Relazione progetto esame Algoritmi Paralleli e Sistemi Distribuiti

Francesco Gallo

27/06/2022

Indice

I Prefazione **2**

II Progettazione Concettuale **2**

III Implementazione **4**

1 Struttura delle celle **5**

2 Il modello **5**

IV Parallelizzazione **6**

3 Struttura **6**

4 Prove simulative **7**

5 Valori prestazionali **9**

Parte I

Prefazione

Per lo sviluppo di questo progetto non si è attinto ad alcune fonte esterna (mio malgrado e anche stranamente, online pare non esista alcuna implementazione ne tanto meno modello logico per rappresentare la struttura di questo progetto), tutta la struttura, la fisica e la logica alla base del progetto sono stati completamente ideati e implementati da me, perciò potrebbero esserci alcune imprecisioni nalla rappresentazione del fenomeno preso in esame.

Parte II

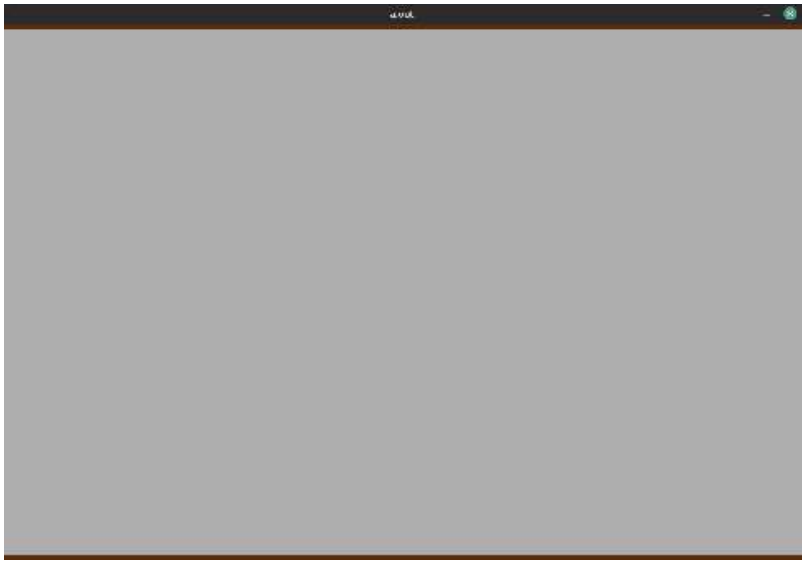
Progettazione Concettuale

L'idea di base per il progetto è stata un modello che simulasse il processo di formazione di stalattiti e stalagmiti all'interno di una caverna, la struttura del progetto a livello logico è quindi una matrice in cui le celle possono assumere 3 stati differenti:

- aria
- roccia
- acqua

ognuno di questi stati è definito nella struttura interna di ogni cella della matrice.

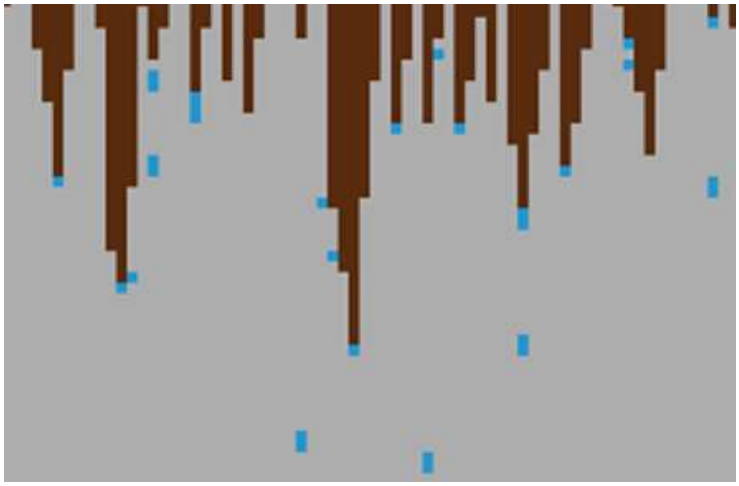
Il funzionamento è il seguente, i bordi superiori e inferiori vengono inizializzati allo stato di roccia per simulare il suolo ed il margine superiore della caverna;



Dopo di che l'idea è che vengano generate in modo casuale e con percentuale variabile delle gocce d'acqua sulla parete superiore della caverna, le gocce soggette ad una pseudo-fisica definita nel programma tenderanno a cadere.

Ogni goccia d'acqua al suo passaggio lascerà dei sedimenti nella cella in cui si trova e come è ovvio l'accumulo di sedimenti col passare del tempo andrà a formare della roccia.

Per come è stata concepita la goccia d'acqua può scorrere sulla roccia, ma non può cadere quando è *attaccata* ad una roccia superiormente a meno che non abbia un certo *peso* ovvero una certa quantità d'acqua al suo interno, si è fatta questa scelta per emulare nel modo più realistico possibile il fenomeno nella sua forma naturale e quindi cercando di simulare il comportamento fisico dei tipi ti materia utilizzati.



Così come in natura è possibile che la stalagmite presenti una struttura con una piccola percentuale di porosità, ciò può accadere anche in questo modello (non mentirò e dico subito che quest'ultima *feature* è stata implementata in modo del tutto fortuito).

Parte III
Implementazione

Per semplicità di comprensione la struttura implementativa verrà trattata come fosse sequenziale, e successivamente verrà introdotta la parellizzazione.

Per implementare il sopra spiegato modello si è optato per un vettore dinamico per favorire l'efficienza avendo così i dati contigui in memoria, che per comodità di utilizzo e soprattutto per semplicità di lettura e stesura del codice, verrà trattato in tutto il codice del programma come una matrice grazie ad una funzione che si occuperà di convertire le coordinate matriciali in un indice per accedere alla corrispondente cella del vettore. In particolare la rappresentazione virtuale del vettore sottoforma di matrice vede il vettore muoversi lungo le colonne della matrice come segue:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & .. \end{bmatrix} - - - > \begin{bmatrix} 0 & 3 & .. \\ 1 & 4 & .. \\ 2 & .. & .. \end{bmatrix} \tag{0.1}$$

si è optato per questa scelta per via della prevalente verticale struttura dei movimenti all'interno del modello.

Da qui in poi parleremo di matrice per facilità esplicativa, ma si alluderà in verità ad un vettore.

1 Struttura delle celle

Per strutturare le singole celle lo stratagemma adottato è alquanto particolare, a scelta è ricaduta sulla seguente soluzione poichè per gestire la complessa struttura delle celle sarebbe servita una **struct** che a livello di memoria sarebbe stata particolarmente pesante, senza contare che avrebbe ampiamente inficiato le performances. La struttura di ogni cella è quindi di soli 8 bit, in realtà ne sarebbero serviti solo 5 ma purtroppo in memoria la dimensione minima che si può salvare è un byte.

Di questi bit i primi 2 andranno a contenere la quantità d'acqua (da 0 a 3), mentre i successivi 3 la quantità di sedimento (da 0 a 6), in particolare:

- se l'aqua è 0 e il sedimento è < 6, allora la cella verrà vista come una cella d'aria;
- se il sedimento ha valore 6 allora la cella viene vista come roccia;
- nel caso in cui il valore dell'acqua sia >0 la cella ovviamente verrà vista come aqua.

Per far si che questi bit in memoria siano tutti contigui, si è fatto uso della **union** messa a disposizione dal linguaggio C, per evitare di appesantire troppo la spiegazione se si vogliono conoscere i dettagli implementativi si consulti il codice sorgente.

2 Il modello

Il modello su cui si basa l'implementazione è quello di un automa cellulare, quindi ogni cella avrà dei vicini e in particolare viene utilizzato il vicinato di Moore; in base valore dei vicini verrà settato il valore della cella corrente.

Tutto il processo si svolge utilizzando 2 matrici, una di lettura e una di scrittura, e procede come segue:

1. viene letta una cella dalla matrice di lettura;
2. in base al valore dei vicini verrà settato l'opportuno valore nella corrispettiva cella nella matrice di scrittura;
3. si ritorna al punto 1 finchè non si è iterata tutta la matrice di lettura;

4. avviene ora uno swap tra la matrice di lettura e quella di scrittura, cosicché la matrice di scrittura venga considerata la nuova matrice di lettura e viceversa;

Questo processo verrà iterato per un numero N di volte, il cosiddetto numero di step.

Applicazione 2.1 *Per la rappresentazione grafica invece è stata scelta la libreria Allegro5.*

Parte IV
Parallelizzazione

3 Struttura

Per rendere il programma parallelo, ovvero fare in modo che il programma venga eseguito *contemporaneamente* su più processori, in particolare sui vari cores della cpu. Il programma per fare ciò utilizza il paradigma del message passing, quindi più processori eseguono in modo concorrente una loro copia del programma, ogni processo ha una propia memoria locale e quindi non potrà accedere alle risorse di un altro. Il message passing consiste quindi nella comunicazione tra i vari processi attraverso lo scambio di messaggi.

Per fare ciò si è optato per l'utilizzo della libreria MPI, che fornisce all'utente un'interfaccia per gestire il message passing. Ogni processo avrà quindi una sua sotto-matrice, infatti dichiarando una variabili globalmente questa verrà copiata in ogni processo, in questo modo viene evitato il classico *master-slave* che invece impone che sia il processo principale a inviare agli altri pezzi di dati.

Per la realizzazione del programma viene utilizzata una topologia virtuale cartesiana; in poche parole viene fatto una sorta di mapping virtuale dei processi, in modo tale che sia possibile rappresentarli logicamente nello spazio e quindi avere una maggiore semplicità di implementazione oltre che ad un incremento dell'efficienza; in particolare la struttura logica che assumono i vari processi potrebbe essere rappresentata come segue:

$$[P_1] \quad \rightarrow \quad [P_2] \quad \rightarrow \quad [P_3] \quad \rightarrow \quad .. \tag{3.1}$$

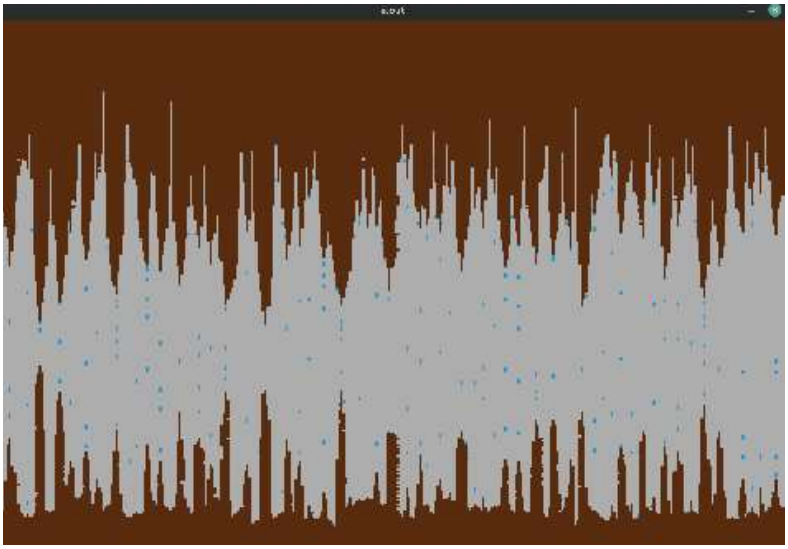
I processi sono quindi distribuiti lungo una dimensione, quella orizzontale; si è optato per questa scelta poichè l'overhead introdotto dalle comunicazioni

dovrebbe essere minore, è stata esclusa una topologia a due dimensioni poichè inadeguata al tipo di modello da rappresentare.

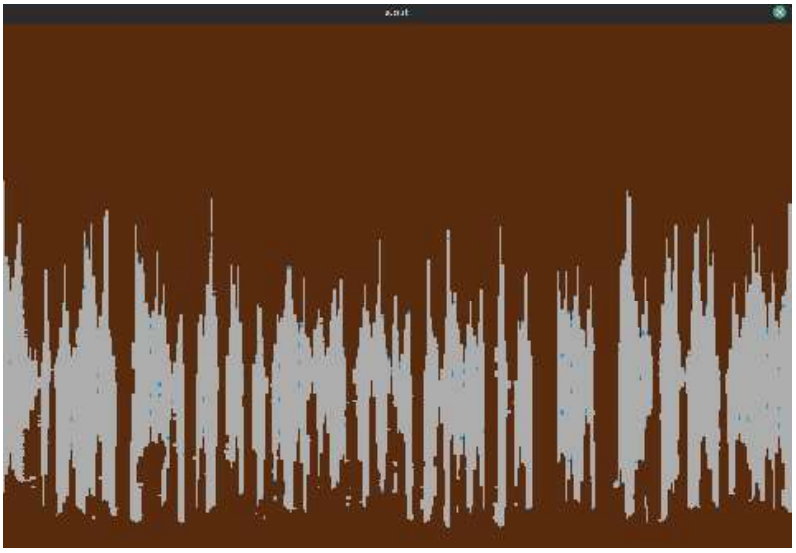
Anche per il passaggio dei dati sono state definiti dei nuovi tipi di dato (MPI Datatypes), questi verranno utilizzati per agevolare gli invii e le ricezioni dei messaggi tra i processori, in particolare un Datatype rappresenta la colonna della matrice, un altro la sotto-matrice stessa, quest'ultimo viene utilizzato solo per la stampa tramite allegro, poichè quest'ulitmo permette ad un solo processo di stampare sul display.

4 Prove simulative

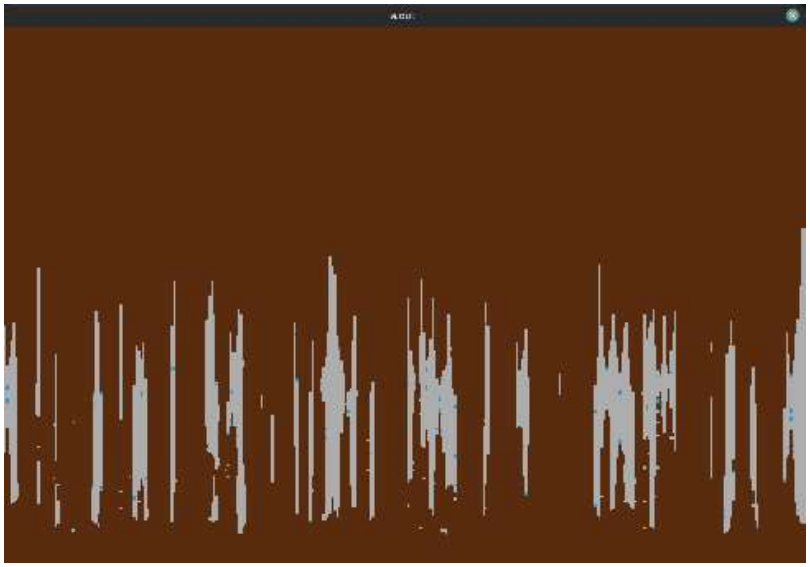
Il programma viene fatto iterare per un certo numero di passi e in base a quante iterazioni vengono effettuate possiamo osservare lo stato globale delle varie matrici, in particolare:



dopo 3500 iterazioni..



dopo 5000 iterazioni...



dopo 5500 iterazioni...

5 Valori prestazionali

Ora diamo un'occhiata a come la parallelizzazione giovi alle prestazioni, osserviamo oltre ai tempi con diverso numero di processori, anche i tempi con differenti dimensioni delle matrici:

Remark 1 ***Claim 2 Osservazione 5.1 Applicazione 5.1*** *Tutti i tempi sono stati calcolati ognuno sulla base di 10 simulazioni. Hardware di riferimento: - Processore: intel core i7-9750h, Ram: 16 Gb DDR4 2666MHz, GPU: Nvidia GTX 1660 Ti 6GB, portatile collegato all'alimentazione. I tempi sono stati presi ovviamente disattivando l'implementazione grafica.*

dimensione 240*360 tempi presi dopo 1000 steps

NProcs	1	2	3	4	5	6
Tempo	2.1398	1.1095	0.7295	0.5782	0.4877	0.4179
SpeedUp	1	1.9286	2.9332	3.7008	4.3875	5.1204
Efficienza	1	0.9643	0.9777	0.9252	0.8775	0.8534

Come possiamo osservare lo speedUp conferma a pieno il guadagno in termini di tempo introdotto dalla parallelizzazione, ma come testimonia l'efficienza, le performance non assumono un miglioramento lineare, in particolare con l'aumentare del numero di processori l'efficienza diminuisce, questo è dato dal fatto che c'è un maggior overhead dovuto alla quantità maggiore di comunicazioni.

dimensione 480*720 tempi presi dopo 1000 steps

NProcs	1	2	3	4	5	6
Tempo	8.4592	4.3579	2.9615	2.2371	1.9003	1.5592
SpeedUp	1	1.9318	2.8564	3.7813	4.4515	5.4253
Efficienza	1	0.9659	0.9521	0.9453	0.8903	0.9042

Aumentando invece le dimensioni del problema in termini di dati possiamo osservare come l'efficienza tende a migliorare rispetto al caso precedente, ciò è dovuto al fatto che le comunicazioni avvengono in proporzione minore poichè la grana del parallelismo va a diventare più grossa, infatti tra 2 comunicazioni viene fatto molto più lavoro.

dimensione 120.*180 tempi presi dopo 3500 steps

NProcs	1	2	3	4	5	6
Tempo	1.6859	0.8391	0.5199	0.3547	0.2792	0.2648
SpeedUp	1	2.0092	3.2427	4.7530	6.0383	6.3667
Efficienza	1	1.0046	1.0809	1.1883	1.2077	1.0611

Riducendo la dimensione delle matrici otteniamo uno speedUp superlineare, in questo particolare programma la ragione può essere che essendo le matrici di byte ed essendo in questo caso relativamente piccole, siano allineate alla dimensione della cache interna dei processori e quindi i tempi di accesso sono notevolmente ridotti

dimensione 240*360 tempi presi dopo 3500 step

NProcs	1	2	3	4	5	6
Tempo	7.1768	3.7586	2.3937	1.8879	1.5489	1.3415
SpeedUp	1	1.9094	2.9982	3.8015	4.6335	5.3498
Efficienza	1	0.9547	0.9994	0.9503	0.9267	0.8916

Aumentando infatti la dimensione ma mantenendo lo steso numero di passi osserviamo un ritorno ai normali valori di efficienza registrati precedentemente, poichè il caso particolare sopracitato non è più verificato.

In conclusione possiamo quindi affermare che il programma ha una buona scalabilità, in particolare l'efficienza migliora all'aumentare delle dimensioni del problema.