

Relazione Assignment-01
per l'esame
“Programmazione Concorrente e distribuita”

Gabriele Basigli

9 aprile 2025

Indice

1	Analisi	2
2	Design	4
2.1	Architettura	4
2.2	Approccio Multithread	5
2.3	Approccio Executor Framework	5
2.4	Approccio Virtual Thread	6
3	Comportamento	7
3.1	Multithread e Virtual Thread	7
3.2	Executor	8
4	Test delle performance	9
4.1	Speedup	9
4.2	Efficiency	10
4.3	Throughput	10
4.4	Conclusione	13
5	Verifica con JPF	14

Capitolo 1

Analisi

Il problema consiste nel progettare e sviluppare una versione concorrente della simulazione dei boids. La simulazione deve essere implementata in tre versioni diverse utilizzando tre approcci distinti:

- programmazione multithread in Java
- basato su task utilizzando il Java Executor Framework
- Thread virtuali in Java

La GUI fornisce:

- pulsante Start/Reset Button per avviare la simulazione (Start) e per bloccarla (Reset) riportandola allo stato iniziale
- pulsante Pause/Resume Button mette in pausa o riprende la simulazione in corso
- un campo di input che permette di impostare il numero di boids nella simulazione, l'utente può scrivere un numero e premere Enter per aggiornare il modello, tuttavia questa operazione può essere eseguita solo se il modello è in pausa
- slider che consentono la modifica in tempo reale dei parametri di separazione, allineamento e coesione.

La criticità maggiore riguarda la gestione dell'accesso concorrente alle risorse condivise, in particolare all'insieme dei boids. Per garantire la correttezza della simulazione, il calcolo dello stato dei boids è stato suddiviso in tre fasi, ciascuna delle quali deve essere completata da tutti i boids prima di passare alla successiva:

- **Calcolo della nuova velocità:** ogni boid determina la propria nuova velocità in base alla posizione e al comportamento dei boids vicini, tenendo conto dei pesi relativi a separazione, allineamento e coesione.
- **Aggiornamento della velocità:** al termine della fase precedente, tutti i boids aggiornano il proprio vettore velocità con il valore appena calcolato.
- **Aggiornamento della posizione:** una volta che tutti i boids hanno aggiornato la propria velocità, ciascuno di essi aggiorna la propria posizione nello spazio in base al nuovo vettore velocità.

Capitolo 2

Design

2.1 Architettura

Tutte e tre le versioni della simulazione si basano sull'adozione del pattern MVC (Model-View-Controller). Questa scelta progettuale ha permesso di mantenere separati i diversi livelli di responsabilità, favorendo la modularità e la manutenibilità del codice.

Il Model è rappresentato dall'entità `BoidsModel`, che gestisce lo stato della simulazione, ovvero la posizione e la velocità dei boids, e fornisce le operazioni per aggiornarli. Il modello è indipendente dalle modalità di visualizzazione o interazione.

La View, definita tramite l'interfaccia `BoidsView`, ha il compito di osservare e rappresentare graficamente lo stato del modello.

Il Controller è rappresentato dalle tre diverse implementazioni dell'interfaccia `BoidsSimulator`, ognuna delle quali incapsula una strategia diversa per la gestione della concorrenza (`multithread`, `executor` o `virtual thread`). Ha il compito di orchestrare l'esecuzione della simulazione, gestendo l'avanzamento delle fasi, la sincronizzazione e l'interazione con l'utente tramite flag di pausa (`pause`), `reset` e ripresa (`resume`).

Quando l'utente interagisce con l'interfaccia grafica, ad esempio premendo un pulsante di `start`, `stop`, `resume` o `reset`, la view si limita a modificare lo stato delle Flag corrispondenti. Queste Flag, che fungono da semplici segnali di stato, vengono lette ciclicamente dal controller, il quale agisce di conseguenza. Dal momento che queste Flag sono condivise tra il thread della GUI e i thread (o `virtual thread`) della simulazione, è fondamentale che siano `thread-safe`.

2.2 Approccio Multithread

Un problema emerso nell'utilizzo dei Multithread riguarda la suddivisione del carico di lavoro. L'approccio che prevede l'assegnazione di un thread per ciascun boid si è dimostrato inefficiente, principalmente a causa dell'elevato overhead introdotto dalla gestione di un numero molto elevato di thread. Questo comporta un notevole consumo di risorse e un peggioramento generale delle performance, specialmente al crescere del numero di boid simulati. Per questo motivo, è stato adottato un approccio più efficiente, basato sull'utilizzo di un numero di thread pari al numero di core logici disponibili nella macchina (ottenuto tramite `Runtime.getRuntime().availableProcessors()`), incrementato di una unità per massimizzare l'uso della CPU. I boid vengono quindi suddivisi equamente tra i thread, in questo modo ognuno lavora su un sottoinsieme di boid in maniera sequenziale.

La classe `BoidsParallelSimulator` implementa la logica della simulazione. Si occupa della suddivisione del lavoro tra i thread, della sincronizzazione tra le varie fasi della simulazione e dell'aggiornamento della vista grafica.

La transizione tra una fase (2) e la successiva avviene in modo sincronizzato grazie a un meccanismo a barriera. A tal fine, è stata implementata una versione personalizzata di una `CyclicBarrier`, denominata `MyCyclicBarrier`, costruita mediante l'uso di semafori. Questa consente di sospendere l'avanzamento dei thread partecipanti fino a quando tutti non abbiano raggiunto la barriera, garantendo così che ogni fase venga completata collettivamente prima di procedere oltre. Inoltre, la classe `MyCyclicBarrier` consente opzionalmente l'esecuzione di un'azione personalizzata (`Runnable`) ogni volta che la barriera viene "rotta", ovvero nel momento in cui tutti i thread hanno completato la fase corrente. Questa funzionalità viene sfruttata per aggiornare la GUI una volta che tutte le fasi sono state completate.

Poiché i thread `BoidUpdateWorker` vengono avviati una sola volta ma possono ricevere una nuova partizione di boid anche durante l'esecuzione (tramite il metodo `setBoidsPartition()`), è necessario garantire che ogni thread veda immediatamente l'ultima versione aggiornata della lista, un mancato aggiornamento visibile potrebbe portare il thread a lavorare su una partizione obsoleta o errata. A tal scopo il campo privato della classe che contiene la partizione di boid è stato reso "volatile".

2.3 Approccio Executor Framework

Questo approccio rispetto al precedente risulta essere meno complesso da implementare. Il numero di worker nel pool è stato scelto dinamicamente,

in base al numero di core logici disponibili sulla macchina, incrementato di una unità (`Runtime.getRuntime().availableProcessors() + 1`), al fine di massimizzare l'utilizzo delle risorse disponibili, come nel caso `multithread`. A differenza dell'approccio precedente, in cui ogni thread si occupava di un sottoinsieme di boid, in questa versione ogni boid è associato a tre diverse `Callable`, ciascuna responsabile di una delle tre fasi del ciclo di simulazione:

- Calcolo della nuova velocità (`ComputeVelocityTask`)
- Aggiornamento della velocità (`UpdateVelocityTask`)
- Aggiornamento della posizione (`UpdatePositionTask`)

Tutte le task relative a una fase vengono invocate contemporaneamente tramite il metodo `invokeAll`, che blocca il thread principale fino al completamento di tutte le operazioni.

Il metodo ausiliario `waitForCompletion()` è responsabile del blocco esplicito in attesa del completamento di ciascuna `Future` tramite la chiamata `get()`. In questo modo si garantisce la sincronizzazione tra le fasi, rispettando la logica della simulazione. Inoltre, il controller (`BoidsExecutorSimulator`) consente di rigenerare dinamicamente il pool di thread e ricreare le task in seguito a modifiche del numero di boid, oppure in caso di reset della simulazione. Infine, la GUI viene aggiornata periodicamente al termine di ogni iterazione.

2.4 Approccio Virtual Thread

In questa ultima versione viene adottato un approccio basato sui `Virtual Thread` di Java. Inizialmente la soluzione adottata consisteva nell'utilizzo di un virtual thread per ogni boid. Ogni virtual thread esegue in parallelo le operazioni di aggiornamento dello stato del singolo boid. Questo procedimento si è rivelato essere estremamente inefficiente in termini di performance. Il nuovo approccio si ispira direttamente alla struttura della versione `multithread` classica, ma utilizza `virtual threads`. In questa versione, i boid vengono suddivisi in partizioni statiche, assegnate a un numero fisso di worker thread (nel caso specifico, 40 thread virtuali, numero ottenuto attraverso l'analisi delle prestazioni 4). Ogni worker è responsabile dell'elaborazione di un sottoinsieme dei boid durante tutte le fasi della simulazione: calcolo della velocità, aggiornamento della velocità e aggiornamento della posizione. La sincronizzazione tra le fasi è garantita tramite le stesse barriere introdotte in precedenza. Questa soluzione si è dimostrata molto più performante della precedente implementazione naïve a `virtual threads`, avvicinandosi, e in alcuni casi superando, le prestazioni della versione basata su thread tradizionali.

Capitolo 3

Comportamento

3.1 Multithread e Virtual Thread

La versione multithread e virtual thread hanno un comportamento identico, la rete in figura 3.1 descrive in generale il comportamento del sistema.

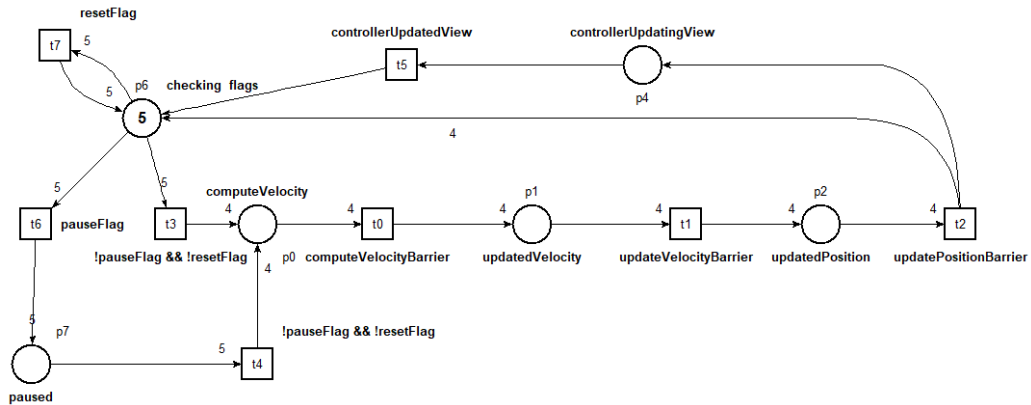


Figura 3.1: Comportamento del sistema Multithread e Virtual Thread

3.2 Executor

Questa versione, sebbene differisca dalle due precedenti ha un comportamento molto simile.

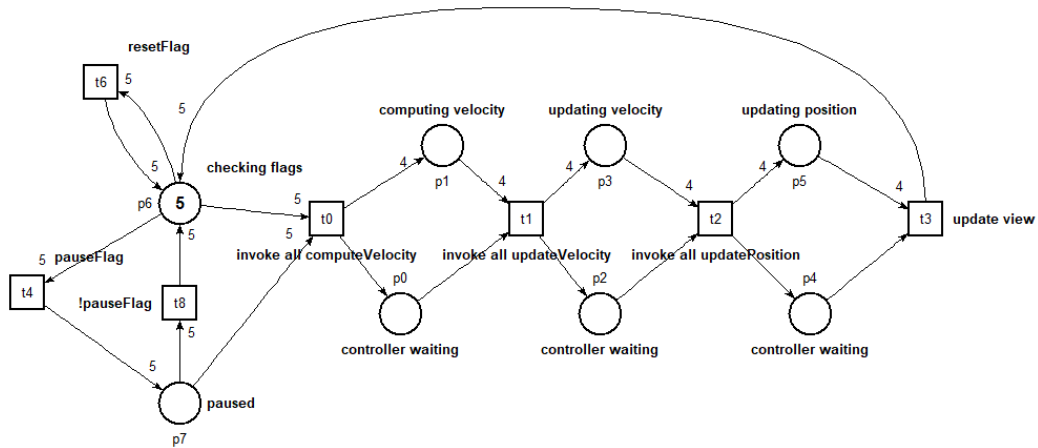


Figura 3.2: Comportamento del sistema

Capitolo 4

Test delle performance

Tutti i test sono stati eseguiti mantenendo costanti i seguenti parametri: numero di iterazioni pari a 1000 e numero di boid pari a 3500. Inoltre, per evitare che il rendering grafico influenzasse negativamente le prestazioni misurate, tutti i test sono stati eseguiti in assenza di GUI.

Tutti i test di performance sono stati svolti sulla stessa macchina:

MacBook Pro 13-inch 2020.

Processore 2 GHz Intel Core i5 quad-core 8 thread

Memoria 16 GB 3733 MHz LPDDR4

4.1 Speedup

Il parametro di riferimento per la valutazione delle prestazioni è lo speedup (figura 4.1), definito dalla seguente formula

$$Speedup = \frac{T_{sequenziale}}{T_{parallelo}}$$

A causa dell'impossibilità di intervenire manualmente, a livello di codice, sul numero effettivo di thread utilizzati nella versione basata su Virtual Threads, lo speedup calcolato per questa versione è in funzione del numero dei virtual thread.

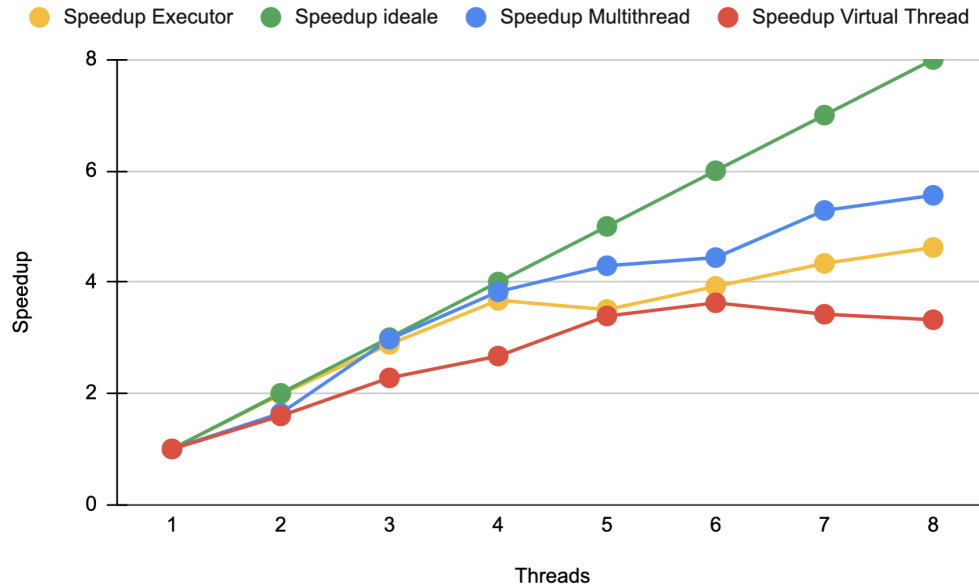


Figura 4.1: Speedup a confronto

4.2 Efficiency

A seguito del calcolo dello speedup è stato calcolato il parametro di efficiency (figura 4.2) nel seguente modo:

$$Efficiency = \frac{Speedup}{N_{processori}}$$

Questo parametro indica "quanto" efficacemente vengono sfruttati i processori. Il valore ideale è 1, raggiunto quando tutti i processori vengono sfruttati appieno.

4.3 Throughput

Il terzo parametro calcolato, è il throughput (figura 4.3), ovvero il numero di iterazioni (o aggiornamenti) completati nell'unità di tempo.

Per la terza versione, il throughput risulta particolarmente interessante da analizzare in quanto, utilizzando 40 virtual thread, si osserva un netto miglioramento delle prestazioni rispetto alle altre versioni. In figura 4.4 è riportato il confronto del throughput tra la versione sequenziale e quella basata su virtual thread, con un numero fisso di 40 thread. In figura 4.5 viene invece mostrato il confronto dei tempi di esecuzione tra le due versioni.

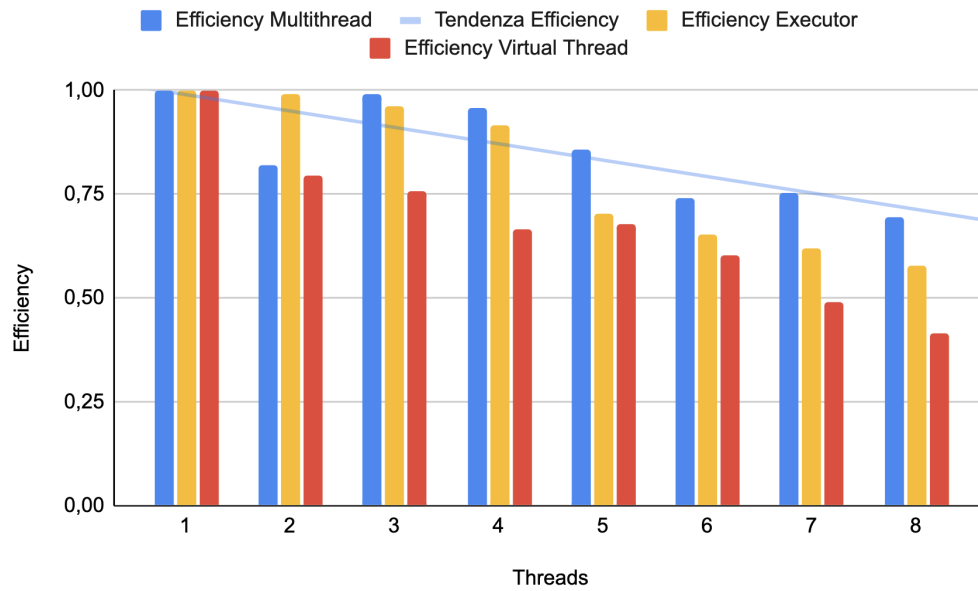


Figura 4.2: Efficiency a confronto

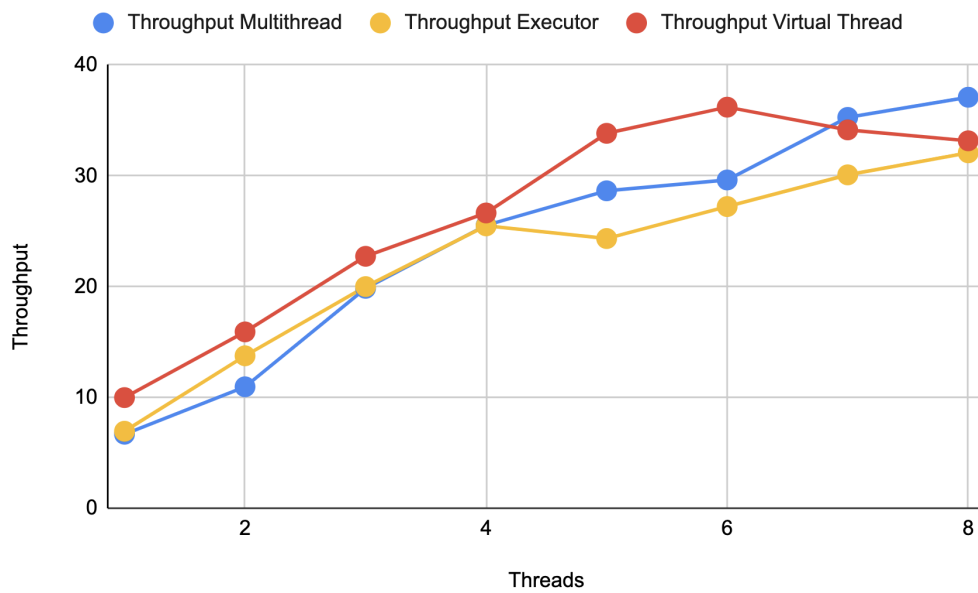


Figura 4.3: Throughput a confronto

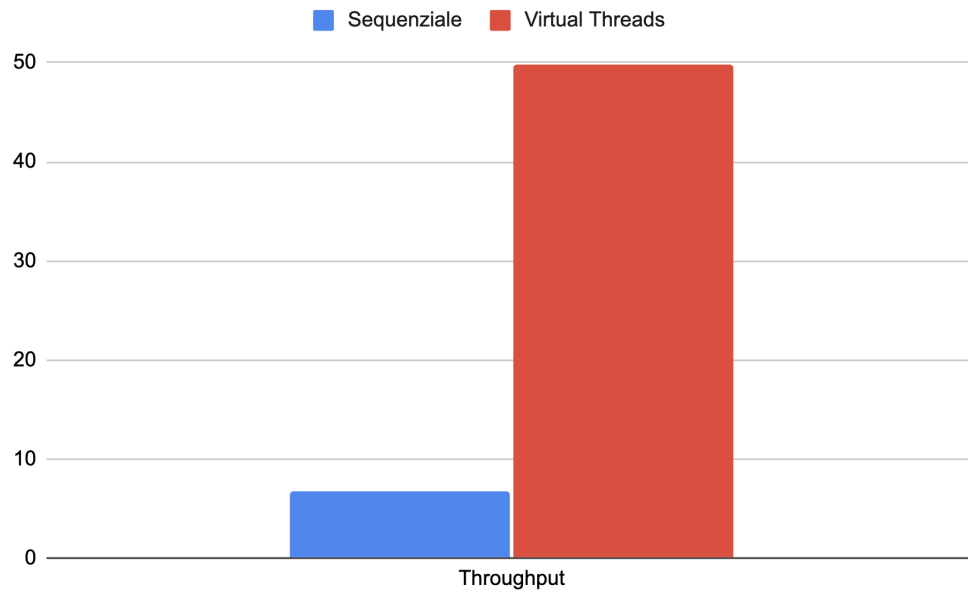


Figura 4.4: Versione 40 Virtual Threads

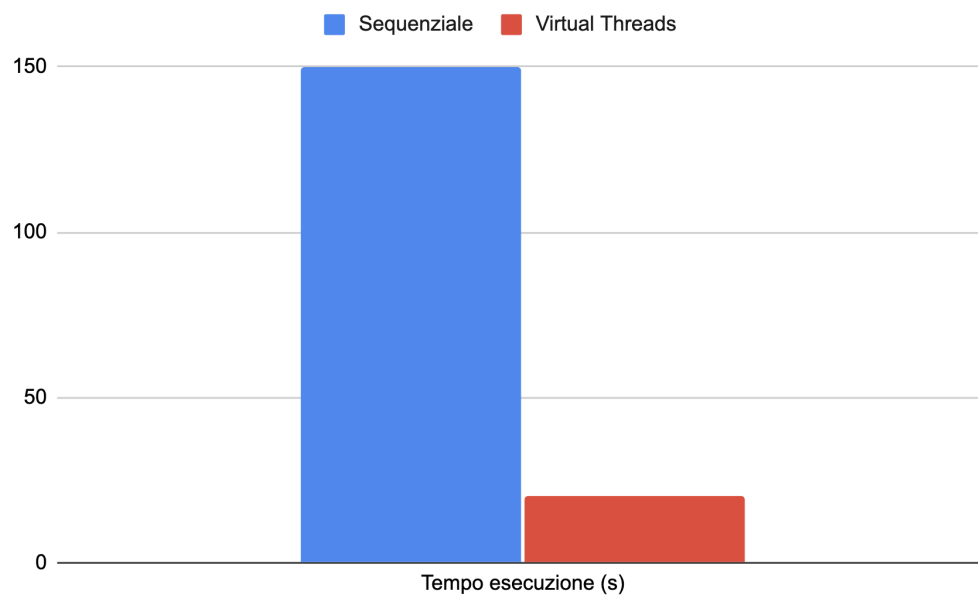


Figura 4.5: Tempo di esecuzione 40 Virtual Threads

4.4 Conclusione

Dall'analisi dei risultati sperimentali emergono considerazioni interessanti in merito all'efficienza e alla scalabilità delle tre soluzioni proposte.

La versione multithread, si è rivelata la più performante. Tuttavia, presenta una maggiore complessità implementativa, sia in termini di gestione della sincronizzazione che nella ripartizione delle entità da elaborare.

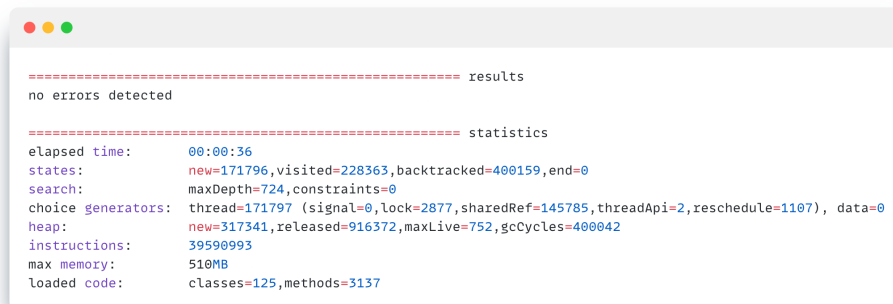
La versione con `ExecutorService` rappresenta un buon compromesso tra prestazioni e semplicità di sviluppo. L'utilizzo di un pool di thread consente di evitare la creazione esplicita e continua di thread, semplificando la gestione e rendendo il codice più modulare. Sebbene leggermente meno performante rispetto all'approccio multithread puro, questa soluzione garantisce comunque risultati soddisfacenti.

Infine, la versione basata su virtual thread, inizialmente meno efficace rispetto alle alternative, ha mostrato notevoli miglioramenti quando ottimizzata con un numero fisso di 40 virtual thread.

Capitolo 5

Verifica con JPF

Al fine di identificare eventuali corse critiche, la prima versione è stata testata con lo strumento JPF. Per ottenere dei risultati in un tempo ragionevole, la versione sottoposta al test è stata "snellita". Il risultato è il seguente



```
===== results
no errors detected

===== statistics
elapsed time:      00:00:36
states:           new=171796,visited=228363,backtracked=400159,end=0
search:           maxDepth=724,constraints=0
choice generators: thread=171797 (signal=0,lock=2877,sharedRef=145785,threadApi=2,reschedule=1107), data=0
heap:             new=317341,released=916372,maxLive=752,gcCycles=400042
instructions:     39590993
max memory:       510MB
loaded code:      classes=125,methods=3137
```

Figura 5.1: Risultato JPF

Eseguendo il test con JPF risulta che il modello non presenta errori. Pertanto non sono presenti corse critiche.