PC-2023/24 Mid Term Project

Melina Berrio Pari
melina.berrio@edu.unifi.it

Abstract

Il progetto implementa un risolutore di labirinti in C++ che utilizza DFS per la generazione e un algoritmo casuale per la risoluzione. Ottimizzato con OpenMP, il sistema distribuisce il carico computazionale tra più core, migliorando le prestazioni rispetto alla versione sequenziale.

1. Generazione del laberinto

L'algoritmo di generazione del labirinto utilizza il metodo di ricerca in profondità (DFS), che esplora le celle adiacenti in modo ricorsivo. Questo approccio garantisce che il labirinto generato sia complesso e abbia un solo percorso valido tra qualsiasi coppia di celle, evitando cicli e garantendo una struttura coerente e risolvibile.

Pseudocodice:
Input: dimensioni del labirinto

1. Inizializzazione d'una griglia piena di muri
2. Scegliere un punto di partenza e rimuovere i muri adiacenti
3. Iterazione: DFS
   While ci sono celle non visitate:
   - Seleziona una cella corrente
   - Seleziona una cella adiacente non visitata:
     Rimuovi il muro
     Marca la cella come visitata e aggiorna la cella corrente
     Altrimenti, torna alla cella precedente.

Output: Labirinto salvato su file.

Per la gestione del stesso si è creata una classe "Maze" di cui gli attributi sono:

- *grid*: Griglia 2D che rappresenta il labirinto. Ogni cella può indicare un muro, passaggio, l'uscita, l'inizio.
- *startX, startY, exitX, exitY:* Coordinate della partenza e uscita

I metodi principali sono per salvare e caricare un laberinto saveToFile, loadFromFile:

2. Implementazione Sequenziale

L'implementazione sequenziale del risolutore di labirinti è progettata per simulare l'esplorazione di particelle alla ricerca dell'uscita in un labirinto. La classe Particle gioca un ruolo cruciale in questa simulazione, gestendo i movimenti e le azioni delle particelle. Gli attributi principali della classe sono:

- x, y: le coordinate correnti della particella nel labirinto.
- visitedCells è un vettore che memorizza tutte le celle visitate dalla particella durante la sua esplorazione.

La funzione **move** è responsabile della gestione del movimento della particella nel labirinto. Essa utilizza un set di direzioni predefinite (nord, sud, est, ovest) per determinare il passo successivo. La particella si sposta solo se la nuova posizione è valida e non è bloccata da un muro del labirinto.

La funzione g**etVisitedCells** restituisce il vettore contenente tutte le celle visitate dalla particella, permettendo di analizzare il percorso esplorato.

La simulazione viene avviata creando e inizializzando un vettore di particelle. Durante il processo, la classe Particlegestisce i movimenti e le azioni di ciascuna particella. La simulazione prosegue fino a quando almeno una particella trova l'uscita del labirinto. Ad ogni iterazione, ogni particella si muove nel labirinto, e i percorsi esplorati vengono aggiornati. I tempi di esecuzione per ogni configurazione di particelle vengono misurati e registrati utilizzando la libreria chrono di C++.

3. Implementazione Parallela

Per ottimizzare le prestazioni della simulazione, il ciclo principale che gestisce il movimento delle particelle è stato parallelizzato utilizzando OpenMP. La direttiva `#pragma omp for` è utilizzata per distribuire il carico di lavoro della simulazione tra i thread disponibili, consentendo a ciascun thread di gestire un gruppo di particelle in parallelo. Questo approccio sfrutta le capacità di elaborazione multi-core, riducendo significativamente il tempo di esecuzione complessivo.

La sincronizzazione dei dati condivisi è cruciale quando più thread possono modificare variabili condivise contemporaneamente. Per garantire l'integrità dei dati e prevenire conflitti, si utilizza la direttiva `#pragma omp critical` . Questa direttiva garantisce che solo un thread alla volta possa accedere e modificare le variabili globali all'interno della sezione critica.

Nel contesto della simulazione, le variabili che richiedono una gestione attenta sono:

- **foundExit**: Indica se almeno una particella ha trovato l'uscita.
- **particleThatFoundExit**: Memorizza l'indice della particella che ha trovato l'uscita.

Dopo che tutte le particelle hanno completato i loro movimenti, è essenziale aggregare i risultati. La sincronizzazione tra i thread assicura che tutte le operazioni parallele siano completate correttamente prima di procedere con la raccolta dei risultati finali. Se almeno una particella ha trovato l'uscita, il risultato viene registrato e i dettagli del percorso di questa particella vengono salvati.

4. Applicazione

Per valutare le performance e l'efficacia dell'implementazione, è stato considerato un laberinto di 50x50 con numeri di particelle fissati a 50 e 100. Inoltre, sono state testate varie configurazioni di core: 2, 4, 6, 8 e 12.

Table 1. Tempo di esecuzione (secondi) in modo sequenziale

Particle	Time
50	65.9904
100	165.6108

Le tabelle 2 e 3 mostrano i tempi di esecuzione per le simulazioni parallele con 50 e 100 particelle rispettivamente, utilizzando diverse configurazioni di core. La colonna "Speedup" rappresenta il miglioramento delle prestazioni rispetto all'esecuzione sequenziale e si calcola come il rapporto tra il tempo di esecuzione sequenziale e il tempo di esecuzione parallelo:

$$SpeedUp = \frac{T_s}{T_p}$$

Table 2. Tempo di esecuzione (secondi) in modo parallelo per 50 particelle

Mode	Time	Speedup
1-sequential	65.9904	
2-cores	22.6146	2.918044095
4-cores	42.445	1.554727294
6-cores	35.6578	1.850658201

8-cores	8.9655	7.360481847
10-cores	1.479	44.61825558
12-cores	4.6958	14.0530687

Table 3. Tempo di esecuzione (secondi) in modo parallelo per 100 particelle

Mode	Time	Speedup
1-sequential	165.6108	
2-cores	210.2548	0.787667154
4-cores	158.445	1.045225788
6-cores	8.652	19.14133148
8-cores	49.7745	3.32722177
10-cores	6.8178	24.29094429
12-cores	2.9304	56.51474201

I risultati dei test con 50 particelle mostrano un notevole miglioramento dei tempi di esecuzione con l'uso di più core. L'esecuzione parallela con 12 core ha dimostrato uno speedup di 14.05 rispetto al caso sequenziale, evidenziando un miglioramento significativo delle prestazioni. Tuttavia, l'efficacia della parallelizzazione può variare a seconda della configurazione e della dimensione del problema, come per 100 particelle con 2 core, lo speedup è inferiore a 1 (0.79), suggerendo che l'overhead della parallelizzazione supera i benefici per una configurazione con pochi core. Questo fenomeno può essere attribuito al costo di sincronizzazione e gestione dei thread che può influire negativamente su problemi di dimensioni relativamente piccole.

Con 10 e 12 core, si raggiungono valori elevati di speedup (fino a 56.51 per 100 particelle con 12 core), mostrando che l'uso di un numero maggiore di core è altamente vantaggioso quando il problema è sufficientemente grande da giustificare la parallelizzazione.