Assigment #1 - "The Game of Life"

Martina Magnani
martina.magnani8@studio.unibo.it

Nicola Piscaglia
nicola.piscaglia2@studio.unibo.it

Mattia Vandi
mattia.vandi@studio.unibo.it

1 Analisi del problema

L'obiettivo della consegna è implementare, in Java, una versione concorrente del gioco *"The Game of Life"* usando la programmazione concorrente multi-threaded.
Il gioco consiste nel calcolare e visualizzare l'evoluzione della matrice di celle che caratterizza il gioco, come sequenza di fotogrammi (ognuno dei quali rappresenta lo stato del mondo).
Nella matrice, ogni cella può essere in uno dei due stati possibili, *live* o *dead*.
Dato lo stato $s(t)$ della matrice, lo stato $s(t+1)$ si computa con le seguenti regole:

- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha zero o al più una cella vicina *live* (e le altre *dead*), nello stato $s(t+1)$ diventa *dead* ("muore di solitudine")
- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha quattro o più celle vicine *live*, nello stato $s(t+1)$ diventa *dead* ("muore di sovrappopolamento")
- una cella $m[i,j]$ che nello stato $s(t)$ è *live* e ha due o tre celle vicine *live*, nello stato $s(t+1)$ rimane *live* ("sopravvive")
- una cella $m[i,j]$ che nello stato $s(t)$ è *dead* e ha tre celle vicine *live*, nello stato $s(t+1)$ diventa *live*

Il gioco deve presentare un'interfaccia grafica con pulsanti `start` e `stop` con cui si fa partire e fermare il gioco. Ogni stato del gioco deve essere visualizzato, insieme al numero di celle nello stato *live*.

1.1 Requisiti del sistema

- Il programma deve funzionare anche con matrici di dimensioni significative (ad es. 5000x5000);
- Massimizzare il throughput, minimizzando il tempo di calcolo di ciascun fotogramma ed eventualmente anche della sequenza di fotogrammi;

- Massimizzare la reattività della GUI;
- Studiare e implementare meccanismi di coordinazione/sincronizzazione basati su semafori o monitor.

2 Descrizione della soluzione proposta

2.1 Architettura del sistema

L'architettura del sistema è stata scomposta in tre livelli utilizzando il pattern Model View Presenter con modello ad Interattori, di conseguenza vi sono tre moduli: il dominio applicativo, gli interattori e l'interfaccia grafica:

- Nel primo livello vi è una rappresentazione object-oriented delle componenti del problema;
- Nel secondo livello è stata incapsulata la logica di aggiornamento del gioco;
- Nel terzo livello viene organizzata l'interfaccia grafica che permette la visualizzazione del gioco e l'interazione con l'utente.

2.2 Implementazione

2.2.1 Dominio applicativo

Abbiamo implementato il dominio applicativo attraverso due classi: `Cell` e `Board`.
L'enumerazione `Cell` rappresenta lo stato di una cella della scacchiera che può essere *live* o *dead*. La classe `Board` rappresenta la scacchiera di gioco. Questa è caratterizzata da un'*altezza* e una *larghezza* variabili e dalla possibilità di recuperare/impostare lo stato di una cella date le coordinate x e y.

2.2.2 Interattori

Abbiamo implementato un interattore `BoardUpdater` che si occupa dell'aggiornamento della scacchiera di gioco. Di questo interattore ne esistono due versioni, una sequenziale e una concorrente.

La versione sequenziale (`SequentialBoardUpdater`) l'aggiornamento della schermata di gioco viene eseguito sul thread corrente senza la creazione di nuovi thread.

Nella versione concorrente `ConcurrentBoardUpdater` è possibile configurare il numero di `Worker` che verranno utilizzati per aggiornare la schermata di gioco (ognuno dei quali verrà eseguito su un thread indipendente).
Il metodo `start` si occupa della creazione di un nuovo thread per ogni `Worker` e di metterlo in esecuzione.
Il metodo `stop` si occupa di terminare l'esecuzione dei `Worker`.
Il metodo `update` gestisce l'aggiornamento della scacchiera di gioco (board). Per prima cosa si occupa di creare una nuova scacchiera che rappresenterà lo stato della scacchiera corrente dopo l'aggiornamento. Successivamente divide il lavoro fra i `Worker` cercando di bilanciarlo il più possibile. Dopodiché sospende la propria esecuzione in attesa che tutti i `Worker` abbiano terminato l'aggiornamento della porzione di scacchiera assegnatagli.

Ogni `Worker`, appena messo in esecuzione, aspetta che gli venga assegnata la porzione di scacchiera da aggiornare. Una volta terminata l'attesa, se ancora in esecuzione, effettua l'aggiornamento delle celle seguendo le regole del gioco (indicate nell'analisi del problema). Una volta terminato il lavoro, il `Worker` lo segnala al `CuncurrentBoardUpdater` di aver terminato l'aggiornamento della propria porzione di scacchiera e si mette in attesa di essere eseguito nuovamente.

Per ogni `Worker` la sincronizzazione con `CuncurrentBoardUpdater` è effettuata tramite due semafori binari: il primo (`startUpdate`) viene utilizzato per segnalare al `Worker` che può iniziare l'aggiornamento della propria porzione di scacchiera, il secondo (`finishedUpdate`) viene utilizzato per segnalare al `CuncurrentBoardUpdater` che il `Worker` ha terminato l'aggiornamento della sua porzione di scacchiera.

In questo modo `ConcurrentBoardUpdateer` segnala ad ogni `Worker` che la relativa porzione di scacchiera è pronta per essere aggiornata. Così attende che ogni `Worker` finisca il proprio lavoro per poi restituire la nuova scacchiera.

2.2.3 Interfaccia utente

L'interfaccia utente è stata realizzata con Java FX ed è costituita da due schermate: nella prima si permette all'utente di settare le dimensioni della scacchiera di gioco (altezza e larghezza) ed il numero di `Worker`. Nella seconda si visualizza la scacchiera e sono presenti due pulsanti *start* e *stop*. Lo *start* permette l'avvio e la pausa del gioco, mentre lo *stop* la terminazione.
Utilizzando il pattern Factory sono stati incapsulate in una apposita classe le funzioni per la costruzione delle finestre grafiche. Le callback di queste finestre sono definite da opportuni `Presenter` che incapsulano la logica di controllo dei componenti (form, pulsanti, ...) di cui sono costituite.
La computazione relativa alla coordinazione dell'aggiornamento matriciale e grafico della scacchiera è effettuata su un thread separato `GameUpdater` (task di Java FX in cui è incapsulato il Game Loop) e questo permette di mantenere l'interfaccia grafica (gestita dal Java FX Application Thread) reattiva ed in grado di intercettare l'input dell'utente.
Il Presenter addetto alla gestione degli eventi di gioco (`GamePresenter`) utilizza un servizio di rendering offerto dalla classe `RenderingService` nella quale sono state incapsulate le funzioni relative al disegno della scacchiera di gioco. Quest'ultima è stata realizzata utilizzando un componente grafico `Canvas` sul quale vengono disegnate le celle attraverso la classe `PixelWriter` di Java FX: una classe ottimizzata per il disegno dei singoli pixel.

3 Dinamica del sistema

La dinamica del sistema è stata rappresentata formalmente con le Reti di Petri. Le piazze (*places*) del livello corrispondono ai `Worker` che concorrono all'aggiornamento della scacchiera di gioco; in questo esempio è stato scelto di mostrare il sistema nel caso se ne utilizzino 4.

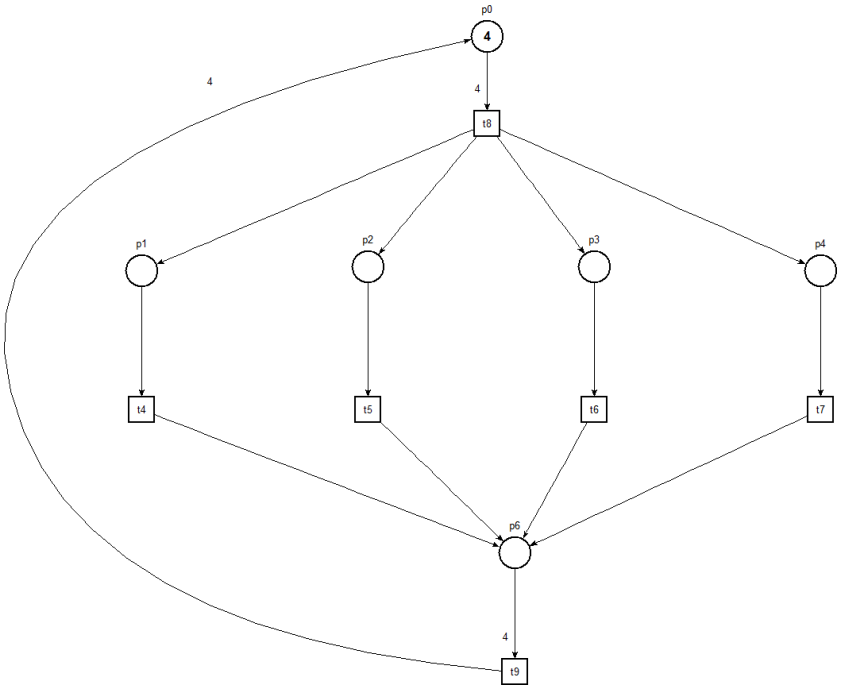

Figura 1: Petri Net che modella la dinamica della computazione dello stato di gioco

4 Analisi delle prestazioni

La tabella sottostante mostra gli speedup S (misura quantitativa delle performance) che il sistema raggiunge a seconda del numero di worker utilizzati.
Lo speedup viene misurato calcolando il rapporto tra il tempo di esecuzione del sistema utilizzando un algoritmo sequenziale e il tempo di esecuzione del sistema utilizzando un algoritmo parallelo.

$$S = \frac{T_1}{T_N} \tag{1}$$

Dove:

- $N \leftarrow$ numero di worker utilizzati
- $T_1 \leftarrow$ tempo di esecuzione del sistema utilizzando un algoritmo sequenziale
- $T_N \leftarrow$ tempo di esecuzione del sistema utilizzando un algoritmo parallelo con N worker

	2	3	4	5	6	7	8	9	10
Min.	1.77	2.43	3.04	2.93	2.89	2.90	2.86	2.85	2.85
Max.	2.11	2.87	3.41	3.22	3.40	3.26	3.38	3.06	3.45
Avg.	1.81	2.50	3.06	2.94	2.93	2.91	2.92	2.85	2.94

Tabella 1: Lo speedup è stato calcolato su 25 iterazioni considerando una schacchiera di gioco 5000x5000. Sulle colonne sono specificati il numero di worker utilizzati per la parallelizzazione dell'algoritmo, sulle righe i valori statistici (minimi, massimi e medi) degli speedup calcolati. Gli speedup sono stati calcolati su un Apple Macbook Pro (Retina, 15", metà 2015) dotato di un processore Intel Core i7 quad-core a 2.2GHz. È possibile eseguire i benchmark lanciando la classe `pcd.ass01.Benchmark` e visualizzare sia le velocità di computazione di uno stato della scacchiera, sia i relativi speedup ottenuti.

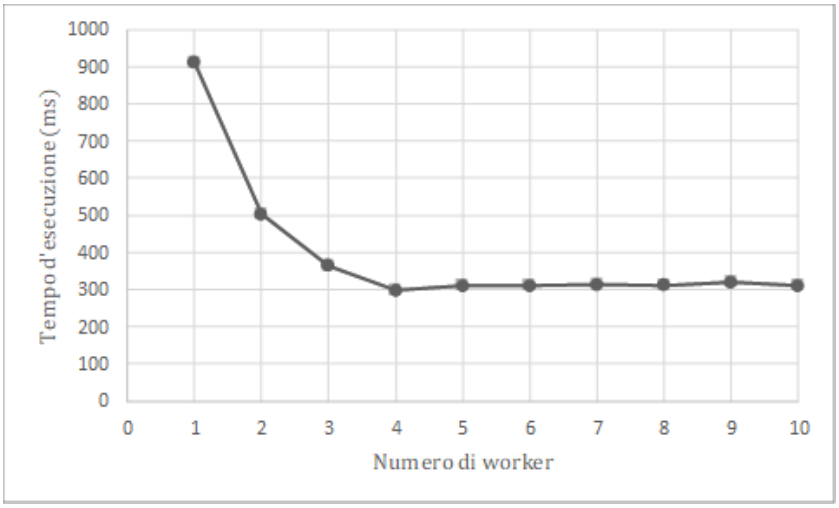

Figura 2: Grafico che mostra l'evoluzione dei tempi di esecuzione (in ms) al crescere del numero di worker. I tempi di esecuzione sono stati calcolati come media su 25 iterazioni considerando una schacchiera di gioco 5000x5000. I tempi di esecuzione sono stati calcolati su un Apple Macbook Pro (Retina, 15", metà 2015) dotato di un processore Intel Core i7 quad-core a 2.2GHz.