# Report Assignment-01
# Programmazione Concorrente e Distribuita

Muhamad Huseyn Salman

April 9, 2025

# Contents

# 1 Analysis

Nel paralellizare il codice, principale il problema da affrontare era la gestione delle letture e scritture di ogni boid da una lista condivisa. Ogni boid per aggiornare la propria velocità ha bisogno di far riferimento ai boid vicini, di conseguenza deve leggerne lo stato. Senza nessun tipo di sincronizzazione, c'è il rischio che un worker vada a leggere la velocità di un boid vicino mentre un altro worker lo sta modificando.
E' inoltre necessario che i worker si sincronizzino non solo tra di loro ma anche con la View, dandogli il tempo di disegnare i boid prima di poterli aggiornare all'istante di tempo succesivo.

# 2 Design

## 2.1 State Management

Ogni versione è propensa a tecniche di parallelizzazione diverse, ma la sincronizzazione è piuttosto simile. E' stata adottata un'architettura MVC in cui lo stato di esecuzione della simulazione è incapsulato dentro a degli oggetti monitor `Flag`. Quando uno dei bottoni viene premuto, la View segnala il cambiamento di stato al Controller tramite dei metodi `notify`, che aggiornano lo stato della flag corrispondente (ai bottoni di suspend/resume e reset sono state assegnate flag separate).

Le flag vengono usate ad ogni iterazione dai worker per controllare se continuare o meno l'esecuzione, e dal main per controllare se resettare la simulazione. Nello specifico, i diversi bottoni hanno questo effetto:

- `Suspend/Resume`: nello stato suspend i worker non possono iniziare una nuova iterazione, il bottone reset viene abilitato ed il main controlla attivamente se viene premuto.

- `Reset`: può essere premuto solo quando la simulazione è sospesa e fa terminare i worker, mentre il main crea una lista di boid aggiornata alla nuova dimensione per i nuovi worker.

## 2.2 Concurrency Management

In tutte le versioni, per evitare letture concorrenti a scritture, sono state separate le due operazioni in fasi distinte. Prima di iniziare una nuova fase è

necessario che tutti i worker abbiano terminato la fase precedente. Ad esempio, inizialmente tutti i boid leggono le informazioni necessarie per calcolare la propria velocità, e solo dopo procedono tutti all'aggiornamento del campo. Per ogni boid le fasi sono:

1. Lettura e calcolo della velocità
2. Scrittura della nuova velocità
3. Calcolo e scrittura della nuova posizione
4. Lettura ed aggiornamento della GUI

L'aggiornamento della GUI avviene solo in una fase in cui i boid non vengono modificati, pertanto è stato utilizzato il metodo `SwingUtils.invokeAndWait()` dal main per aggiornare la GUI in modo sincrono, onde evitare che i worker possano modificare i boid mentre la GUI li sta ancora disegnando.

## 2.3 Versions Details

Di seguito vengono evidenziate le differenze tra le singole versioni, considerando che quanto detto finora è valido per tutte le versioni dato che è stato adottato un approccio simile.

**Version 1: Platform Threads** In questa versione sono stati creati 'N' thread (N = numero di core disponibili), ognuno dei quali si occupa di un sottoinsieme di boid. Sono state usate diverse barriere cicliche per mantenere i thread sincronizzati nelle varie fasi di calcolo/lettura e scrittura dei boid.

La barriere utilizzate sono:

1. Barriera per il calcolo della velocità (`counter = N+1`)
2. Barriera per l'aggiornamento della velocità (`counter = N`)
3. Barriera per l'aggiornamento della velocità (`counter = N+1`)

Per rompere la prima e l'ultima barriera è necessario che anche il main si metta in attesa, oltre agli N worker. Questo perché il main si occupa di aggiornare la grafica, quindi rompendo la terza barriera sa che può aggiornare la GUI dato che i thread hanno finito di scrivere le posizioni aggiornate. Invece la prima barriera non può essere rotta finché il main non ha finito di aggiornare la GUI, altrimenti i thread modificherebbero i campi dei boid mentre vengono disegnati.

*JPF Model Checking*: è stato realizzato un modello semplificato del codice, rimuovendo la GUI, semplificando il calcolo della velcoità e rimuovendo l'utilizzo del random nell'inizializzazione dei boid. Non sono stati rilevate corse critiche nell'utilizzo delle barriere. Tuttavia non è stata modellata l'interazione con la View sia per mancanza di tempo sia perché il tempo di esecuzione e la quantità di memoria richiesta da JPF per eseguire cresce esponenzialmente rispetto al numero di istruzioni nel codice.

**Version 2: Executor Service** In questa versione sono stati create delle classi `Task` diverse per le varie fasi della simulazione. In ogni fase viene creato un task per ogni boid. Questi vengono invocati in parallelo tramite un `ExecutorService` di tipo `FixedThreadPool`. Si attende il completamento di ogni task aspettando sull'oggetto `Future` corrispondente e quando tutti i task di una fase sono terminati, si può passare alla fase successiva. Terminate le fasi, si aggiorna la GUI e si ripete il ciclo.

Tipologie di Task creati:

1. Task per il calcolo della velocità
2. Task per l'aggiornamento della velocità
3. Task per l'aggiornamento della posizione

I task vengono interrotti e ricreati solo alla pressione del tasto reset, altrimenti vengono invocati ad ogni iterazione. Alla pressione del tasto suspend, il main non esegue più il codice in cui i task vengono invocati. Alla pressione del tasto reset, vengono creati dei nuovi task in base alla nuova dimensione della lista di boid.

**Version 3: Virtual Threads** Dato che i virtual thread sono molto leggeri e ci permettono di superare il limite di thread fisici del sistema, in questa versione è stato creato un virtual thread per ogni boid, che si occupa dei calcoli e degli aggiornamenti corrispondenti. La sincronizzazione ed il comportamento del sistema alla pressione di un bottone sono analoghe al caso dei thread fisici.

## 2.4 Monitor Implementations

Come richiesto, i meccanismi di sincronizzazione utilizzate sono stati reimplementati da zero, usando solamente Lock e Conditions.

**MyCyclicBarrier:** utilizzando un lock è stata costruita una sezione critica nel metodo await che blocca il thread nel caso in cui il numero di thread già in attesa non sia sufficiente a rompere la barriera. Il thread che rompe la barriera prima la resetta e poi sveglia tutti i thread in attesa. La condizione sul while utilizza una variabile `generation` e non il `counter`, altrimenti ci sarebbe il rischio che un thread controlli la condizione sul counter quando è già stato resettato.

**MyReadWriteLock:** permette letture paralle ma serializza le scritture. Una scrittura può avvenire solamente quando non ci sono letture in corso, in caso contrario viene segnalata una richiesta di scrittura che impedisce a nuovi reader di entrare in sezione critica, evitando in questo modo la starvation del writer. Questa classe è utilizzata internamente dalla classe monitor Flag.

## 2.5 Thread Join Issue

Nella versione 1 e 3 è stato riscontrato un problema nella terminazione dei thread. A differenza dei Task nel caso degli Executor, che terminano ad ogni iterazione e vengono invocati nuovamente in quella successiva, i thread sono stati progettati per terminare solo quando viene premuto il bottone di reset.

Per terminare i thread, inizialmente è stata usata solo la Flag associata al bottone reset. Tuttavia questo non è sufficiente, perché si può verificare il seguente scenario:

1. Alcuni thread iniziano una nuova iterazione (runFlag = true)
2. La simulazione viene messa in pausa (runFlag = false)
3. I rimanenti thread non posso iniziare una nuova iterazione (runFlag = false)
4. I thread del punto 1 sono bloccati sulla prima barriera che non può essere raggiunta dai rimanenti thread

Questo scenario non è un problema se viene fatta ripartire la simulazione. Ma nel caso in cui si voglia farne il reset, i thread che hanno già raggiunto la barriera non potranno mai terminare (fare join sui worker causerebbe un deadlock). Per evitare questo scenario è stata aggiunta all'implementazione della barriera un metodo break() che permette al main di rompere la barriera a comando e risvegliare tutti i thread in attesa. Una volta risvegliati, i thread superano tutte le barriere perché già rotte e notano che la simulazione è in pausa (runFlag = false) e che il bottone reset è stato premuto, quindi terminano.

# 3 Behaviour

Le seguenti Reti di Petri rappresentano il comportamento del sistema a diversi livelli di astrazione. Le piazze rappresentano gli stati che attraversano i worker, mentre i token rappresentano i worker. Queste reti prendono in considerazione il caso di 3 worker ed il main, quindi con 4 token totali.

## 3.1 High-Level Behaviour

Questa prima rete descrive il comportamento iterativo generale del sistema, quindi i vari stati che attraversa durante una normale iterazione, compresi i punti di sincronizzazione (barriere) ed il controllo sulle due flag prima di iniziare una nuova iterazione.

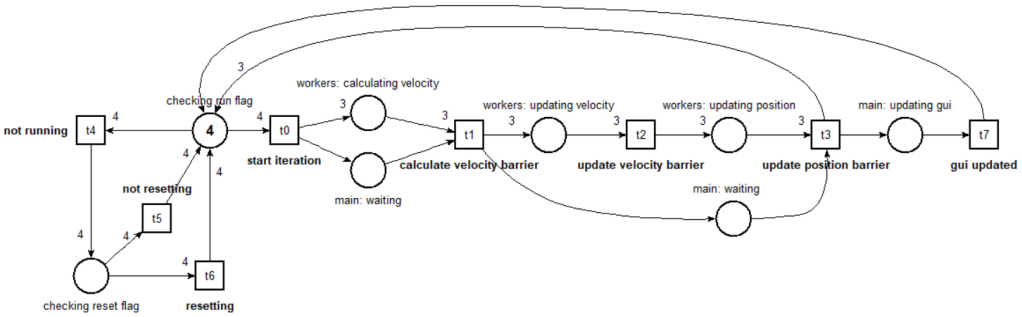


Figure 1: General Behaviour for Threads

Il comportamento del sistema è molto simile anche nella versione basata su Executors e Tasks.

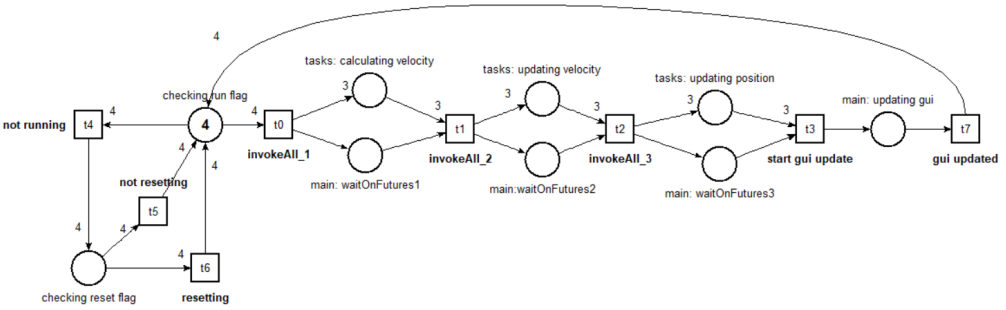


Figure 2: General Behaviour for Tasks

## 3.2 MyCyclicBarrier Behaviour

La seconda rete rappresenta la sezione critica interna alla Barriera. Le barriere sono utilizzate solo nella versione 1 e 3.

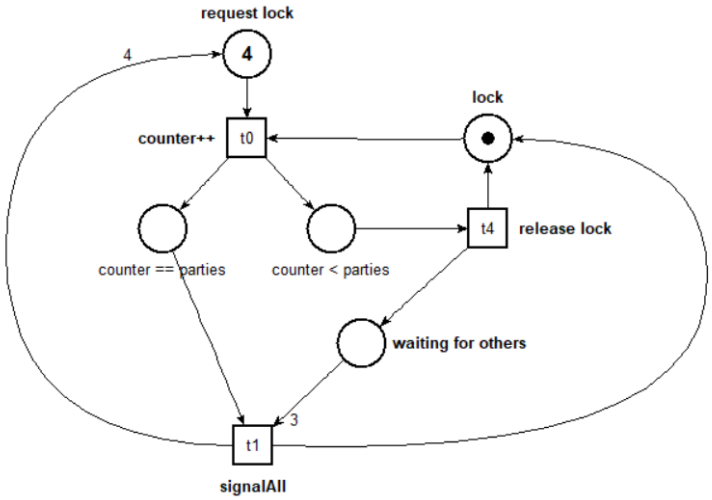


Figure 3: MyCyclicBarrier PetriNet

## 3.3 MyReadWriteLock Behaviour

La terza rete rappresenta il funzionamento interna alla Flag, che utilizza il ReadWriteLock.

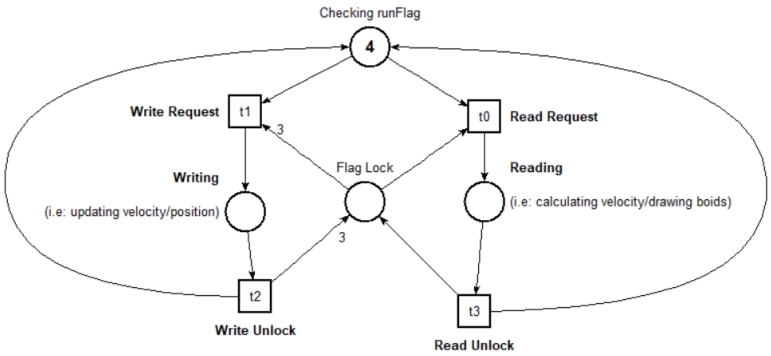


Figure 4: MyReadWriteLock PetriNet

# 4 Performance Measurements

Sono state realizzate delle versioni senza GUI per misurare le performance evitando la `sleep` tra i frame e senza l'overhead grafico.

## 4.1 Speed Up

Nel caso dei thread fisici è stato misurato lo speedup all'aumentare dei thread e confrontato con lo speedup lineare. Possiamo notare che lo speed up si

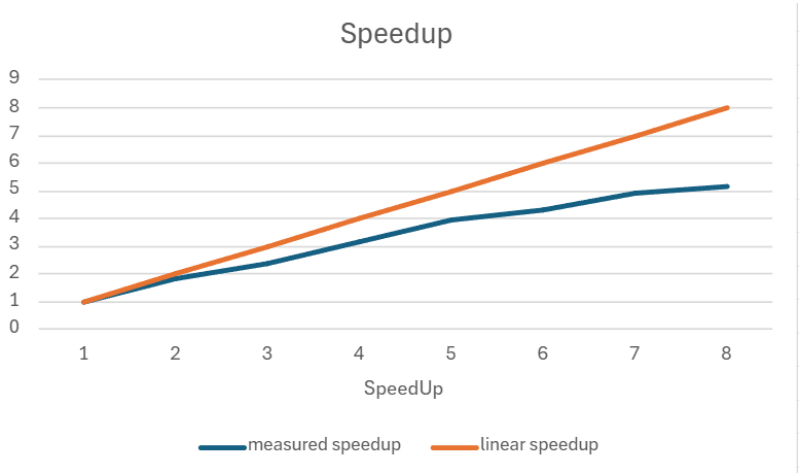


Figure 5: SpeedUp Graph

distacca da quello lineare all'aumentare dei thread probabilmente a causa dei meccanismi di sincronizzazione.

## 4.2 Throughput

Per ogni versione è stato calcolato il throughput, inteso come numero di boid aggiornato in un certo lasso di tempo.

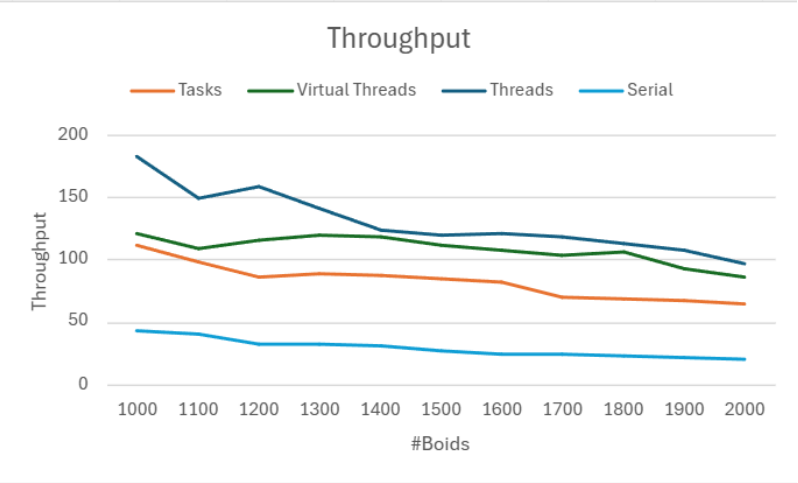


Figure 6: Throughput Graph

In generale all'aumentare dei boid il throughput cala in tutte le versioni, probabilmente a causa dei meccanismi di sincronizzazione ed alla funzione getNearbyBoids che richiede calcoli sempre maggiori.

Come ci si aspettava la versione seriale è quella con il throughput minore. Gli Executor sono i più semplici da implementare e sincronizzare, ma hanno una performance peggiore rispetto a dei thread fisici o virtuali.

I thread fisici sono quelli con throughput maggiore. I virtual thread hanno una performance molto simile ai thread fisici ma con un andamento più stabile.