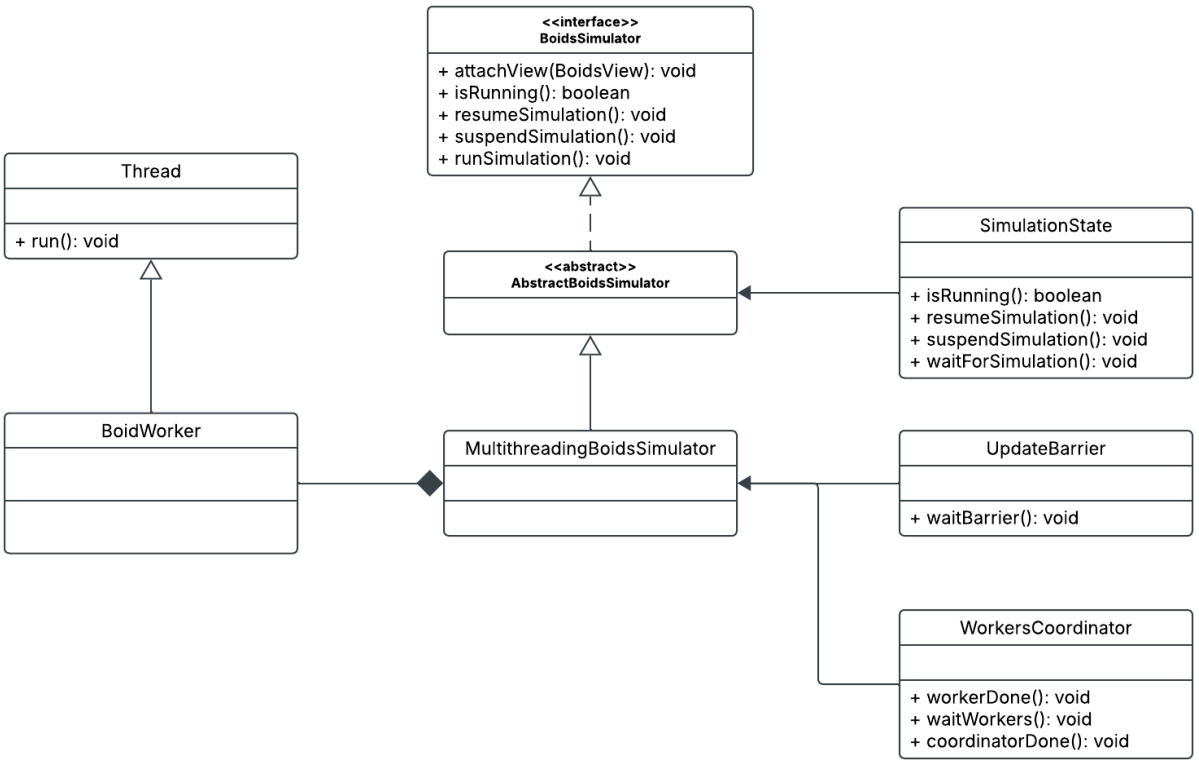


Diagramma UML per la versione multithreaded

Versione task-based

Nella soluzione task-based abbiamo scelto di impiegare un *executer* con pool fissa di thread, pari al numero di core più uno. L'executor esegue un certo numero di *task*, dove i task sono di due tipi: *VelocityBoidsTask*, il quale esegue l'aggiornamento delle velocità di un sottogruppo di boids, e *PositionBoidsTask*, il quale esegue l'aggiornamento delle posizioni di un sottogruppo di boids.

La sincronicità del programma è garantita dall' uso delle *Future*, le quali fanno in modo che si debba attendere che tutti i task abbiano terminato la propria parte prima di ripetere l'operazione. Questo metodo è stato preferito rispetto all'uso di un *latch* in quanto è risultata più stabile rispetto alla controparte.

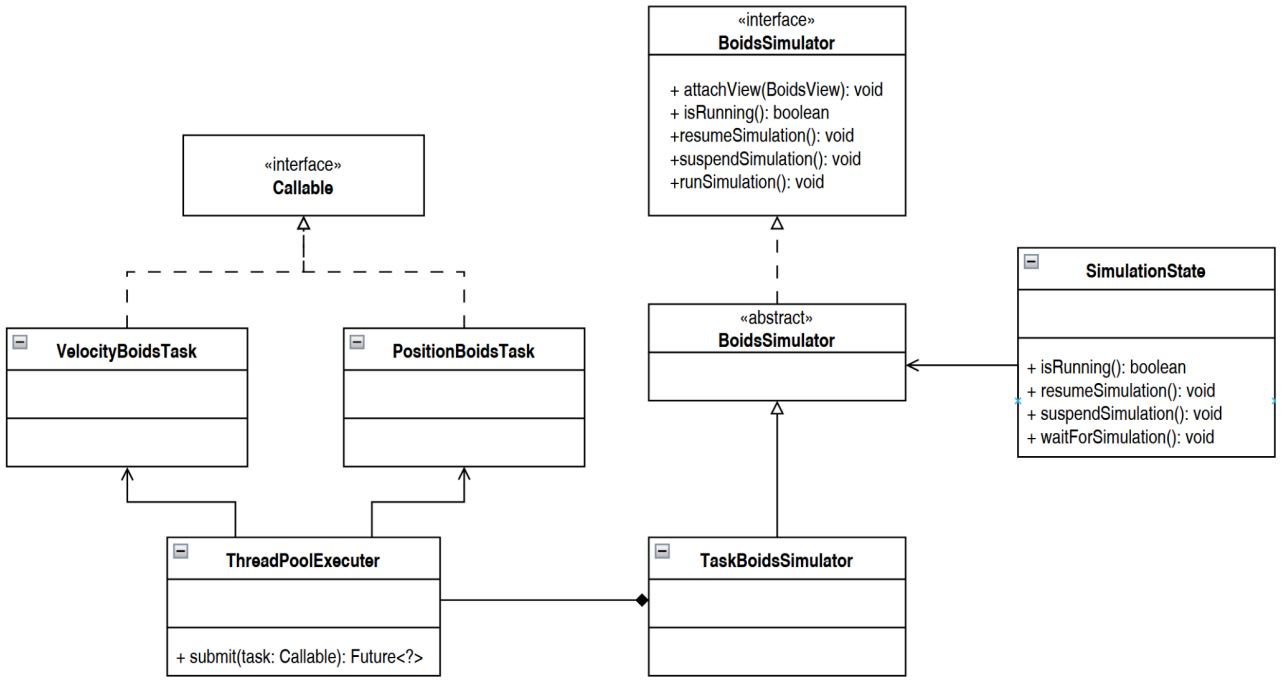


Diagramma UML per la versione task-based

Versione con virtual thread

La soluzione basata sui virtual thread è stata pensata con l'obiettivo di sfruttare al massimo la "leggerezza" di questo tipo di thread.
Siccome i virtual thread sono un'evoluzione dei platform thread e condividono gli stessi meccanismi di base, abbiamo scelto di partire dalla soluzione multithread esistente apportando le modifiche necessarie.
Per quanto riguarda i monitor (*UpdateBarrier* e *WorkersCoordinator*) abbiamo sostituito l'implementazione basata su synchronized utilizzando *ReentrantLock* e *Condition* in modo da regolare l'accesso dei thread alle risorse condivise ed evitare il problema del thread pinning.
In generale, il comportamento dei monitor rimane invariato rispetto alla versione di base, la differenza con la versione multithreaded risiede in *BoidWorker*.
Infatti, abbiamo deciso in questa versione di gestire i boid separatamente e non in sottogruppi, di conseguenza nella classe *VirtualThreadBoidsSimulator* viene creato un virtual thread per ciascun boid, il quale verrà poi gestito dalla classe *BoidWorker*.

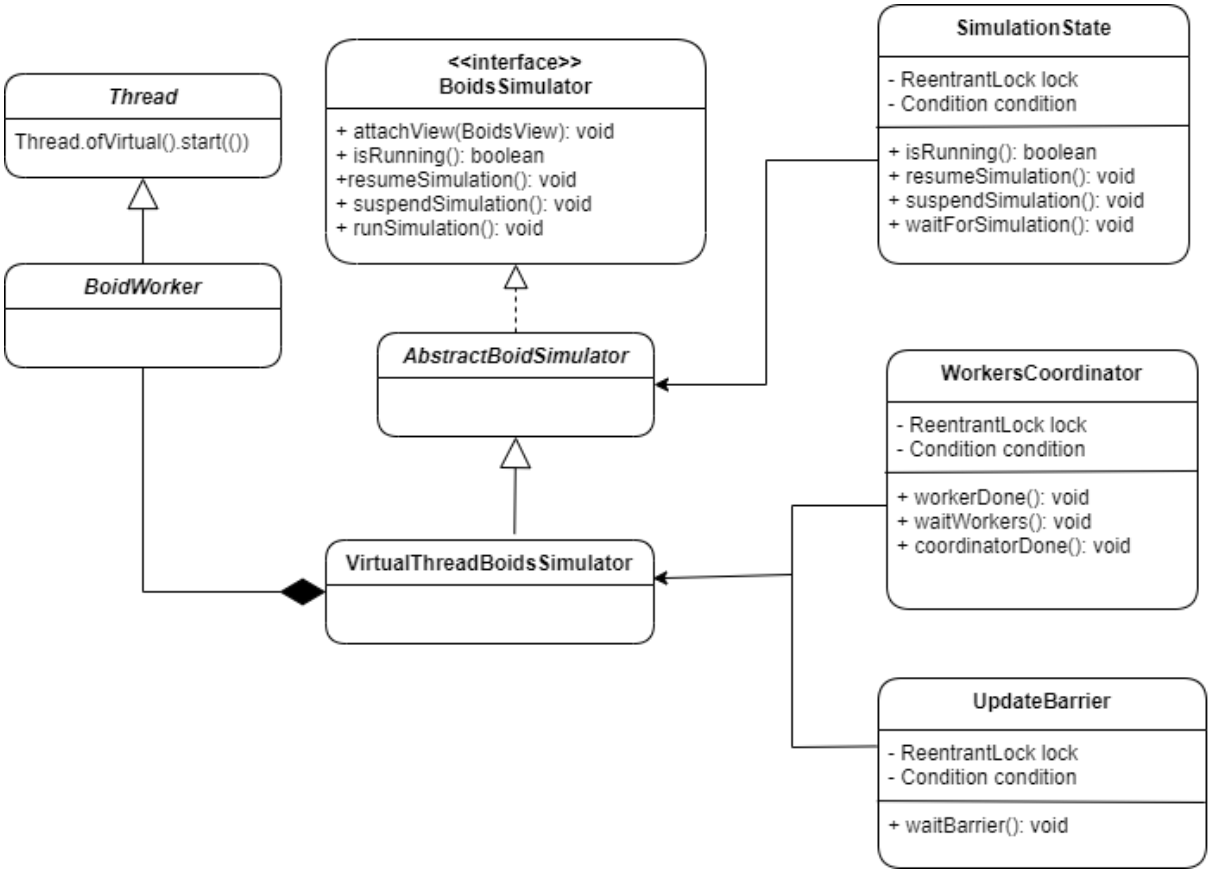
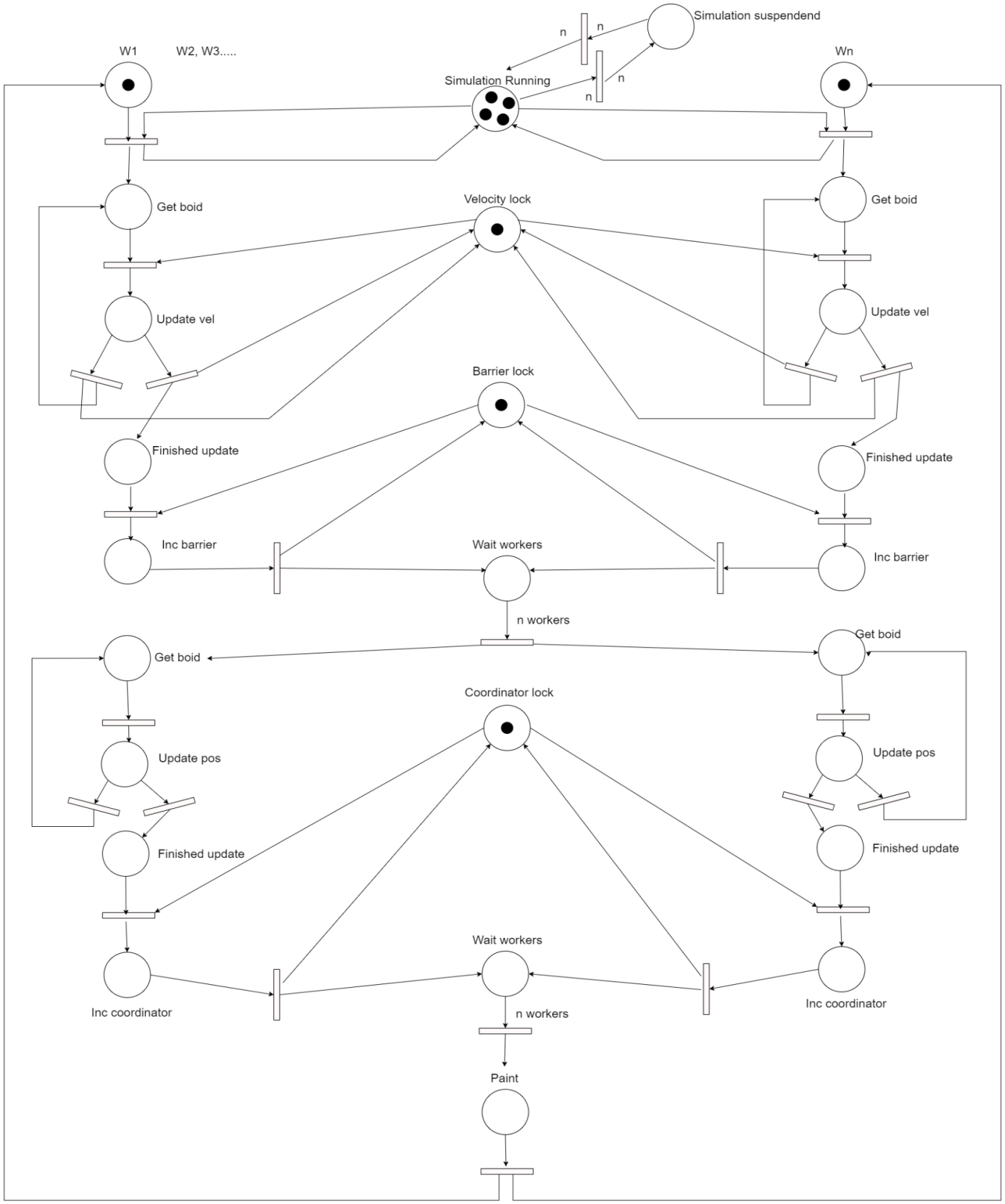


Diagramma UML per la versione virtual thread

Comportamento del sistema



Rete di Petri del sistema multithreaded

La rete di Petri proposta si riferisce alla versione multi-thread della simulazione. Le piazze da W_1 a W_n rappresentano i *workers*, ciascuna inizialmente contenente un token che consente l'avvio del flusso di esecuzione. Tuttavia, per procedere verso la piazza successiva è necessario anche il token del monitor del *simulation state*, che verifica che la simulazione sia effettivamente in corso. Nella piazza associata alla simulazione attiva è presente un token per ogni worker. Quando la simulazione viene sospesa, tutti questi token vengono rimossi, impedendo ai worker di soddisfare le condizioni necessarie per avanzare.
All'avvio, i worker iniziano aggiornando la velocità. Questa operazione richiede l'acquisizione di un lock, poiché il valore della velocità è condiviso e soggetto a modifiche concorrenti. Il lock è rappresentato dalla piazza *velocity lock*, che contiene un unico token per garantire la mutua esclusione.

Una volta terminati gli aggiornamenti delle velocità è necessario acquisire nuovamente il lock per poter incrementare la barriera di sincronizzazione. In questo caso il lock viene rilasciato subito dopo l'incremento. I token dei worker confluiscono poi nella piazza *wait workers* che agisce da piazza di accumulo: solo quando il numero di token presenti è uguale al numero di workers, allora il flusso può proseguire.

Segue poi l'aggiornamento della posizione, che non richiede il lock in quanto utilizza la velocità calcolata precedentemente, la quale non verrà più modificata durante il calcolo delle posizioni.
Una volta finiti gli aggiornamenti delle posizioni*,* entra in gioco il *coordinator* il quale applica un meccanismo analogo a quello della barriera: ciascun worker acquisisce il lock, incrementa il contatore, rilascia il lock e attende che tutti i worker abbiano raggiunto questo punto.

Infine, una volta accumulati gli *n* token, è possibile procedere con la fase di *painting*, seguita dal ripristino dei token nelle rispettive piazze di partenza dei workers, andando così a completare il ciclo.

La versione Task-based differisce leggermente da questa versione. Infatti, al posto dei workers troviamo i *tasks*, i quali però svolgono le stesse funzioni dei primi, ovvero l'aggiornamento delle velocità e delle posizioni di un sottogruppo di boid. Inoltre, un'altra differenza risiede nell'algoritmo di sincronizzazione dei task. Infatti, non vengono più usati una barriera ed un *coordinator*, ma i *future* i quali, tramite la funzione *get*, attendono che i task vengano completati per restituirne il risultato (nel nostro caso di tipo *Void*) .

Infine, il comportamento della versione virtual thread è leggermente diverso da quello della versione multithreaded. In questa versione, infatti, non è necessario lo stato "get boid" in quanto utilizziamo i workers per gestire singoli boid e non liste. Quindi la rappresentazione della rete risulterebbe ancora più lineare rispetto a quella

mostrata precedentemente, in quanto non ci sarebbero i cicli fatti per iterare la sottolista di boid.

Performance

Per il calcolo delle performance del sistema, per ogni diverso approccio, è stato scelto il seguente *modus operandi*:

1. Vengono usati tre diversi numeri di boids: 1500, 2000 e 2500;
2. Per ogni prova, vengono raccolti 10 tempi ottenuti come il tempo passato tra l'inizio degli aggiornamenti e la stampa dei boids, andando poi a calcolare la media.

Tutti i tempi che verranno mostrati hanno come unità di misura i nanosecondi (*ns*) e sono stati calcolati con un sistema avente 12 core.

Versione sequenziale

Innanzitutto, per ottenere un confronto sulle prestazioni del nostro sistema, è stato necessario calcolare i tempi della versione base, ovvero quella sequenziale.

	Numero boids		
Numero giro	**1500**	**2000**	**2500**
1	44282500	57819400	82399200
2	28662100	40719900	55109700
3	24446900	33202500	57721500
4	23512800	36642300	49647300
5	23553400	33156000	44570300
6	23213300	29810100	45009900
7	23230400	30566200	45463300
8	21902000	28897000	47665800
9	22072900	30467200	47277100
10	22180200	29469600	45785800
AVG	**25705650**	**35075020**	**52064990**

Versione multithreaded

	Numero boids		
Numero giro	**1500**	**2000**	**2500**
1	5472700	8267900	9737800
2	5494400	4527800	5162000
3	2751700	4139700	7003800
4	2169500	4194800	5011700
5	2187500	2974000	4572900
6	1933000	3152700	5976600
7	1974000	2961500	4718400
8	4055900	3970900	4811600
9	2166100	3111600	4839600
10	2216700	3281300	5033700
AVG	**3042150**	**4058220**	**5686810**

Versione task-based

	Numero boids		
Numero giro	**1500**	**2000**	**2500**
1	7125100	5471700	6168800
2	3112900	4610300	5302400
3	2481900	3962500	5484800
4	2258600	3647000	4656900
5	2175000	2823400	4896800
6	2411300	3480200	4994500
7	2479900	2848500	5112700
8	2518500	3100500	4898700
9	2203100	3305500	5261700
10	2553600	3258900	5153400
AVG	**2931990**	**3650850**	**5193070**

Versione con virtual threads

	Numero boids		
Numero giro	**1500**	**2000**	**2500**
1	3935500	5463000	11598000
2	3355100	5949100	10971800
3	3737100	5554900	10273600
4	4084700	5700100	10757900
5	5063800	5041500	7657900
6	4566700	5137600	8081100
7	3263500	6088700	9057000
8	4134600	6607300	7056700
9	3771800	5728000	7014700
10	3676400	6766400	7847900
AVG	**3958920**	**5803660**	**9031660**

Risultati

Una volta raccolti tutti i tempi di esecuzione, abbiamo potuto calcolare lo *speedup* e *l'efficienza* tramite le due formule viste a lezione:

$$S = \frac{T_1}{T_N} \qquad E = \frac{S}{N}$$

Dunque, i risultati ottenuti sono i seguenti:

Risultati						
	1500 boids		2000 boids		2500 boids	
Approccio	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**	**Speedup**	**Efficiency**
Multithreaded	8,449829	**0,704152490**	8,642956764	**0,720246397**	9,155394676	**0,762949556**
Task-based	8,767304	**0,730608733**	9,607357191	**0,800613099**	10,02585946	**0,835488288**
Virtual Threads	6,493096	**0,541091383**	6,043603519	**0,503633626**	5,764719885	**0,480393323**

In tutti i casi, gli approcci concorrenti hanno portato ad un miglioramento delle prestazioni rispetto alla versione sequenziale. È curioso vedere come nel caso del multithreaded e del task-based l'efficienza cresce all'aumentare del numero di boid, mentre nel caso dei virtual thread l'andamento è inverso. Questo dimostra come, avendo un numero elevato di boid, l'approccio che prevede di creare un virtual

thread per ognuno sia meno efficiente rispetto ad avere un numero fissi di thread fisici che gestiscono un sottogruppo di boids.

Verifica della parte multithreaded

Per verificare che non ci fossero errori relativi ai thread nella prima parte dell'assignment (deadlock, starvation…), abbiamo utilizzato il template di *Java Path Finder* messo a disposizione dal prof. Aguzzi.
Per testare il codice con *JPF*, abbiamo dovuto astrarre i concetti principali. Innanzitutto, è stata tenuta solo la logica in quanto la grafica non era rilevante per i nostri scopi.
Inoltre, abbiamo creato una classe *DoubleGenerator* la quale restituisce dei valori decimali prefissati. Questa classe è stata usata in *BoidsModel* quando vengono creati i vari boids in quanto la classe *Math.random()* avrebbe creato un elevato indeterminismo.
Dunque, il codice creato per la generazione e la gestione dei *workers* per la parte multithreaded dell'assignment è rimasta identica, l'unica differenza sta nel fatto che non viene più svolto un *while(true)* ma, i vari aggiornamenti, vengono svolti un numero finito di volte.

Questo è il risultato stampato sul terminale da *JPF*: