marco.canducci@studio.unibo
08/04/2018

Assignment #1
The Game of Life

Gruppo di lavoro

Il progetto è stato sviluppato cooperativamente da tre persone: il sottoscritto Canducci Marco (833069), Schiavi Daniele (843692) e Gondolini Monica (855202). Come consigliato dal docente, non c'è stata una separazione dei compiti tra i vari componenti del gruppo, ma si è cercato di progredire insieme, approfittando della presenza dei compagni per valutare e discutere proposte e scelte progettuali: lavorando in questo modo, si ritiene di esser stati spronati ulteriormente a fare del proprio meglio, poiché si è sempre cercato di trovare una motivazione valida per ogni nuova idea / modifica / implementazione prima di presentarla agli altri membri, convincerli della sua validità, e metterla quindi in atto.

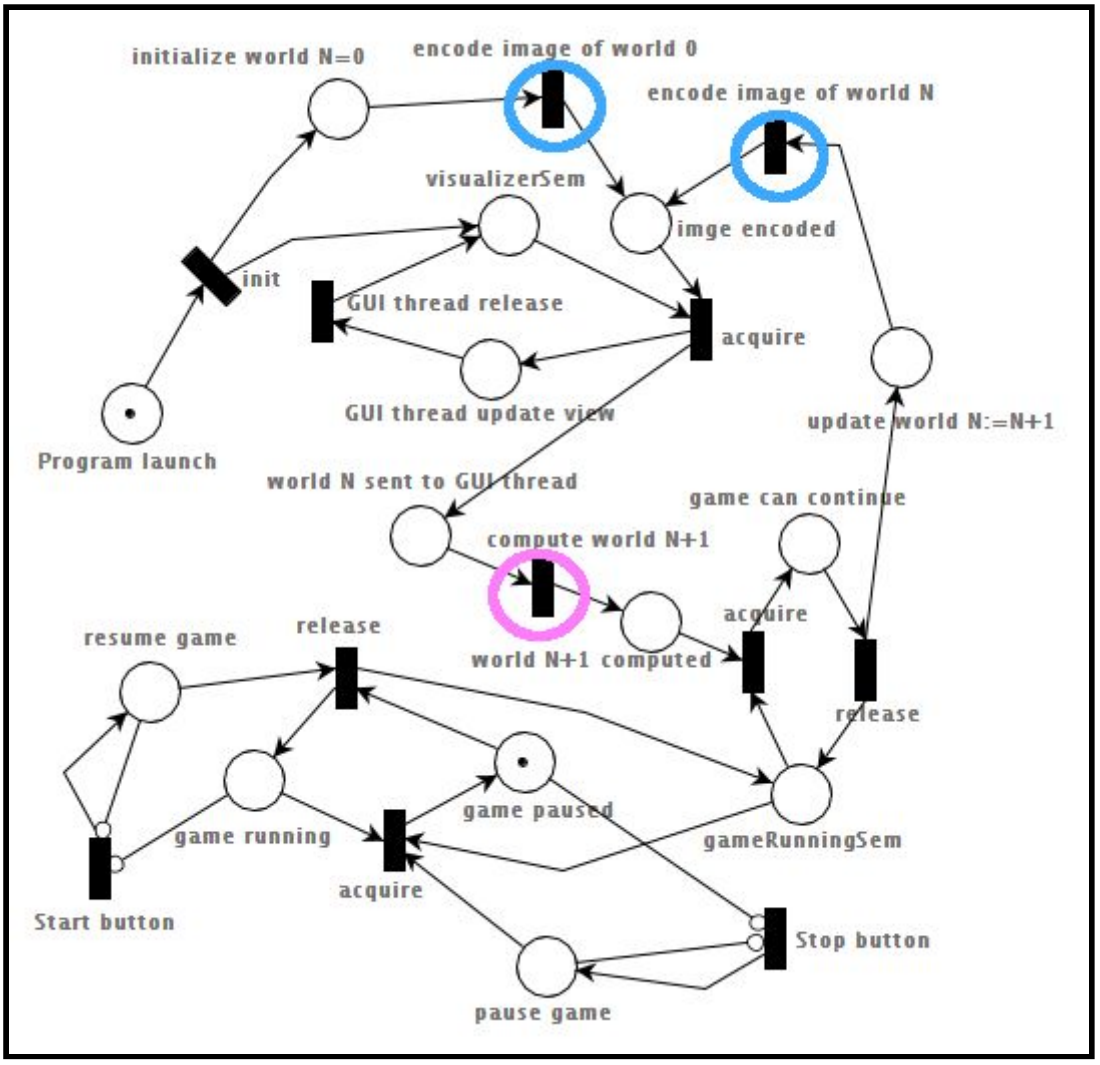
Analisi del problema

Per prima cosa (con l'ausilio della rete di Petri riportata di seguito) si vuole illustrare sommariamente il funzionamento generale del programma, focalizzandosi in particolar modo sui meccanismi di **sincronizzazione tra la GUI e lo stato di avanzamento del gioco**. Verranno poi brevemente descritte le caratteristiche di funzionamento e di configurazione dei due semafori presenti in questa rappresentazione ad alto livello del sistema.

Successivamente si scenderà più nel dettaglio, andando a *zoomare* idealmente sui cerchietti azzurri e rosa disegnati sulla prima rete, espandendo tali sezioni in una ulteriore rete di Petri: quest'ultima descriverà la modalità di **sincronizzazione del thread principale con i suoi *Worker*,** introdotti con lo scopo di parallelizzare il più possibile il carico di lavoro e, conseguentemente, massimizzare il *throughput*.
NB: ad un colore diverso corrispondono sottotipi di *Worker* diversi (quelli atti alla computazione dello stato successivo di gioco, nel cerchietto azzurro, e quelli atti alla codifica dell'immagine *bitmap*, nel cerchietto rosa). Tuttavia, essendo la loro sincronizzazione gestita allo stesso modo, si è ritenuto sufficiente riportare un'unica rete, rappresentativa di entrambi gli scenari.

Infine, si riporterà molto concisamente il modo in cui si è deciso di **suddividere il lavoro tra i vari *Worker***, presentando una semplice, esplicativa, immagine d'esempio.

Rete di Petri principale: Main



Al lancio dell'applicazione viene inizializzato il mondo di gioco (allo stato/era N=0) ed immediatamente codificato in un'immagine *bitmap*. Tale immagine viene quindi fornita al **thread della GUI** (*FX Application Thread*) che avrà il compito di applicarla all'interno del pannello munito di *scroll bars*. Solo una volta applicata l'immagine, l'*FX Application Thread* rilascerà il semaforo ***visualizerSem***, utilizzato per sincronizzare la visualizzazione del mondo corrente sulla GUI con l'evoluzione del mondo di gioco (per i dettagli sulle motivazioni e sull'utilizzo di tale semaforo si faccia riferimento al sottocapitolo dedicato).

Nel frattempo, per aumentare la successiva **reattività della GUI**, è possibile calcolare lo stato del mondo N+1 già da prima che venga premuto il pulsante *Start*. Una volta terminata la computazione si attende, nel caso non si sia già verificato, che l'utente prema appunto il pulsante Start e venga di conseguenza rilasciato il semaforo *gameRunningSem* (si faccia riferimento al sottocapitolo dedicato per ulteriori dettagli sul suo funzionamento e sulle motivazioni che hanno portato al volerlo configurare come **semaforo *strong***).

Premendo il pulsante *Start* mentre il gioco non è nello stato "*running*" si rilascia il semaforo *gameRunningSem*, il quale permetterà al *Game Thread* di proseguire / iniziare l'evoluzione del mondo di gioco (eccezion fatta per il caso in cui l'utente prema il pulsante di *Stop* prima che il *Game Thread* abbia acquisito il semaforo, poiché alla pressione di tale tasto sarà il *thread* della GUI a tentare l'*acquire*). Si è tentato di modellare questo meccanismo sulla rete di Petri sfruttando degli **archi inibitori** e, dopo aver fatto qualche prova, ci si ritiene soddisfatti del risultato ottenuto nonostante non si avesse esperienza pregressa a riguardo.

Una volta acquisito il semaforo, il *Game Thread* lo si rilascerà **immediatamente** e ritenterà di acquisirlo nuovamente al giro successivo. Si vuole riporre particolare enfasi sul fatto di voler rilasciare il semaforo il prima possibile perché, anche in questo caso, si ritiene che tale dettaglio possa aumentare ulteriormente la **reattività della GUI**: nel frattempo l'utente potrebbe aver premuto il pulsante di Stop, e l'*FX Application Thread* potrebbe quindi stare cercando di acquisire il semaforo per mettere in pausa il gioco. In una soluzione precedente (dove il semaforo non veniva rilasciato immediatamente dopo l'acquisizione) si creava un *Thread* a parte (da quello della GUI) ogni volta che veniva premuto il pulsante di Stop per non tenere in attesa l'*FX Application Thread*. Essendo però dispendioso creare e distruggere *Thread*, si è optato per una soluzione che cercasse di evitare tale problema e, rendendosi conto che non ci fosse alcuna motivazione per il *Game Thread* di mantenere il possesso del *gameRunningSem*, si è ritenuto opportuno farglielo rilasciare il prima possibile e farlo quindi eventualmente acquisire direttamente dall'*FX Application Thread* (senza il bisogno di creare altri *thread*).

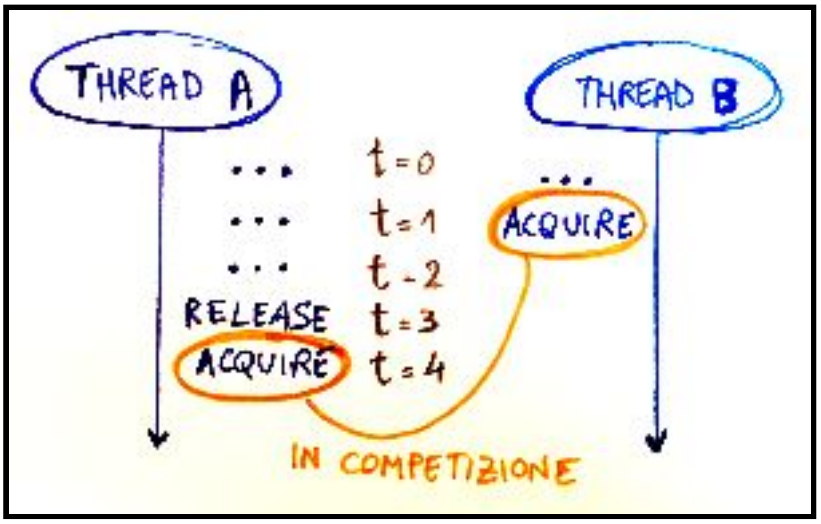
Arrivati a questo punto (sulla rete di Petri alla piazza "*update world N:= N+1*") si sostituisce al mondo corrente quello calcolato in precedenza dai *GameWorkers* e lo si fornisce ai *ControllerWorkers*, i quali avranno il compito di codificarlo in un'immagine *bitmap* rappresentata da un *array* di *byte*.

Di seguito ci si sincronizza con la rappresentazione a video del mondo precedente: si desidera accertarsi che quest'ultimo sia stato visualizzato prima di proseguire (per ulteriori spiegazioni su tale scelta si veda il sottocapitolo dedicato al *Semaforo "visualizerSem"*). Una volta quindi acquisito il semaforo *visualizerSem* (rilasciato precedentemente dall'*FX Application Thread*) si richiede la visualizzazione della nuova immagine appena codificata proprio all'*FX Application Thread*, che la applicherà sul pannello della GUI appena possibile, dopodichè rilascerà il semaforo *visualizerSem* per il giro successivo.

Infine, ci si prepara computando il mondo N+1 e richiedendo solo successivamente il permesso di poter continuare il gioco al *gameRunningSem*, ricominciando eventualmente tutto il giro daccapo. **NB**: essendo la computazione del mondo N+1 considerata onerosa in termini di tempo, si è deciso di richiedere il semaforo solo successivamente, in modo tale che se l'utente dovesse premere il pulsante di Stop durante la computazione (scenario probabile, visto il tempo richiesto dalla computazione) la **GUI risulterebbe molto reattiva** poiché fermerebbe immediatamente la sequenza di immagini visualizzate.

Semaforo "gameRunningSem"

Si è ritenuto necessario configurare questo semaforo come ***Strong*** (o, per utilizzare la terminologia di Java, ***Fair***) poiché durante lo sviluppo ci si è accorti della possibile eventualità descritta nello schema sotto-riportato: è stato verificato che un'*acquire* effettuata in seguito ad una release sullo stesso flusso di esecuzione potrebbe andare in competizione con *acquire* effettuate da altri flussi di controllo sullo stesso semaforo prima della *release*.



Nella figura il *Thread A* rappresenta una vecchia versione del *Game Thread* (dove l'*acquire* del *gameRunningSem* accadeva nell'istruzione immediatamente successiva alla sua *release*), mentre il *Thread B* rappresenta il flusso di esecuzione che, in seguito alla pressione del tasto Stop, cerca di fermare il gioco con un'*acquire*. In tale scenario, utilizzando semafori *weak*, si aveva uno sgradevolissimo effetto ritardo alla pressione del tasto Stop: l'evoluzione del mondo di gioco non si fermava istantaneamente sulla GUI, ma anzi continuava per un numero casuale di ere prima di arrestarsi.

Si vuole infine sottolineare come nell'implementazione finale questo scenario risulti altamente improbabile (poichè ora si susseguono numerose operazioni tra la *release* e la successiva *acquire* del *Thread A*), ma si ritiene fondamentale non fare alcun tipo di assunzione sull'evoluzione dei due flussi di controllo paralleli, e si è quindi scelto di utilizzare comunque un semaforo *strong* per garantire ***fairness*** a prescindere.

Semaforo "visualizerSem"

Personalmente si ritiene che non compromettere la **reattività della GUI**, sarebbe meglio mantenere sincronizzati il mondo corrente del "*model*" con quello visualizzato dalla GUI. Infatti, nel caso si lasciasse proseguire l'evoluzione del mondo a prescindere da quanto visualizzato, potrebbero accumularsi le immagini inserite in coda al "*Platform.runLater()*" e, di conseguenza, alla pressione del tasto STOP si potrebbe ottenere uno sgradevole effetto ritardo, causato proprio dalla inevitabile visualizzazione delle immagini accumulate in precedenza.

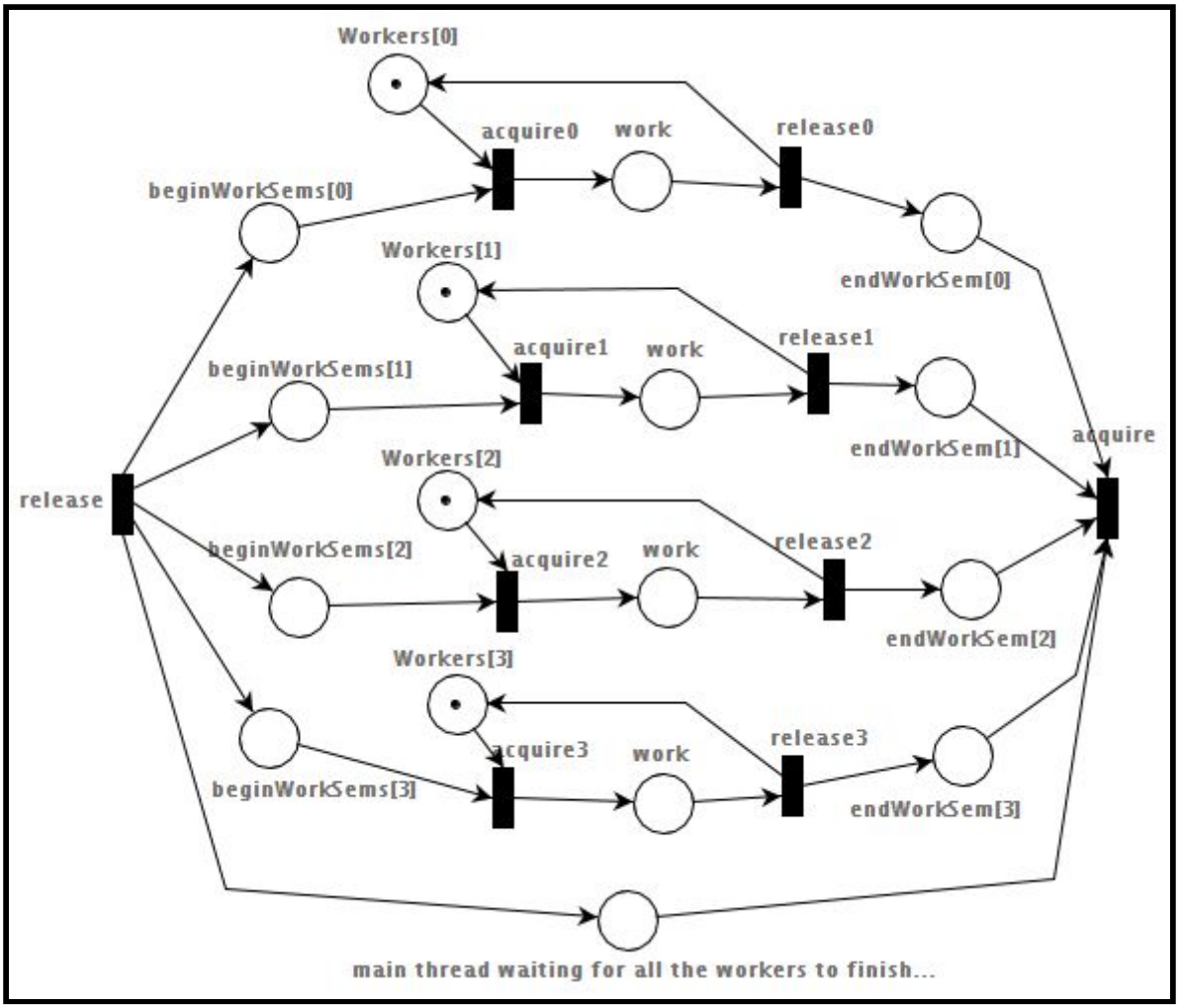
Come compromesso sia per mantenere questo tipo sincronizzazione, sia **massimizzare il throughput** della sequenza di fotogrammi, si è deciso di calcolare il mondo N+1 durante l'attesa della avvenuta visualizzazione del mondo N; nel caso però il calcolo del mondo N+1 terminasse prima della visualizzazione del mondo N, allora si dovrà attendere una notifica di avvenuta visualizzazione del mondo N prima di aggiornare il model ed iniziare quindi il calcolo del mondo N+2.

Un'alternativa che si sarebbe potuta considerare sarebbe stata quella di continuare a computare i mondi successivi, tenendo però in considerazione l'ultimo di essi visualizzato: si sarebbe così realizzato una sorta di *buffering* controllato. Si è però deciso di non implementare questa soluzione per un paio di ragioni:

1. Si sarebbe aggiunta una certa, non necessaria, complessità al problema: si è preferito concentrarsi su altri aspetti di questa esercitazione.

2. Nel caso si riuscissero ad accumulare K nuovi stati del mondo sul buffer, ci si dovrebbe anche poi preoccupare della frequenza a cui farli visualizzare, per non rendere la visualizzazione delle immagini in serie particolarmente veloce o lenta a seconda se l'immagine che si sta visualizzando faccia parte del buffer (e fosse quindi istantaneamente già pronta) oppure fosse un'immagine rappresentativa di un mondo che si è appena calcolato (impiegando una certa quantità di tempo non trascurabile).

Rete di petri secondaria: Workers

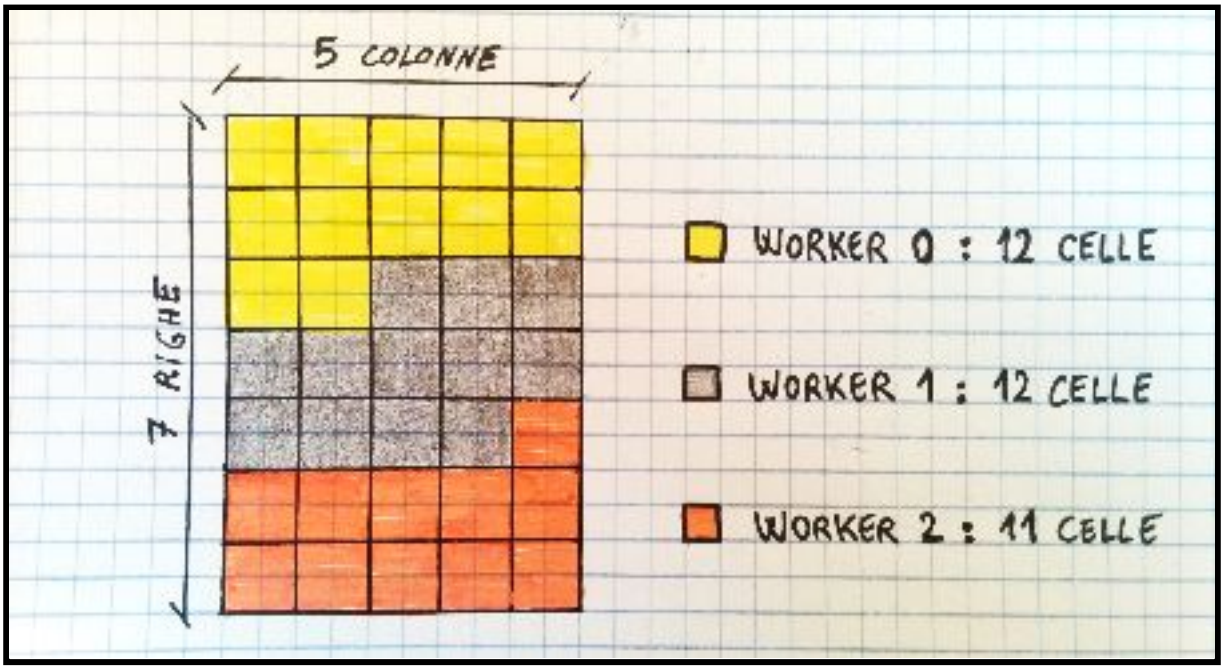


La rete qui rappresentata corrisponde ad uno *zoom* più dettagliato sui cerchietti azzurri e rosa della rete di Petri principale riportata a pagina 2. In essa è possibile vedere come il tutto sia semplicemente gestito da una coppia di *array* di semafori condivisi dai *workers* e dal *Thread* principale del programma.

Tale sistema di coordinazione è stato realizzato allo stesso modo sia per i worker adibiti al calcolo del mondo successivo (*GameWorkers*, nel cerchietto rosa), sia per i worker adibiti alla codifica dell'immagine *bitmap* (*ControllerWorkers*, nel cerchietto azzurro): il Thread principale rilascia tutti i semafori *beginWorkSems* ed attende che tutti i *worker* finiscano il loro lavoro facendo acquire sui semafori *endWorkSem*. NB: nelle *acquire* finali viene imposta inevitabilmente **sequenzialità** nell'ordine di acquisizione (dal semaforo 0 a quello N); tuttavia, in questo caso si ritiene tollerabile poiché è comunque necessario attendere che tutti i *worker* abbiano terminato, e di conseguenza risulta trascurabile anche il caso peggiore in cui si facessero *acquire* in ordine inverso rispetto all'ordine di terminazione.

Suddivisione del lavoro tra i Workers

Nonostante possa sembrare quasi maniacale, si è deciso di suddividere il lavoro tra i vari *worker* con granularità a livello di singola cella. In aggiunta, nel caso il numero di celle non fosse divisibile per il numero di *worker*, il resto verrà ri-distribuito tra loro, in modo da ottenere una suddivisione il più possibile omogenea. Si riporta di seguito un semplicissimo esempio illustrativo dove vengono assegnate le celle di un mondo di gioco di 7 righe per 7 colonne a tre lavoratori.



Essendo la suddivisione effettuata solo in fase di inizializzazione di gioco, si è ritenuto opportuno farla il più accurata possibile, anche perché potrebbe in un futuro fare la differenza nel caso di matrici relativamente piccole elaborate da un numero molto elevato di *workers* (si pensi ad esempio se il calcolo di queste matrici fosse demandato ad una GPU!).

Dettagli e scelte implementative

Implementazione dei worker

Essendo le due tipologie di worker utilizzate all'interno del nostro programma molto simili, si è ritenuto naturale costruire una classe astratta Worker, che definisse l'implementazione delle parti comuni (e quindi anche la **gestione della sincronizzazione** con il *thread* principale), e dichiarasse un metodo astratto (***work***) indicativo del lavoro specifico che differenzia un *Worker* specializzato dall'altro (*GameWorker vs ControllerWorker*).

Si vuole infine puntualizzare un piccolo dettaglio sui *GameWorker:* si è ritenuto utile per motivi di ***performance*** far contare ad ogni lavoratore la popolazione futura del suo settore di mondo **durante la computazione** del mondo successivo. Il conteggio si sarebbe potuto fare successivamente, sporcando di meno il codice, ma in questo modo si sono ottenute ottime prestazioni e si è migliorato il ***throughput***. Altro esperimento che si è fatto è stato quello di aggiornare il valore della popolazione del mondo futuro tramite metodi pubblici *synchronized* (incapsulando quindi la popolazione futura una sorta di *monitor*), ma il prevedibile rallentamento derivato è stato davvero eccessivo per giustificare una soluzione del genere, solo leggermente più elegante.

Varie ed eventuali

Si è cercato di sviluppare l'applicazione seguendo quanto più possibile i pattern architetturali del ***Model View Controller*** e dell'***Observer***.

Seguire il primo dei due è risultato particolarmente facile grazie all'impostazione di *default* dei progetti sviluppati con ***Java FX***, i quali agevolano la corretta suddivisione dei compiti tra la parte puramente grafica e quella puramente logica dell'applicazione. Questo ha permesso di realizzare molto rapidamente (in seguito) anche una versione del programma senza GUI, utilizzata per testare le *performance* anche in assenza di interfaccia grafica (si vedano i risultati nel capitolo prossimo capitolo: "Prove di performance").

Per quanto riguarda invece il secondo, si è scoperto durante lo sviluppo che la classe ***Observable*** e l'interfaccia ***Observer*** risultano segnalate come **obsolete da Java 9**: si è optato quindi per una soluzione *ad-hoc* (e semplificata) sulla falsariga di tale pattern, introducendo una interfaccia ***Visualizer*** che rappresenterebbe l'Observer della classe Game.

Nel caso lo scenario applicativo fosse risultato più complesso (o ci fosse stata la necessità di notificare più interfacce grafiche contemporaneamente) sarebbe valsa la pena approfondire *PropertyChangeEvent* e *PropertyChangeEvent*: alternative ai vecchi *Observer* / *Observable* proposte da *java.beans*.

Come ultimo dettaglio implementativo si vuole sottolineare di come le specifiche di Java FX affermino che un qualunque *thread* possa **preparare una *Canvas* prima di legarla alla sua *View***, e di come questa sia da alcuni consigliato farla preparare da un *Thread* diverso dall'*FX Application Thread* per aumentare la reattività della GUI (che risulterà meno carica di lavoro).

Per questo motivo, ad ogni iterazione del mondo, nell'applicazione presentata viene creata una nuova *Canvas* dal *Game Thread*, il quale la prepara per il *thread* della GUI, il quale dovrà solamente preoccuparsi di applicarla sul pannello.

Prove di performance

Avendone la possibilità, si è voluto vedere non solo quanto migliorassero le *performance* nel calcolo del nuovo mondo a partire da quello precedente, ma si è voluto verificare per curiosità quanto le prestazioni cambiassero in una CPU **AMD con 4 core fisici** rispetto ad una CPU **Intel con 2 core fisici e 2 logici in HyperThreading**.

I risultati ottenuti sono stati rilevati tramite una media su tre diverse prove indipendenti effettuate su un mondo di gioco di dimensione 5000x5000; per ogni prova si è a sua volta effettuata una media su circa 100 iterazioni di gioco.

Avendo cercato di sfruttare la parallelizzazione anche in fase di codifica dell'immagine, si è ritenuto opportuno ed interessante vedere quanto fosse efficace la parallelizzazione sia utilizzando una versione del programma semplificata (senza GUI) che una versione completa.

Come prevedibile, il *throughput* in generale, così come lo *speedup* e l'*efficienza* sono risultati migliori in assenza di interfaccia grafica, ma non in modo drammatico, anzi, ci si ritiene notevolmente soddisfatti della velocità dell'evoluzione del mondo di gioco in presenza della GUI, oltre che della sua reattività, testata manualmente.

Prestazioni senza GUI

# Processori	Tempo (ms)	Speedup	Efficienza
1	811	-	-
2	472	1.72	0.860
3	298	2.72	0.907
4	238	3.41	0.852

*Risultati ottenuti con 4 core fisici: **AMD Phenom X4 965 Black Edition** @ 3.40GHz (marzo 2009).*

# Processori	Tempo (ms)	Speedup	Efficienza
1	744	-	-
2	374	1.99	0.995
2 + 1	338	2.20	0.733
2 + 2	287	2.59	0.648

*Risultati ottenuti con 2 core fisici e 2 logici: **Intel Core i5-2410M** @ 2.30GHz (febbraio 2011).*

Prestazioni con GUI

# Processori	Tempo (ms)	Speedup	Efficienza
1	1360	-	-
2	767	1.77	0.887
3	595	2.29	0.762
4	540	2.52	0.630

*Risultati ottenuti con 4 core fisici: **AMD Phenom X4 965 Black Edition** @ 3.40GHz (marzo 2009).*

# Processori	Tempo (ms)	Speedup	Efficienza
1	1073	-	-
2	641	1.67	0.835
2 + 1	628	1.71	0.570
2 + 2	573	1.87	0.468

*Risultati ottenuti con 2 core fisici e 2 logici: **Intel Core i5-2410M** @ 2.30GHz (febbraio 2011).*

Alla luce dei risultati ottenuti con il processore *Intel*, si ritiene perlomeno interessante segnalare come la casa produttrice californiana abbia quest'anno deciso (in controtendenza rispetto al recente passato) di aumentare il numero di *core* fisici nella sua gamma di *cpu desktop* (Coffee Lake), eliminando di controparte i core logici in tutti i modelli, eccezion fatta per i top di gamma (i7 serie 8700).

Principali problematiche riscontrate

Inizialmente è risultato complesso capire come visualizzare in tempi ragionevoli matrici di grandi dimensioni (dell'ordine di 5000x5000). Non essendo abituati a lavorare con immagini *bitmap*, si disegnava ogni singolo pixel/cella del mondo di gioco tramite una chiamata a funzione *DrawRect*: di conseguenza, per passare da una configurazione di gioco a quella successiva, trascorrevano più di una decina di secondi impiegati quasi esclusivamente per "disegnare" l'immagine. Dopo aver appreso come codificare opportunamente lo stato di gioco in un array di *bytes* e, solo successivamente, disegnarla con un'unica istruzione (***setPixels***) sulla *Canvas* di gioco, la velocità nella transizione da un'immagine all'altra è migliorata drasticamente.

L'altro problema principale, sempre legato alla visualizzazione del mondo di gioco, è stato incontrato facendo prove con mondi di gioco dell'ordine di 10000x10000 pixel. Si è poi scoperto che esiste una limitazione sulla dimensione massima di una *Canvas* di *JavaFX*, dipendente dalla scheda grafica. In particolare, si è verificato (anche sperimentalmente) che nelle macchine possedute il limite fosse di **8192x8192** *pixel*.

[https://stackoverflow.com/questions/17563827/maximum-dimensions-of-canvas-in-javafx]

Possibili soluzioni alternative

Barriera ciclica

I meccanismi di coordinazione/sincronizzazione tra i vari flussi di controllo è stata realizzata tramite ***semafori***: scelta personalmente sentita come naturale già dalle prime fasi dell'analisi del problema e dello sviluppo. Ragionando a posteriori, con conoscenze aggiuntive ed una miglior comprensione del sistema, si ipotizza che si sarebbe potuta anche implementare una soluzione che sfruttasse una ***barriera ciclica***: questo tipo di *monitor* (visto nel dettaglio due giorni fa, durante lo scorso laboratorio) sembrerebbe un'appropriata ed elegante alternativa al doppio *array* di semafori (*beginWorkSems* ed *endWorkSems*) utilizzati per la coordinazione dei *worker*.

Algoritmo alternativo per il calcolo del mondo successivo

In una seconda possibile soluzione (ritenuta però molto meno interessante della prima ai fini del corso) si sarebbe potuto verificare come sarebbero cambiate le *performance* tenendo traccia non solo del mondo attuale, ma anche di quali celle avessero cambiato il loro stato durante l'ultimo turno: è infatti chiaro che nessuna cella cambierà di stato nel turno immediatamente successivo a quello corrente se in quello precedente né essa stessa né i suoi vicini avevano alterato il loro stato. Si può ipotizzare che l'*overhead* introdotto da una soluzione simile sarebbe potuto esser compensato nel caso il mondo di gioco si fosse stabilizzato, o comunque nel caso la popolazione fosse risultata particolarmente sparsa. Infine, si sarebbe potuta addirittura realizzare una soluzione ibrida, in grado di alternare autonomamente tra le due modalità di calcolo (quella *standard* implementata da noi, e questa alternativa qui descritta), a seconda della popolazione e/o del numero di celle che avessero cambiato il loro stato nei turni precedenti.

Configurazioni di gioco

All'interno del *package* ***gameoflife.config*** sono presenti due file di configurazione: ***GameConfig.java*** e *ViewConfig.java*. Il primo è risultato di particolare comodità sia durante lo sviluppo che durante lo svolgimento di test e prove di *performance*. Se ne riporta di seguito un estratto contenente le impostazioni di gioco ritenute più importanti.

```
final public static int WORKER_THREADS = Runtime.getRuntime().availableProcessors();

final public static InitType INIT_TYPE = RANDOM;  // RANDOM, GLIDER, COLUMN, CROSS;
final public static double RANDOM_ALIVE_FACTOR = 0.15;

final public static int ROWS = 5000;
final public static int COLUMNS = 5000;
final public static boolean BORDERS = false;

final public static int PERFORMANCE_UPDATE_FREQUENCY = 100;
```

Nella prima riga è possibile scegliere con **quanti *Worker*** lavorare, tale valore influenzerà allo stesso modo sia il numero di GameWorkers (il cui scopo è quello di calcolare lo stato futuro del mondo), sia il numero di ControllerWorkers (il cui scopo è quello di codificare l'immagine prima di farla applicare sulla GUI dall'*FX Application Thread*).

Con la seconda e la terza costante hanno è possibile scegliere la **configurazione iniziale** del mondo di gioco:

- Configurazione *RANDOM*, accompagnata da un *RANDOM_ALIVE_FACTOR* che indica la probabilità che ha ogni cella di essere inizialmente viva o morta. *RANDOM* è la scelta di default, e quella utilizzata anche nelle prove di performance.
- GLIDER: risultato sorprendentemente utile per testare mondi di piccole dimensioni (con e senza bordi) grazie al quale è stato possibile verificare rapidamente la presenza di problematiche gravi di sincronizzazione tra *workers*.
- COLUMN e CROSS: utilizzate molto raramente, sono state però utili per verificare nelle fasi iniziali di sviluppo per verificare che queste figure note si comportassero come previsto.

Il terzo gruppo di costanti riguarda la **composizione del mondo di gioco**, esse si ritengono sufficientemente auto-esplicative, ci si limita perciò solamente a precisare che, in assenza di bordi (*WITH_BORDERS = false*) la parte inferiore risulta comunicante con quella superiore, così come quella di sinistra risulta comunicante con quella di destra.

PERFORMANCE_UPDATE_FREQUENCY indica, infine, ogni quante evoluzioni del mondo stampare su *console* una media del tempo impiegato per il calcolo di ciascuna evoluzione. Questo parametro è ovviamente rilevante solamente nel caso si sia settato un cronometro, (oggetto di classe *utils.Chrono*), tra i campi opzionali dell'istanza di *Game*.