# UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI INFORMATICA



Corso di Laurea in Informatica

# Life-like cellular automaton engine


Progetto di:
**Pasquale Tudda**
**Mat. 219754**

# Indice

# Capitolo 1

# Introduzione

## 1.1 Cos'è un automa cellulare?

Un automa cellulare è un modello matematico che consiste, nel caso più generale, in una griglia costituita da **celle**.

Ad ogni generazione ogni cella ha un comportamento che dipende dalla cosiddetta **regola** dell'automa cellulare.

In poche parole ogni cella guarda il suo vicinato, e in base a questo criterio e allo stato della cella attuale l'algoritmo stabilisce lo stato della cella nella prossima generazione, e così via per un numero finito di generazione.

Conoscendo la disposizione iniziale delle celle, il numero di generazioni e la regola dell'automa è sempre possibile calcolare qualsiasi generazione successiva, in quanto gli automi cellulari

## 1.2 Il gioco della vita

L'automa cellulare più famoso è sicuramente il cosiddetto gioco della vita ("*Game of life*" in inglese), sviluppato dal matematico britannico John Conway verso la fine degli anni sessanta.

Ogni cella può assumere esclusivamente due valori **viva** o **morta**.

### 1.2.1 Quali sono le regole?

Le regole del gioco della vita sono molto semplici:

- Qualsiasi cella viva con meno di due celle vive adiacenti muore, come per effetto d'isolamento;
- Qualsiasi cella viva con due o tre celle vive adiacenti sopravvive alla generazione successiva;
- Qualsiasi cella viva con più di tre celle vive adiacenti muore, come per effetto di sovrappopolazione;
- Qualsiasi cella morta con esattamente tre celle vive adiacenti diventa una cella viva, come per effetto di riproduzione.

Figura 1.1: "Glider gun" preso dal celebre automa Game Of Life

### 1.2.2 Il vicinato

Come accennato prima ogni cella ad ogni generazione guarda lo stato delle celle vicine.

Nel caso del gioco della vita ogni cella guarda esattamente ogni cella attorno a se che non dista più di una cella, come mostrato in rosso in figura (1.2).

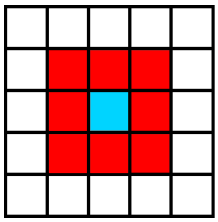


Figura 1.2: Vicinato di Moore in rosso

## 1.3 Automi cellulari simili al gioco della vita

Un automa cellulare per essere considerato simile al gioco della vita deve rispettare tre condizioni:

- deve essere modellato su una **griglia di celle**;
- ogni cella deve avere due **stati** (viva o morta);
- la vista del **vicinato** deve essere quella di Moore (figura 1.2);

## 1.4 Cos'è una string rule?

Una string rule è un modo molto comodo per sintetizzare le regole di un automa cellulare in una singola riga.

Per la realizzazione del mio progetto ho stabilito che il formato della *string rule* utilizzata fosse del tipo:

$$B[0\text{-}8]/S[0\text{-}8]$$

Dove **B** sta per **Born** (*nato* in inglese) ed **S** sta per **Survivor** (*sopravvissuto* in inglese). Le cifre che seguono la lettera B si riferiscono al numero esatto di vicini che una cella "morta" deve avere per nascere, mentre le cifre che seguono la S indicano il numero esatto di vicini che una cella "viva" deve avere per sopravvivere alla prossima generazione.

Seguendo questo pattern, ogni volta che una cella non rispetta una delle due condizioni sopra riportate, può essere considerata morta.

Ad esempio la string rule del gioco della vita di Conway, appurate le regole descritte nella sezione 1.2.1 di questo capitolo si riassumono in:

$$B23/S3$$

# Capitolo 2

# Il progetto

## 2.1 In cosa consiste?

Il programma consiste in un **simulatore di automi cellulari** simili al gioco della vita, quindi, specificando la string rule che si vuole nel file *settings.json* è possibile generare ed osservare qualsiasi automa cellulare ammesso che si conoscano le sue regole.

Il programma utilizza la libreria grafica Allegro5 ma è possibile anche specificare di non voler usare la componente grafica attraverso una semplice define all'interno del file *main.cpp*.

Inoltre, cosa fondamentale nella realizzazione del progetto, il programma può essere eseguito sia in **sequenziale** su un solo processore, sia in **parallelo** su più processori.

All'interno del file di configurazione *settings.json* è possibile specificare:

- se inizializzare la griglia in modo randomico (se lasciato a 0 stamperà sempre lo stesso pattern, future usata in fase di testing)
- il numero di generazioni
- la dimensione della griglia;
- la dimensione del display
- il colore delle celle vive
- il colore delle celle morte
- la possibilità di mostrare a schermo il numero di generazioni
- il colore del testo mostrato a schermo
- quanto tempo far aspettare il programma tra una generazione e l'altra in millisecondi

Inoltre è stata implementata la possibilità di raccogliere i dati in degli appositi file di log in modo da poter calcolare una serie di benchmark.

## 2.2 Strumenti utilizzati

Per la realizzazione del progetto ho usato diversi linguaggi e tecniche di programmazione, in ordine di importanza:

- **C++**: per quanto riguarda la creazione dell'automa stesso con l'utilizzo del protocollo **MPI** per la parallelizzazione e la comunicazione tra processi e la libreria **Allegro5** per la gestione della grafica;
- **json**: per il file di configurazione;
- **makefile**: per compilare velocemente il codice sorgente dati la grande quantità di argomenti tra cui il linking della libreria Allegro5
- **bash**: per eseguire facilmente gli eseguibili in MPI dati i diversi parametri che richiede il lancio del programma;
- **python**: per la semplicità con la quale si possono raccogliere i dati relativi al tempo di compilazione e la creazione dei grafici;

## 2.3 Versione sequenziale

La versione sequenziale del programma è relativamente semplice, infatti ad ogni iterazione l'algoritmo legge lo stato di ogni cella della matrice della generazione corrente, ne guarda il vicinato, conta il numero di vicini e in basse alla string rule stabilisce lo stato della cella andando a modificare la matrice della prossima generazione. Terminata la lettura e le scrittura degli stati viene stampata con Allegro5 la generazione corrente (se la feature è abilitata), le due matrici vengono scambiate e l'algoritmo ricomincia.

In questo modo si calcolano tutte le generazioni della formazione iniziale di celle finché non si arriva al massimo di generazioni indicato.

E' stato necessario "linearizzare" le due matrici, quindi inserendo il contenuto delle celle lungo un solo array, in memoria contigua.

## 2.4 Versione parallela

Per poter parallelizzare l'esecuzione del programma ho ritenuto necessario fare alcune scelte fondamentali:

- Creare una topologia virtuale, con **MPI_Cart_create**, bidimensionale di una colonna e tante righe quanti sono i processi, con la possibilità che il primo processo e l'ultimo comunichino in modo toroidale;

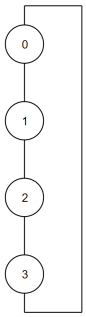



Figura 2.1: Ecco come sarebbe la topologia se il programma venisse lanciato con 4 processi.

- è stata inizializzata per ogni processo una matrice più piccola di dimensione $\frac{\text{dimensione totale}}{\text{numero di processi}} + 2$, dove due sono le righe fantasma utili allo scambio dei bordi con i processi vicini. L'operazione di scambio bordi avviene usando un invio di dati non bloccante usando **MPI_Isend** in direzione *up*, quindi al processo sopra quello corrente viene inviata la prima riga della sottomatrice che viene ricevuta nella riga fantasma del processo sovrastante, e in direzione *down* viene inviata l'ultima riga della sottomatrice del processo corrente che viene ricevuta nella riga fantasma del processo sottostante. Entrambe le ricezioni avvengono in modo bloccante, con **MPI_Recv** per evitare che le funzioni di transizione leggano dati non ancora arrivati a destinazione;

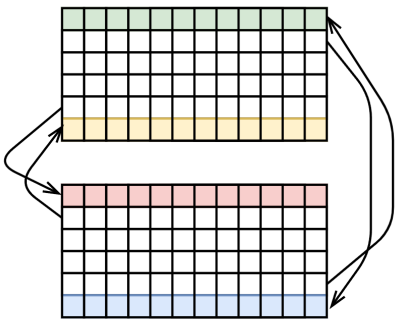


Figura 2.2: Sottomatrice con celle fantasma

- dopo lo scambio dei bordi su ogni sottomatrice viene calcolata, come per la versione sequenziale, la generazione successiva attraverso una funzione di transizione;
- tutti i processi inoltre condividono una matrice di dimensione $n \times n$ che viene riempita nel processo 0 alla fine del calcolo di ogni generazione, prima della stampa, attraverso

un'operazione di raccolta con **MPI_Gather**, in cui tutti i processi inviano la loro sottomatrice senza celle fantasma alla matrice principale;

- lo scambio di righe avviene utilizzando un tipo customizzato che **MPI** mette a disposizione, ovvero **MPI_Type_contiguous** il quale contiene elementi contigui in memoria. Questa scelta permette uno scambio molto più rapido delle righe specificando nelle operazioni di send e recive quante righe si vuole inviare o ricevere. Questa matrice è utile al processo 0, ovvero quello principale, a stampare in modo seriale la matrice così da avere una visualizzazione sincrona sullo schermo;

- infine, come per la versione sequenziale, al termine di ogni iterazione le sottomatrici di ogni processo della generazione corrente vengono scambiate con la sottomatrice della generazione successiva;

# Capitolo 3

# Benchmarks

Sul programma sono stati eseguiti una serie di test per verificare che la parallelizzazione dell'automa cellulare abbia dato i risultati sperati in termini di prestazioni, qui di seguito sono riportati tutti i dati raccolti e i relativi test.

## 3.1 I dati

Per raccogliere i dati ho scritto uno script in Python che non fa altro che eseguire il programma prima con un solo processo, in sequenziale, arrivando ad usare fino a 6 processi contemporaneamente. Questo in virtù del fatto che la macchina su cui sono stati eseguiti i test ha al massimo 6 processori.

I seguenti dati sono stati raccolti lanciando il programma su una matrice di dimensione $300 \times 300$.

| Tempi di esecuzione su 500 iterazioni | |
|---|---|
| Processori | tempo medio (sec) |
| 1 | 6,178 |
| 2 | 5,357 |
| 3 | 3,887 |
| 4 | 2,934 |
| 5 | 2,551 |
| 6 | 2,098 |

I seguenti dati sono stati raccolti lanciando il programma su una matrice di dimensione $600 \times 600$.

| Tempi di esecuzione su 500 iterazioni | |
|---|---|
| Processori | tempo medio (sec) |
| 1 | 23,202 |
| 2 | 22,927 |
| 3 | 14,833 |
| 4 | 8,345 |
| 5 | 6,723 |
| 6 | 5,736 |

## 3.2 Speed-up

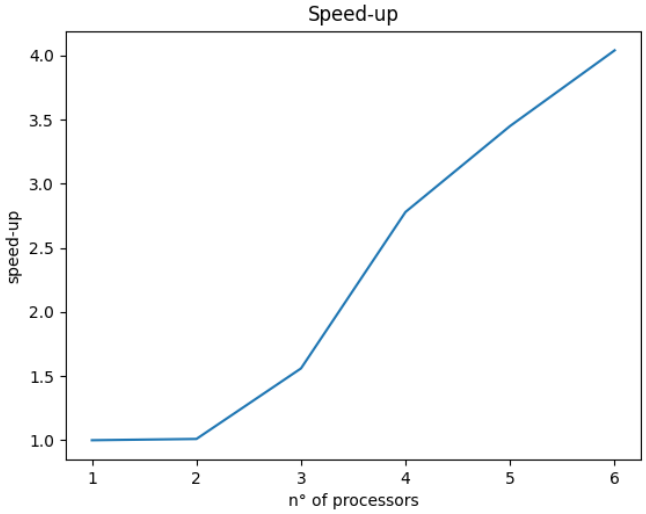
Lo **speed-up** indica il **beneficio** che un programma trae dalla sua parallelizzazione e viene calcolato in questo modo:

$$S(n) = \frac{t_s}{t_p}$$

Dove $t_s$ è il tempo di esecuzione sequenziale, mentre $t_p$ è il tempo di esecuzione parallelo su $n$ processori.

Usando i dati raccolti in precedenza:

| Speed-up | | |
|---|---|---|
| Processori | su matrice 300x300 | su matrice 600x600 |
| 1 | - | - |
| 2 | 1,15 | 1,01 |
| 3 | 1,59 | 1,56 |
| 4 | 2,11 | 2,78 |
| 5 | 2,42 | 3,45 |
| 6 | 2,94 | 4,04 |




Come si può notare il programma ha una buona scalabilità all'aumentare della dimensione del problema, arrivando a toccare circa uno speed-up di ben 4.

## 3.3 Overhead

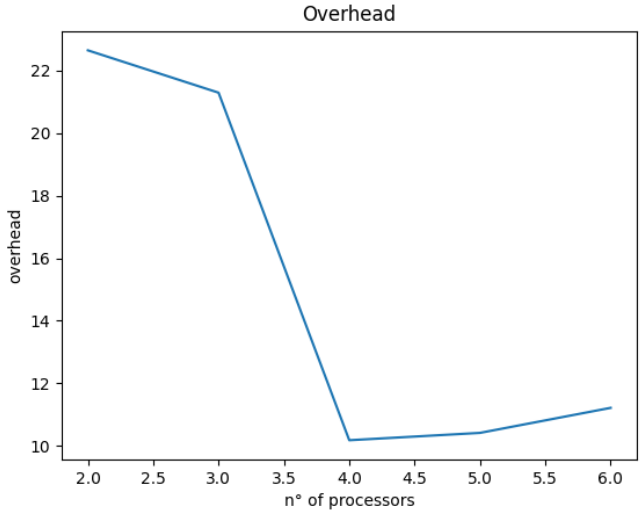
L'overhead indica il tempo extra che il programma genera a causa delle operazioni che avvengono durante la sua esecuzione, ad esempio lo scambio di messaggi all'interno di MPI, l'uso di Wait e Barrier, potrebbero generarlo.

Esso si calcola in questo modo:

$$T_o = pt_p - t_s$$

Usando solo i dati relativi all'esecuzione sulla matrice 600x600

| Processori | Overhead |
|---|---|
| 1 | - |
| 2 | 22,652 |
| 3 | 21,297 |
| 4 | 10,178 |
| 5 | 10,413 |
| 6 | 11,214 |




## 3.4 Efficienza

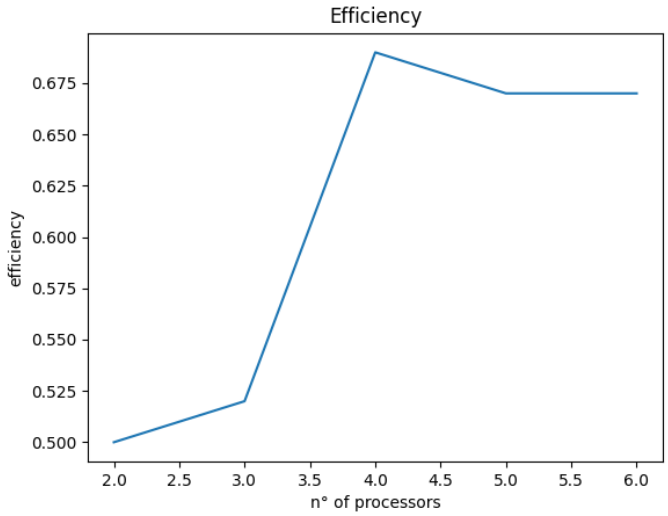
L'**efficienza** è un valore che indica la percentuale di utilizzo di una unità di processamento parallelo, dove 0 sta per inutilizzato e 1 per utilizzato al 100%, ed è calcolato in questo modo:

$$E = \frac{S}{p} \qquad \text{con } E \in [0, 1]$$

Dove $S$ è lo speed-up e $p$ il numero di processori.

Questa volta l'efficienza verrà calcolata esclusivamente sui dati relativi alla matrice $600 \times 600$.

| Processori | Efficienza |
|---|---|
| 1 | - |
| 2 | 0,50 |
| 3 | 0,52 |
| 4 | 0,69 |
| 5 | 0,67 |
| 6 | 0,67 |




Come si può vedere l'efficienza di stabilizza attorno a 0,67, questo indica che ogni processore lavora al massimo al 67%