
Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

# Cellular Automata Framework

Antonio Zegarelli

Matricola: 561281

SPM Course

Anno Accademico 2020/2021

# Indice

# 1 Introduzione

L'obiettivo di questo progetto è la progettazione, implementazione e analisi di un framework per il task Cellular Automata. L'implementazione del modello dovrà essere realizzata sia in fastflow che con l'utilizzo diretto dei thread c++. Nei requisiti è richiesto il supporto di una griglia toroidale, in cui l'ultima riga/colonna è connessa con la prima riga/colonna. La regola per effettuare l'aggiornamento dello stato delle celle si assume sia la stessa per tutte e non cambi col tempo, dovrà inoltre essere applica simultaneamente sull'intera griglia. Viene infine aggiunta la possibilità di scrivere su disco una rappresentazione tramite immagine di ogni iterazione, inoltre l'assunzione fatta per questo progetto è che il tempo di computazione della regola sia bilanciato su tutte le celle.

# 2 Design

Il problema viene riconosciuto come uno stencil pattern dato che sono presenti delle dipendenze strutturali, infatti, quando bisogna calcolare lo stato di una cella è necessario leggere lo stato dei suoi "vicini", il quale a sua volta potrebbe essere sovrascritto. Assumendo che la regola sia bilanciata su tutte le celle, possiamo semplificare la suddivisione del task creando dei range nella matrice, che verranno poi assegnati ai workers.

## 2.1 Farm

Il primo modello scelto è la farm, l'emitter si occupa di assegnare il range ad ogni worker, dato che non richiede molte risorse è stato quindi unito con il collector il quale, nelle versioni in cui è abilitata, si occupa della scrittura delle immagini. I worker invece si occupano di effettuare il calcolo del nuovo stato per ogni iterazione usando i due buffer tra una e l'altra. Inoltre si occupano di assegnare la rappresentazione dello stato all'interno dell'immagine quando la scrittura è abilitata. Dopo ogni iterazione è necessario sincronizzare i worker, questo viene realizzato tramite una barriera. Infine nella versione con la scrittura abilitata, dopo la sincronizzazione, un worker invia l'iterazione al collector per scrivere l'immagine sul disco.

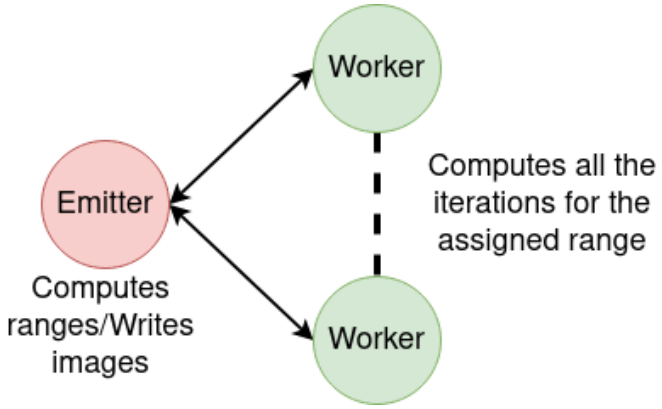


Figura 1: Farm emitter/collector

## 2.2 Parallel For

Realizzando in questo modo la farm si nota che l'implementazione può essere effettuata con un parallel for, parallelizzando anche la scrittura delle immagini per verificare se si ha un aumento nelle performance, dato che potrebbe richiedere molto tempo relativamente al calcolo dell'iterazione quando si utilizzano regole semplici. Ogni worker adesso si occupa della computazione all'interno del range e della scrittura di immagini assegnando un numero di iterazioni bilanciato ad ognuno di essi. In questo modo l'implementazione risulterà più pulita e non sarà necessario dedicare un thread a emitter/collector.

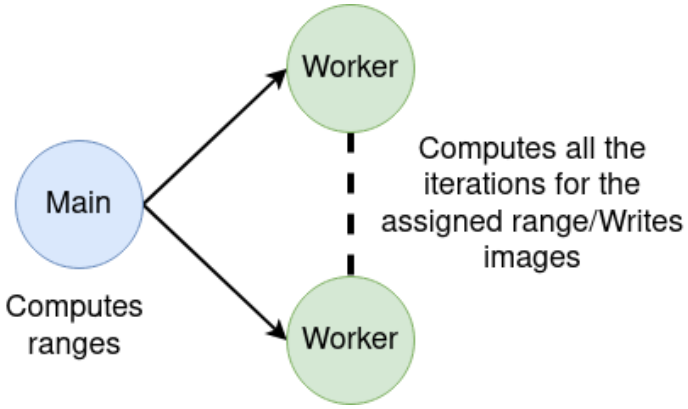


Figura 2: Diagramma modello finale

# 3 Compilazione

Le librerie richieste sono: X11, libpng.

Per compilare le diverse versioni è possibile utilizzare il makefile:

- sequential - sequentialw
- mine - minew
- ff_farm - ff_farmw
- ff_parfor - ff_parforw

Per creare un altro comando per la compilazione, invece, i flag richiesti sono: -pthread -lX11 -lpng e se non presente all'interno del sistema anche -Ipath_to_fastflow. Per abilitare la scrittura delle immagini bisogna effettuare la compilazione utilizzato il flag -DWIMG, il salvataggio verrà effettuato all'interno della cartella frames.

# 4 Utilizzo

Per avere un'implementazione "framework like" è stata creata la classe CellularAutomata<T,C>, la quale fornisce lo skeleton e i metodi viruali su cui effettuare l'override. I tipi richiesti sono:

- T: tipo degli stati
- C: tipo utilizzato per la rappresentazione tramite CImg (deve essere un tipo supportato da CImg)

i metodi principali su cui effettuare l'override sono:

- rule: regola che si occupa di aggiornare lo stato di una cella
- repr: trasforma lo stato in una rappresentazione CImg
- imgBuilder: tper la creazione degli oggetti CImg delle immagini.

Una volta estesa la classe sarà necessario inizializzare i parametri tramite costruttore della superclasse:

- stato iniziale
- #righe, #colonne (grandezza matrice)
- #iterazioni
- #workers (nella farm il totale dei thread sarà #workers+1)

Per aumentare la velocità di esecuzione, la struttura dati rappresentante la matrice deve essere un vettore, come effetto negativo abbiamo quindi che bisogna effettuare un unroll della matrice per renderla 1-d. Dopo aver creato l'oggetto bisogna chiamare il metodo init() che si occupa dell'inizializzazione delle strutture interne e poi si può chiamare il metodo run() che avvia il calcolo.

# 5 Implementazione

Librerie addizionali utilizzate per l'implementazione:

- CImg: usata per la gestione delle immagini
- utimer: usata per ottenere i tempi

## 5.1 Framework

La classe CellularAutomata è implementata come una classe astratta con metodi pure virtual. Lo skeleton è realizzato all'interno del metodo run() il quale utilizza il metodo rule() per la creazione del nuovo stato e il metodo repr() per l'inserimento della rappresentazione di esso all'interno dell'oggetto CImg dell'iterazione. Il metodo imgBuilder() invece viene chiamato all'interno di init() per inizializzare gli oggetti CImg relativi alle immagini che verranno utilizzati da repr.

## 5.2 Strutture dati

Le strutture dati principali utilizzate sono:

- vector<vector<T» matrices: alternating buffers per la gestione delle iterazioni
- vector<range> ranges: lista dei range assegnati ad ogni worker
- vector<CImg<C» images: lista degli oggetti relativi alle immagini CImg

## 5.3 Barrier

Dato che abbiamo uno stencil pattern, uno degli elementi fondamentali è la barriera che viene usata per sincronizzare i worker, quella utilizzata è la ff::Barrier fornita da fastflow che si affida alla pthread_barrier sui sistemi linux.

### 5.4 ffFarm

Per l'implementazione della farm è stato necessario creare le struct relative a emitter/collector e worker. Il main thread si occupa dell'inizializzazione delle strutture dati, la struct emitter/collector viene creata passando nel costruttore il numero di thread, il numero di iterazioni e la lista delle immagini se necessario. Il numero dei thread viene utilizzato dall'emitter come task per inizializzare i worker, invece il numero di iterazioni e la lista delle immagini vengono utilizzate per la scrittura. La struct dei worker viene inizializzata passando nel costruttore il riferimento all'oggetto cellular automata stesso da cui è possibile invocare i metodi.

### 5.5 ffParallel for

L'implementazione del parallel for, a differenza della farm, è molto più semplice e pulita. Data l'assunzione sulla regola, l'assegnamento dei chunks è statico in questo modo è possibile riutilizzare la logica della farm nella quale i range precalcolati vengono associati ai workers, utilizzando i questo caso stepsize 1 sul range 0-nworker. Nel corpo del parallel for vengono effettuate direttamente tutte le iterazioni evitando ogni volta il riavvio.

### 5.6 Pthread

L'implementazione realizzata con pthread segue il modello del parallel for. Il main thread si occupa di inizializzare le strutture tra le quali troviamo vector<thread> contenente i workers. Dopo aver avviato i worker con i rispettivi ranges il thread main aspetta il termine facendo una join() su di essi.

## 6 Performance

L'analisi delle performance si può dividere in due casi, con scrittura delle immagini e senza. In questo modo si può verificare anche se la parallelizzazione della scrittura porta a dei vantaggi concreti. Dato che il problema è parametrizzato dalla grandezza della matrice viene utilizzata una con grandezza di 500x500, come regola invece è stata scelta quella per la game of life di Conway e 100 come numero di iterazioni. Le immagini che verranno

scritte per i test saranno in 2 colori: bianco e nero per rappresentare i 2 tipi di stato della game of life.

Il tempo teorico di completamento sequenziale viene calcolato con la seguente equazione:

$$T_{completion}(n,m) = (T_{updateState}(n,m) + T_{image}(n,m) + T_{imageSave}(n,m)) * N_{iter} \quad (1)$$

occorrono quindi i tempi parametrizzati sul test:

- $T_{updateState}(500,500) = 33466us$: calcolo dello stato
- $T_{image}(500,500) = 13037us$ : rappresentazione dello nell'immagine
- $T_{imageSave}(500,500) = 48082us$ : scrittura di un'immagine su disco

Si ottiene quindi: $T_{completion}(500,500) = 9458500us$

Invece l'equazione per il tempo di completamento parallelo (scegliendo il modello parfor) è:

$$T_{completion}(n,m,nw) = \frac{(T_{updateState}(n,m) + T_{image}(n,m) + T_{imageSave}(n,m)) * N_{iter}}{nw} \quad (2)$$

Analizzando i risultati per il tempo di completamento in 3 si nota subito che parallelizzare anche la scrittura porta a un vantaggio considerevole.

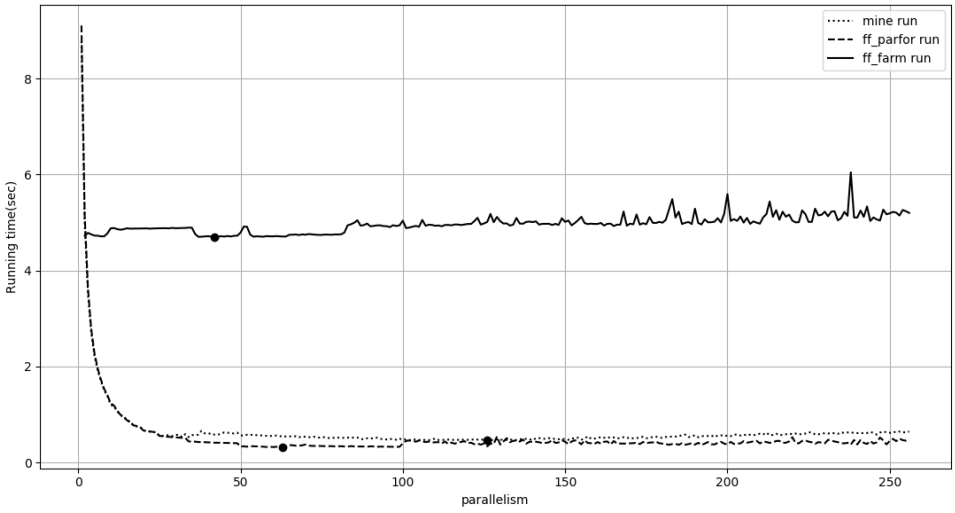


Figura 3: Running time con scrittura abilitata

Questo giustifica perché il modello finale scelto è il parallel for che parallelizza calcolo e scrittura. Osservando adesso lo speedup, nel grafico 4 si nota subito che la differenza

con il teorico è alta. Troviamo inoltre in rosso lo speedup relativo al tempo totale il quale comprende anche la parte relativa all'inizializzazione.

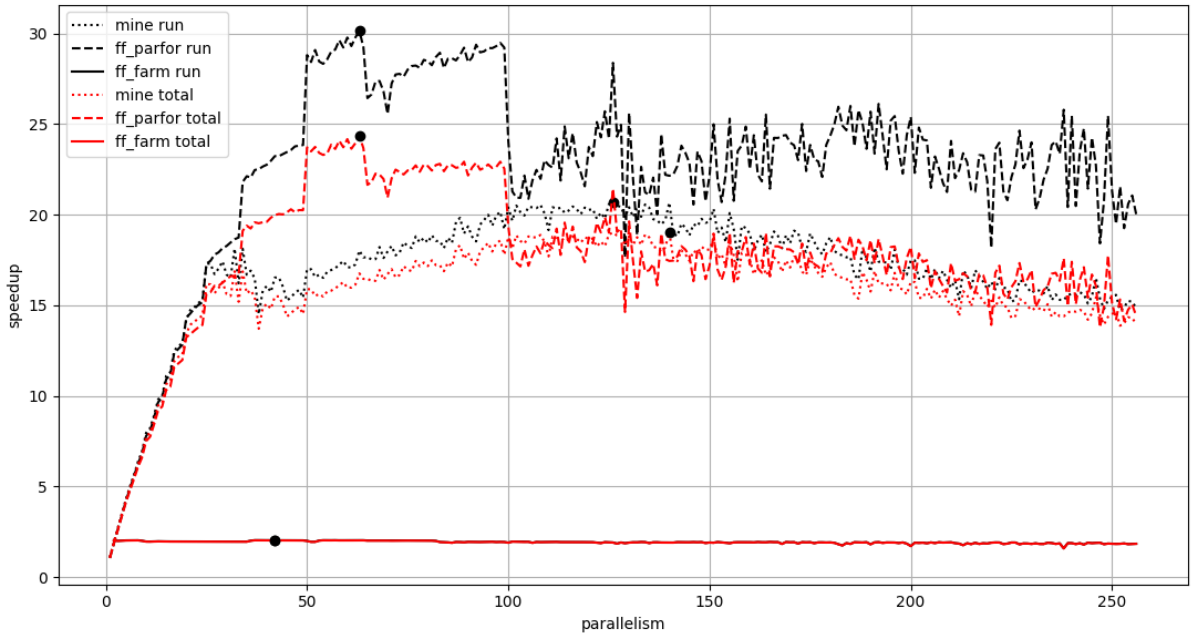


Figura 4: Speedup con scrittura abilitata

Il vantaggio della parallelizzazione sulla scrittura è quindi limitato dal disco, perché come si può vedere comparando i grafici 4 e 5 lo speedup che si riesce a raggiungere con la scrittura attiva è minore di quello che si riesce a raggiungere senza.

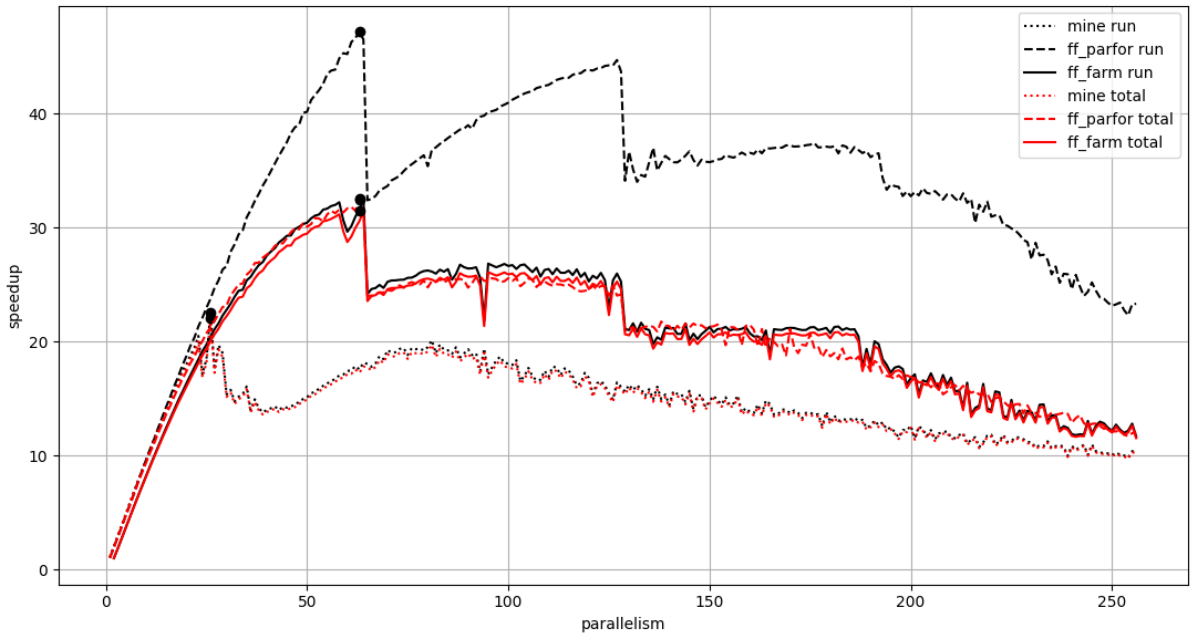


Figura 5: Speedup con scrittura disabilitata

Inoltre si può notare che fastflow inizializza le strutture di parfor e farm in momenti diversi, il parfor in questo modo risulta più veloce se si considera solamente il tempo di run. Considerando invece il tempo totale si vede come i due modelli siano pressoché simili, invece l'implementazione con pthread si trova molto al di sotto mostrando quindi l'efficienza di fastflow e l'utilità del pinning dei thread. Non si riescono comunque a raggiungere le performance teoriche, tra le cause troviamo l'overhead seriale dovuto all'inizializzazione, cache misses e false sharing relativi al tipo di task e implementazione.
Andando ad incrementare la grandezza della matrice a 3000x3000 si può osservare in 6 che lo speedup aumenta, questo andamento infatti segue la Gustafson's law perché in questo caso c'è una crescita della parte non seriale che fa diminuire quindi l'importanza di quella seriale.

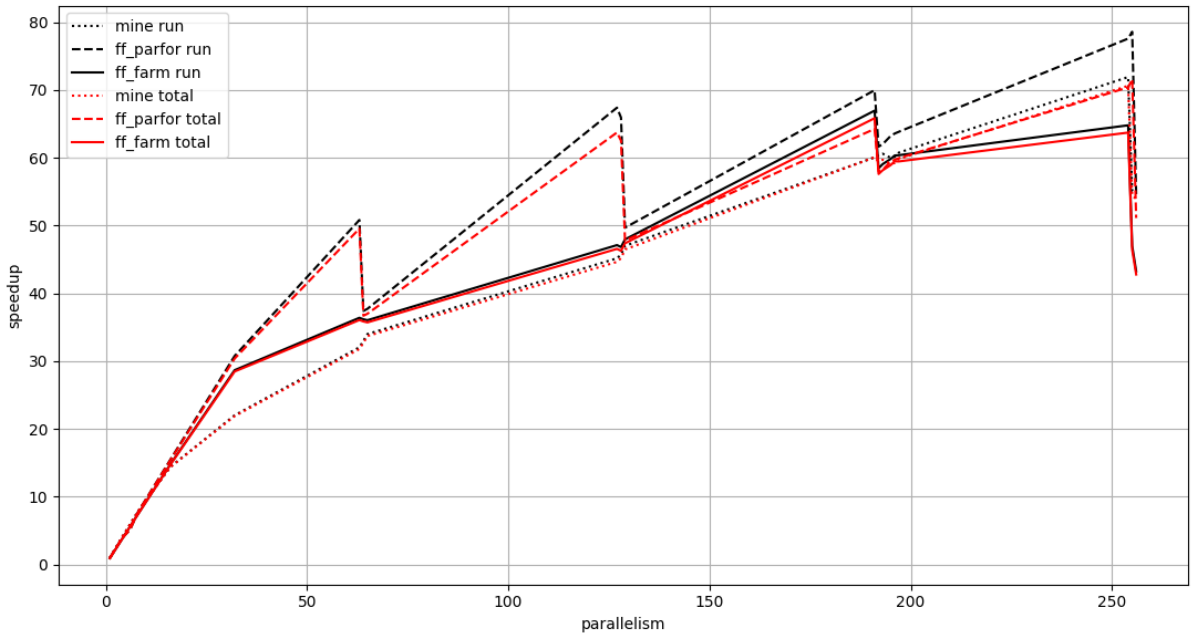


Figura 6: Speedup 3000x3000 con scrittura disabilitata

# 7 Considerazioni finali

Dopo aver analizzato i risultati relativi alle implementazioni risulta chiaro che quelle che utilizzano FastFlow hanno prestazioni nettamente migliori rispetto a quella realizzata solamente con pthread, inoltre anche nella velocità e semplicità di sviluppo del codice si ottengono grandi vantaggi. I risultati ottenuti sono sicuramente ancora migliorabili, andando ad esempio ad utilizzare la tecnica del padding per la riduzione del false sharing. Inoltre si potrebbe considerare la possibilità di avere delle regole non bilanciate, andando ad utilizzare delle politiche di scheduling dinamico.