Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Cellular Automata Framework implementation

Studente:

Simone Rizzo

Professore:

Prof. Marco Danelutto

SPM Course

Anno Accademico 2020/2021

Indice

1 Introduzione

Questo progetto prevede l'implementazione del framework Cellular Automata, realizzando delle programming API. Il framework consiste in una griglia 2D Toroidale in cui ogni elemento possiede uno stato fra un insieme possibili di stati. Lo sato iniziale della matrice viene rappresentato come l'assegnamento dello stato per ogni cella. Ad ogni iterazione vengono aggiornate tutte le celle della matrice mediante una regola definita dall'utente, la quale assegna il nuovo valore in base agli stati del vicinato.

2 Scelte di design

Le specifiche richiedevano l'implementazione di un programma parallelo per effettuare l'aggiornamento degli stati nel Cellular Automata per ogni iterazione. Da una prima analisi del problema è emerso che effettuare l'operazione di aggiornamento di uno stato dato il suo vicinato è un task balanced (ovvero che richiede sempre lo stesso tempo), mentre scrivere le immagini sul disco si è rivelato un task molto più oneroso in termini di tempo e non bilanciato poichè la dimensione dell'immagini cambia in base al colore dei pixel. Per il seguente problema si è scelto di utilizzare il parallel pattern Stencil, questo pattern è molto simile alla Map, ma si differenzia per il fatto che viene utilizzato per dipendenze strutturali dei dati. Nel nostro problema abbiamo esattamente la dipendenza strutturale, nel momento in cui effettuiamo l'aggiornamento di una cella, poichè essa dipende dallo stato nel vicinato che a sua volta può essere modificato da altri worker.

2.1 Primo modello

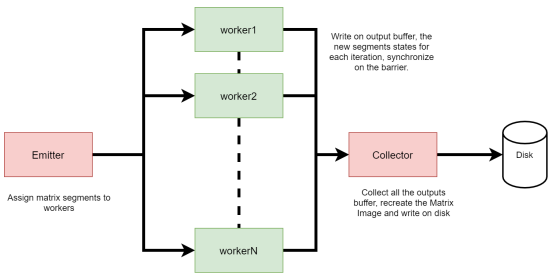


Figura 1: Diagramma primo modello

Il primo modello proggettato prevedeva un Emitter il quale calcolava i segmenti della matrice da assegnare ad ogni worker, ogni worker effettuava il calcolo per ogni iterazione, scriveva su buffer e si sincronizzava per l'iterazione successiva. Il collector aveva il compito di raccogliere tutti i buffer al termine di ogni iterazione e di creare l'immagine da scrivere su disco.

2.2 Secondo modello

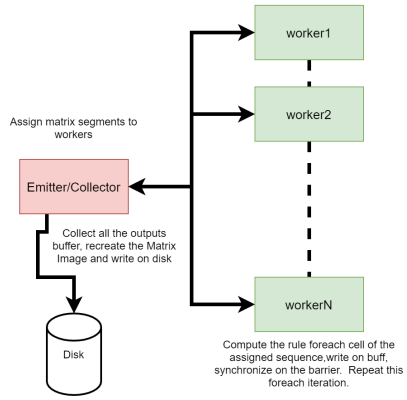


Figura 2: Diagramma secondo modello

Nel secondo modello, a differenza del primo si è scelto di rendere l'emitter anche collector in modo da riutilizzarlo una volta assegnati i segmenti, risparmiando così thread.

2.3 Modello finale

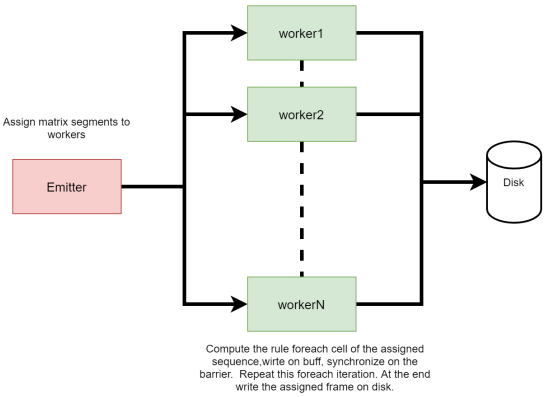


Figura 3: Diagramma modello finale

Il modello finale scelto per l'implementazione è il seguente. A differenza del secondo, quest'ultimo non possiede un collector. Il problema principale degli altri modelli era la scrittura delle immagini su disco poichè veniva svolto in modo sequenziale da un singolo thread. Nei precedenti modelli il tempo di completamento totale era influenzato maggiormente dal collector poichè la scrittura su disco sequenziale richiede un tempo davvero elevato. Per ottimizzare il tutto, ciascun thread al termine del task del Cellular Automata, passa al creare le immagini per ciasciuna iterazione e le scrive direttamente sul disco. In questo modo andiamo a parallelizzare la scrittura riutilizzando i thread già disponibili, diminuendo in modo sostanziale il totale tempo di scrittura rendendolo di circa TscritturaImg/nw.

3 Performance teoriche

Il tempo di completamento nella versione sequenziale teorico del problema e quindi con un thread è data da:

- $T_{createAndWriteImg}(500, 500)$ (Tempo per creare e scrivere un'immagine)= $162, 42ms$
- $T_{updateMatrix}(500, 500)$ (Tempo per calcolare il nuovo stato della matrice 1 worker)= $14, 244ms$

Con matrice 500x500 e $N_{iter} = 150$ abbiamo:

$$T_{completion}(n, m) = T_{updateMatrix}(n, m) * N_{iter} + T_{createAndWriteImg}(n, m) * N_{iter} = circa26, 499sec \tag{1}$$

Utilizzando invece il modello descritto nella sezione precedente 2.3 con nw pari a 200 avremo un tempo di completamento teorico pari a:

$$T_{total}(n, m, nw) = T_{emitter}(n, m, nw) + \frac{T_{updateMatrix}(n, m) * N_{iter}}{nw} + \frac{T_{CreateAndWriteImg}(n, m) * N_{iter}}{nw} \tag{2}$$

$$= 0, 133sec$$

4 Implementazione

In questa sezione vedremo le parti più importanti dell'implementazione del framework in C++. Come libreria esterna è stata utilizzata u.timer per prendere i tempi, cimage per realizzare le immagini da salvare su disco ed infine FastFlow per effettuare dei test sulle performance.

4.1 Strutture dati

A seguire verrano descritte le strutture dati maggiormente utilizzate.

- matrices: è rappresentata mediante un vector<vector<int» utilizziamo 2 vettori di lunghezza $(n * m)$ poichè una viene utilizzata come matrice attuale l'altra come matrice risultante futura. I due vettori si alternano cambiando stato da attuale a futura per ogni iterazione senza dover occupare altro spazio.

- ranges: è un vettore di RANGE, in cui un range è una struct contenente due interi che indicano l'indice di inizio e fine del segmento nella matrice.

- states: è un vector<int> il quale rappresenta i possibili stati di una cella, l'indice rappresenta lo stato, il contenuto il colore del pixel.

4.2 Implementazione della barriera

La parte di codice più significativa dell'intero progetto è senza ombra di dubbio la sincronizzazione fra i thread. Essa è essenziale al termine di ogni iterazione per poter poi partire con l'operazione stencil successiva. Quest'ultima viene gestita mediante la lock su una variabile di tipo integer che funge da contatore. Ogni thread che acquisisce la lock incrementa il contatore e si mette in attesa che raggiunga il numero massimo (attesa attiva), l'ultimo thread ad arrivare resetta il contatore e segnala tutti gli altri sbloccandoli dall'attesa.

4.3 Compilazione

Per la compilazione del codice è stato utilizzato il Makefile, dove si può scegliere quale implementazione compilare: main, ff, ffnopinned.E' inoltre possibile settare le macro: PARALLEL_WRITE(abilita la scrittura delle immagini in parallelo) e LAST_WRITE(abilita la scrittura dell'immagine risultante). Per il corretto funzionamento dell'applicazione è necessario che nel path del make file venga indicata la directory della libreria FastFlow e che sia presente la cartella src/frames per salvare le immagini.

5 Programming API

Per offrire delle programming API è stata realizzata la classe CellularAutomata.cpp la quale offre appunto un' interfaccia semplice ed efficace per poter eseguire qualsiasi tipo di problema in questo modello computazionale. Per utilizzarlo basterà istaniare l'oggetto CellularAutomata e settare nel costruttore i seguenti parametri:

- matrix: come primo parametro vuole una matrice inizializzata sulla quale compiere le iterazioni.
- rule: una funzione function<int(int, int)> che rappresenta la regola con la quale aggiornare lo stato di una cella in base al vicinato.
- numero di iterazioni: indica il numero di iterazioni che devono essere compiute.
- numero dei thread: indica il numero di thread da utilizzare.

Una volta inizializzato l'oggetto, per avviare la computazione basterà invocare il metodo run().

6 Risultati

In questo capitolo verrano mostrati diversi risultati ottenuti sulla macchina condivisa XEON Phi, utilizzando un numero di thread differenti con una matrice di dimensione 500x500 con le regole descritte da Game of Life di Conway's. Per prima cosa è importante

capire quanto ci si discosta da quelle che sono le stime teoriche fatte nel capitolo 3. Di seguito viene fatto il confronto tra i tempi di esecuzione stimati e reali:

Params	Tseq Teo	Tseq Real	Tpar Teo	Tpar Real	Speedup
500x500 150it 200th	26,45s	29,14s	0,133s	0,402s	72,49
500x500 255it 255th	48,174s	52,04s	0,189s	0,592s	87,90

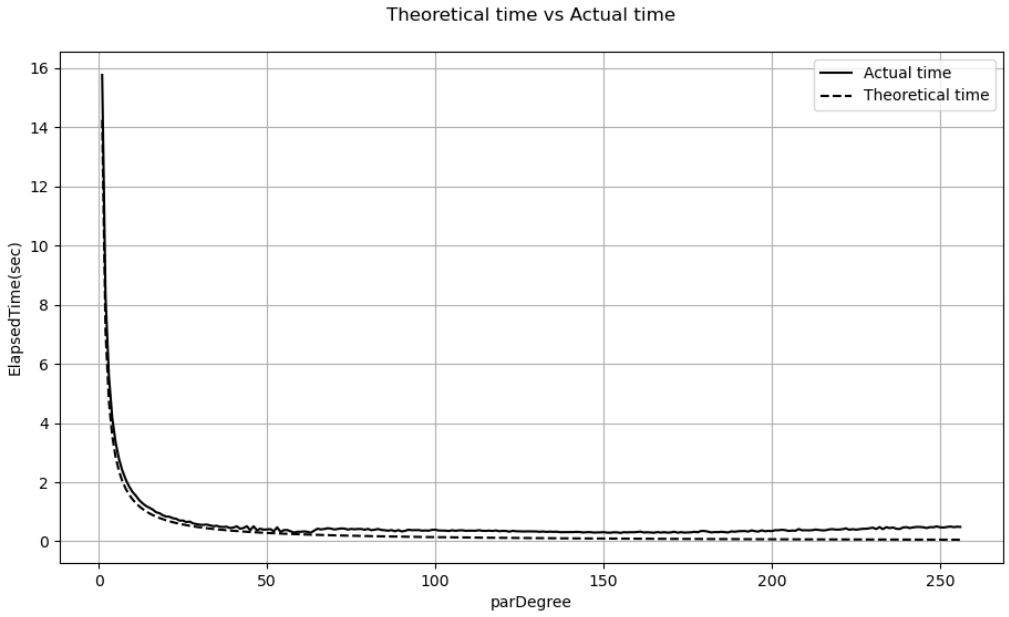
Possiamo vedere come i valori reali siano più alti rispetto ai teorici, questo perchè all'aumentare del numero di thread cresce anche l'overhead di pari passo. Data l'elevata latenza per il calcolo del tempo sequenziale (dovuti alla scrittura delle immagini), si è scelto di effettuare i test successivi disabilitando la scrittura delle immagini, questo anche per il fatto che il test viene eseguito su di una macchina condivisa e quindi con poche risorse disponibili.

Il test è stato svolto sulla matrice 500x500 con 1000iterazioni, lanciando il processo con un differente numero thread e per ogni valore del ParDegree è stata ripetura la procedura per 5 volte calcolando la media. Vedremo a seguire i risultati ottenuti su tre modelli principali:

- Main_impl: rappresenta l'implementazione descritta nella sezione 2.3.
- FF_impl: rappresenta l'implementazione del modello utilizzando il framework FastFlow.
- FF_implNoPinning: rappresenta l'implementazione del modello utilizzando il framework FastFlow con flag NO_DEFAULT_MAPPING il quale disabilita il thread pinning automatico di FastFlow.

6.1 Latency

Il primo metro di paragone utilizzato per confrontare le varie implementazioni è il tempo di esecuzione complessivo dell'applicazione (latency). In questo caso avendo disabilitato la scrittura i tempi saranno ovviamente più bassi e per avere una stima teorica basterà rimuovere dalla formula(3) la parte delle immagini . Di seguito viene fatto un confronto tra i tempi di esecuzione stimati e reali.



Dal grafico si evince che fino a 50 worker la differenza di prestazioni tra stima e realtà è molto sottile. Da lì in poi il tempo di esecuzione della stima tende a 0, mentre il tempo reale tende a stabilizzarsi per poi risalire all'aumentare dei worker. Questo comportamento dell'applicazione è dovuto principalmente al fatto che al crescere dei worker aumenta l'overhead di creazione dei thread.

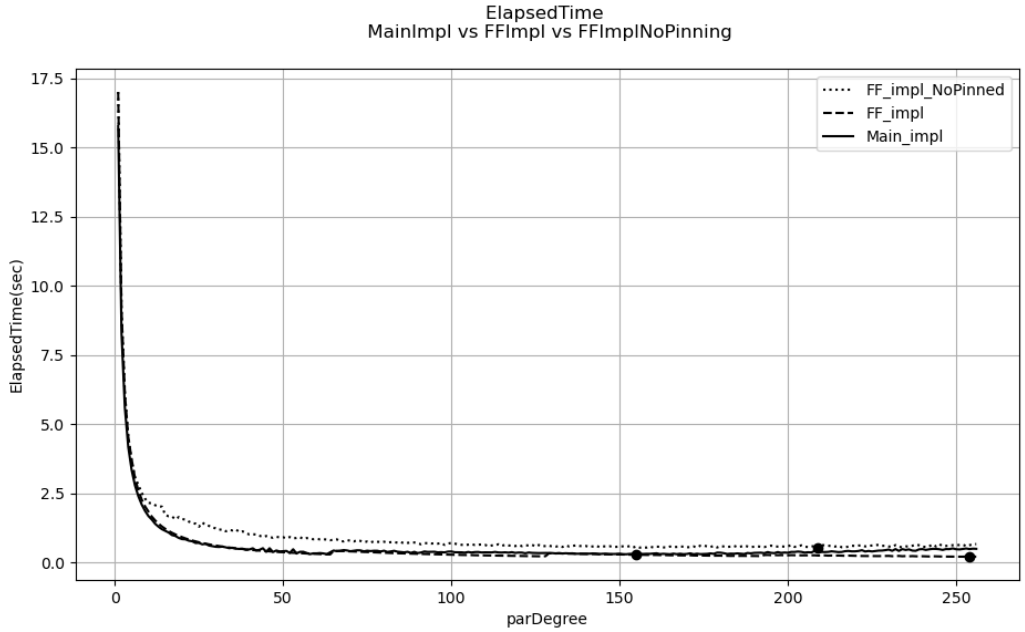


Figura 4: Latency test

Mentre dal grafico riportato sopra, si può notare a grandi linee un andamendo molto simile delle tre funzioni, con la MainImplementation che raggiunge il suo minimo a (0.28s 155th). Con un numero di thread di circa 200 il modello FFImplNoPinning si comporta peggio con (0,51s 209th), mentre le migliori performance le troviamo in FFImpl con (0,20s 254th).

6.2 Speedup

Il secondo metro di paragone tra i due modelli è l'andamento dello speedup, espressa come rapporto tra tempo sequenziale e tempo parallelo con numero variabile di worker.

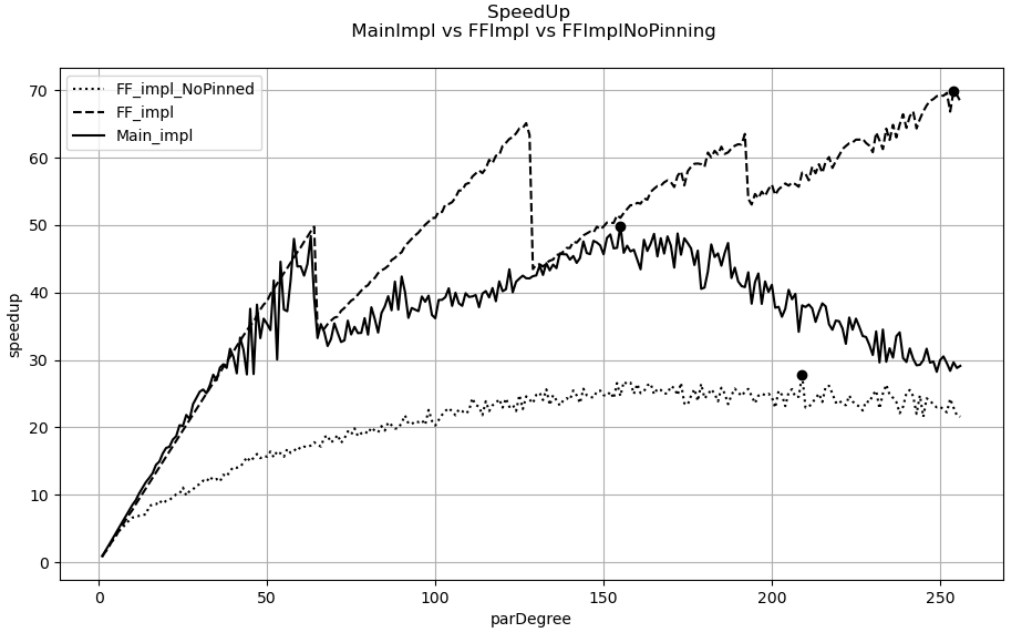


Figura 5: Speedup test

Si nota subito che lo speedup di FFImpl è maggiore (70) rispetto agli altri modelli, ma presenta un andamento al quanto insolito, difatti in concomitanza del parDegree 64, 128, 192 si nota un drastico calo delle prestazioni. Sono stati effettuati dei testi specifici su questi particolari punti, analizzando il chache misses mediante il comando: "perf stat -e cache-misses" senza però avere grandi differenze. Questo calo di prestazioni si presume sia dovuto soprattutto al fatto che il modello FF_impl possiede i thread pinnati e che quindi in concomitanza di certi core specifici per via dell'architettura richiede più tempo, mentre

l'implementazione senza pinning non presenta questa anomalia. Si può osservare come lo speedup di MainImpl comincia ad essere decrescente prima di fastflow, con massimo speedup pari a (50). FF_impl_NoPinned invece raggiunge uno speedup massimo di (27) con un numero di thread molto elevato superiore a (200) quindi ovvero offre un guadagno basso richiedendo allo stesso tempo maggiori risorse.

6.3 Scalability

Il terzo metro di paragone è la scalabilità espressa come rapporto tra tempo parallelo di un worker e tempo parallelo con numero variabile di worker.

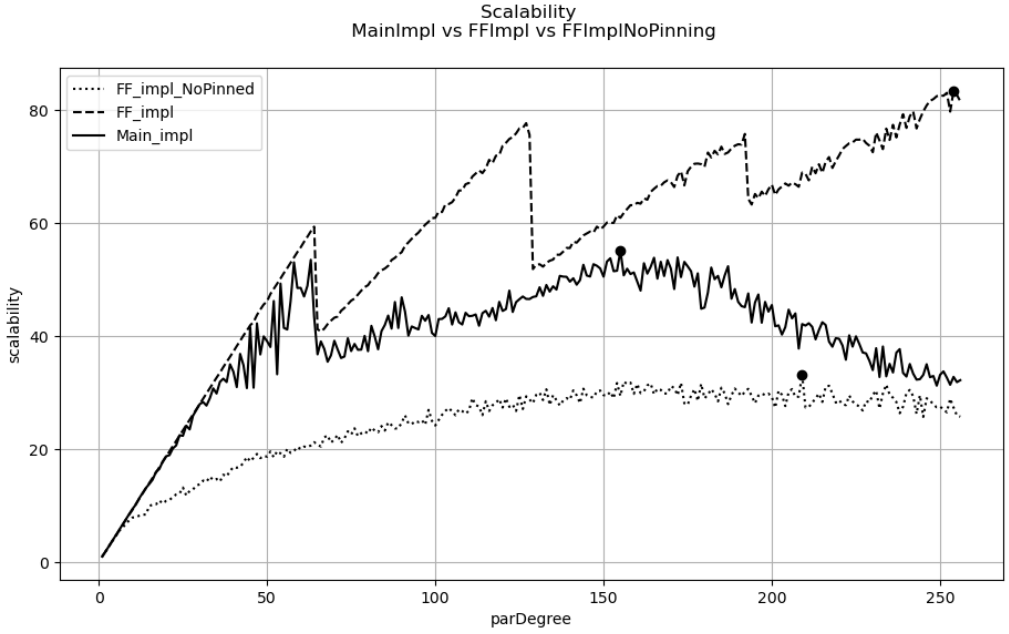


Figura 6: Scalability test

Il grafico rimane qualitativamente identico, conseguenza del fatto che il tempo parallelo con un worker è praticamente uguale al tempo sequenziale (14.24s vs 15.76s). In questo caso i valori di scalability sono (55.1) per Main_Impl, (33.2) per FF_impl_NoPinned e sempre in vetta troviamo l'implementazione di FastFlow pinnata con (83.4).

6.4 Efficiency

L'ultimo metro di paragone è l'efficiency espressa come rapporto tra speedup e numero di worker utilizzati. Questo ci consente di capire quanto effettivamente è utile aumentare il numero di worker a disposizione. L'efficiency comincia ad essere bassa ovvero inferiore al 60%, da 10-11 worker per FF_implNoPinned, mentre per Main_impl e con FF_impl da circa 64worker. Si può notare anche come FF_impl sia comunque sempre più efficiente di Main_impl all'aumentare dei thread.

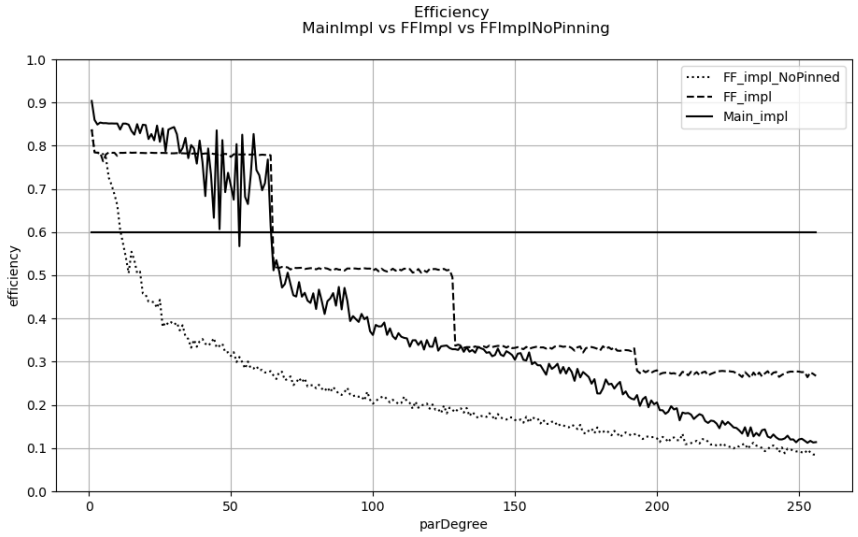


Figura 7: Efficiency test

7 Considerazioni finali

Giunti al termine di questo report risulta chiaro che l'implementazioni tramite FastFlow ha prestazioni migliori rispetto a Main_impl. Si può senz'altro dire che l'applicazione funziona discretamente su un architettura multi-core, consentendo di compiere 150 frames di una matrice 500x500 e scrivere i fotogrammi in 0,402s. Risultato sicuramente ottimo se paragonato alla versione sequenziale che richiedeva come visto 29,14s. Un'altra versione possibile avrebbe potuto effettuare il padding dei valori della matrice riducendo così il false sharing e possibilmente diminuendo le oscillazioni della funzione. Dall'implementazione FastFlow inoltre si è visto un miglioramento al crescere del numero di worker richiesti, ed un minor numero di righe di codice di complessità di sviluppo richiesta.