PC-2022/23 Maze Labyrinth Solver

Francesco Bellezza
E-mail address
francesco.bellezza@stud.unifi.it

Abstract

All'interno di questo progetto abbiamo costruito in linguaggio Python un algoritmo che crea e risolve i labirinti mediante il movimento casuale di una particella che viene generata all'entrata del labirinto e si muove in maniera casuale fino a quando non raggiunge la fine del percorso. Abbiamo realizzato anche una versione parallela del programma e l'abbiamo confrontata con quella sequenziale. I test sono stati effettuati su un pc contenente un Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. Solitamente l'algoritmo parallelo performa meglio, ma l'aumento di performance è molto più evidente per labirinti di dimensione più grande. Inoltre, abbiamo notato che lo speedup medio è molto variabile nei vari esperimenti: questo potrebbe essere dovuto anche alla natura casuale dell'algoritmo.

1. L'algoritmo

Per creare l'intero processo abbiamo sfruttato una suddivisione in classi. Le classi principali utilizzate in questo progetto sono le seguenti:

- Labyrinth, classe preposta alla definizione della struttura di un labirinto. Un labirinto viene rappresentato come una griglia di LabyrinthCell.
- LabyrinthCell, classe adibita alla definizione di una cella del labirinto. Una cella all'interno del labirinto può avere un massimo di 4 muri: la presenza di un muro in una delle possibili 4 direzioni (Nord,Sud,Est,Ovest) viene definita mediante un apposita variabile booleana per ciascun punto cardinale.
- GenerateLabyrinth, classe adibita alla creazione del labirinto. Abbiamo adottato una semplificazione dell'algoritmo di Wilson.
- DrawLabyrinthSolve, classe adibita al disegnare il labirinto e alla sua risoluzione attraverso il movimento delle particelle.

1.1. Generazione del labirinto

Per la generazione del labirinto è stata adoperata una versione semplificata dell'algoritmo di Wilson. Esso consiste nei seguenti passaggi:

- Inserire tutte le celle del labirinto come celle non visitate.
- Scegliere una cella casuale da inserire nel labirinto.
- Scegliere un'altra cella casuale tra quelle che non sono presenti nel labirinto. Da questa cella partiamo per costruire un percorso. Il percorso termina quando raggiunge una cella che è già presente nel labirinto.
- Quando ciò accade, bisogna aggiungere tutte le celle del percorso all'interno del labirinto e riselezionare una cella casuale all'interno del labirinto che non fa ancora parte delle celle visitate e ricostruire un altro percorso come fatto precedentemente. L'algoritmo termina quando tutte le celle sono state visitate.

1.2. Il codice sequenziale

Il metodo di interesse che implementa la creazione della particella che si muove all'interno del labirinto si chiama find_exit_with_random_particles. Il fulcro del metodo consiste nel ciclo while, dove si esce da tale ciclo solo ed esclusivamente se si raggiunge l'uscita del labirinto. La variabile FOUND_PATH che si trova in un file apposito viene utilizzata per l'approccio parallelo e verrà spiegata nel prossimo capitolo. La particella prende una decisione e genera una delle possibili direzioni, (compito del metodo get_random_point_direction). Dopo aver generato la direzione (che può essere una qualunque dei 8 punti cardinali), si genera il movimento che viene visto come un incremento della posizione della particella nel labirinto - che viene rappresentata tramite coppia (ascissa,ordinata) -.

Il metodo get_allowed_movements ritorna la terna di possibili incrementi o decrementi della (x,y) della particella in base alla direzione. Le direzioni sono definite nel file python directions.py. Successivamente il movimento di x e di y viene stabilito scegliendo casualmente un elemento dalla lista generata dal metodo get_allowed_movements. Se il movimento della particella è consentito, allora aggiungiamo il punto al percorso compiuto dalla particella (che si trova all'interno della lista path_particle). Poi è necessario fare un ulteriore controllo importante: controllare se incrementando la coordinata della particella si raggiunge o meno una nuova cella del labirinto (compito del metodo is_particle_go_to_new_cell). Questo è fondamentale perché, per capire se la particella incontra un muro oppure no, è necessario che venga aggiornata correttamente la posizione spaziale all'interno della griglia. Se per caso ci muoviamo in una nuova cella, aggiorniamo path_grid (che contiene il percorso fatto dalla particle relativa alla griglia) e aggiorniamo la posizione corrente. Per controllare se una particella entra o meno in una nuova cella, è sufficiente controllare se il valore della x o della y è maggiore dei delimiter definiti ad inizio metodo. I delimiter sono determinati mediante la lunghezza del singolo muro. Per controllare se la particella si possa o meno muovere in una direzione, abbiamo utilizzato la seguente logica: se nella posizione corrente all'interno della griglia la particella si sta muovendo a Nord, allora se per caso supera il delimiter e la particella sta andando a Nord, significa che tale movimento non è contemplato. Questo si applica a tutte le direzioni. Tale controllo però non è sufficiente, infatti bisogna avere particolare attenzione per i movimenti in diagonale. Infatti può succedere che se la particella va a Nord-Est, bisogna anche controllare che nella cella direttamente a Nord-Est sia presente un muro a Sud-Ovest. Questo viene controllato tramite il metodo diagonal_movement_is_possible. Alla fine del metodo principale, usciti dal while, si disegna il percorso che è stato fatto dalla particella, (disegnandolo dall'uscita all'entrata).

1.3. Il codice parallelo

Per quanto riguarda il codice in parallelo, si utilizza il solito metodo. Solo che stavolta, invece che chiamarlo direttamente, si sfrutta la libreria di python Joblib per creare più processi che eseguono lo stesso metodo. La variabile FOUND_PATH viene utilizzata per quando la prima particella riesce effettivamente a completare il labirinto, infatti, qualora dovesse accadere, bisogna accedere a tale variabile condivisa e controllare se è stata già impostata al valore vero. Se così non fosse, allora si imposta a vero, e tutti gli altri programmi terminano. Quando si controlla il contenuto di tale variabile e si vuole sovrascrivere tale valore, per evitare inconsistenze, si usa un Lock, così che l'accesso a quella sezione di codice è consentito solo da un thread per volta.

2. Analisi della correttezza

Per verificare o meno la correttezza di tale codice sono state effettuate numerose prove e test su molti labirinti. E' possibile verificare l'andamento della particella all'interno del labirinto vedendo il file video che viene generato nella apposita directory.

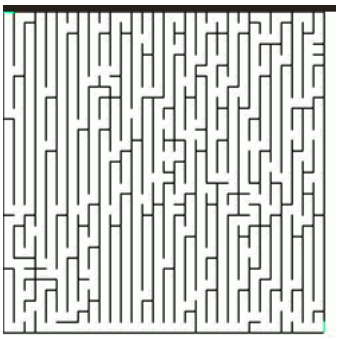
3. Analisi delle prestazioni

3.1. Hardware

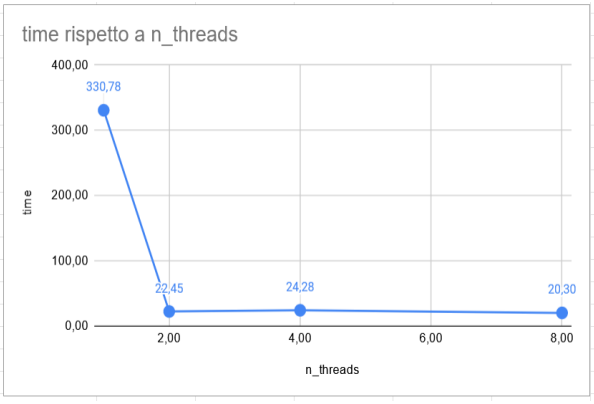
Abbiamo testato questo programma python su un computer con CPU Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. I labirinti vengono generati casualmente e le singole prove sequenziali e quelle con un numero n di thread vengono effettuate sullo stesso labirinto.

3.2. Test su un labirinto di dimensione 30x30

Per il seguente labirinto 30x30:

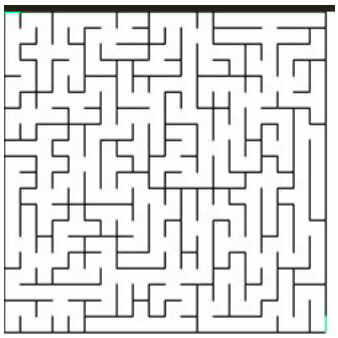


Abbiamo utilizzato prima il metodo sequenziale, successivamente il metodo multithreading e abbiamo ottenuto i seguenti risultati. Sull'asse delle ordinate abbiamo il valore dei secondi, sull'asse delle x il numero dei thread utilizzati. In questo caso il risparmio da un punto di vista temporale è considerevole. Abbiamo infatti uno speedup di circa 14,81.

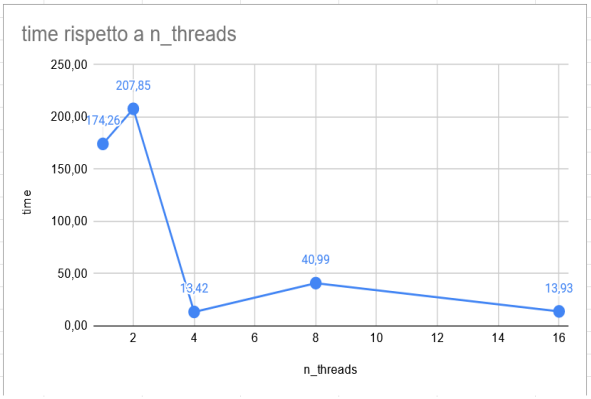


3.3. Test su un labirinto di dimensione 20x20

Proviamo adesso con un labirinto di dimensione più piccola (20x20):

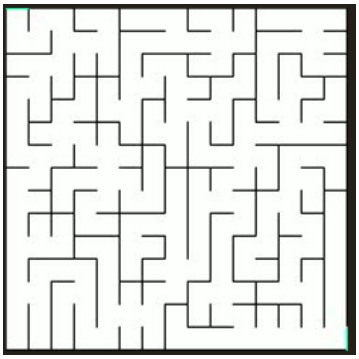


I risultati in questo caso, sono riportati nel seguente grafico:

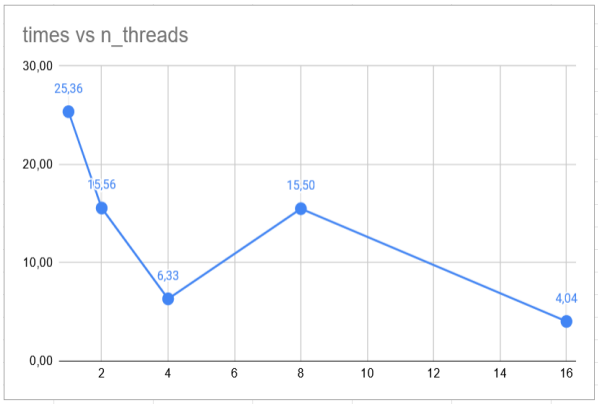


In entrambi i casi che abbiamo visto precedentemente abbiamo prestazioni migliori se scegliamo un adeguato numero di thread. Infatti abbiamo uno speedup di circa 2,5.

3.4. Test su un labirinto di dimensione 15x15



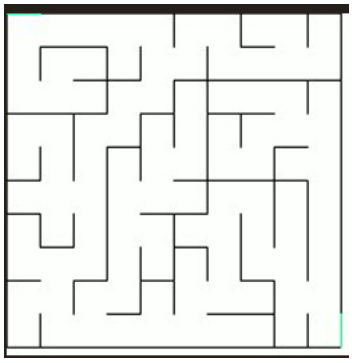
Che ha prodotto i risultati seguenti:



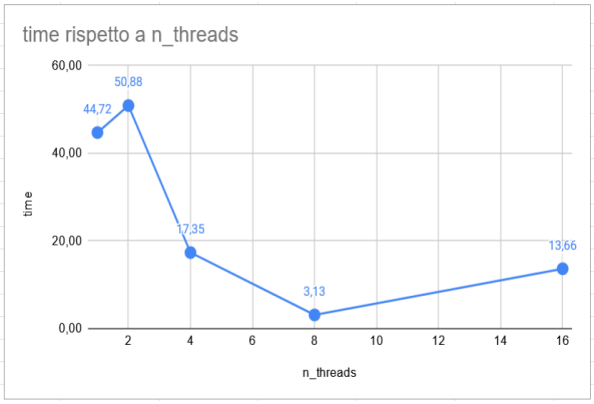
In questo caso lo speedup è di circa 2,10.

3.5. Test su un labirinto di dimensione 10x10

Proviamo adesso a effettuare gli stessi test per un labirinto 10x10:



Abbiamo ottenuto questa volta i seguenti risultati:



In questo caso abbiamo uno speedup di circa 2,45.

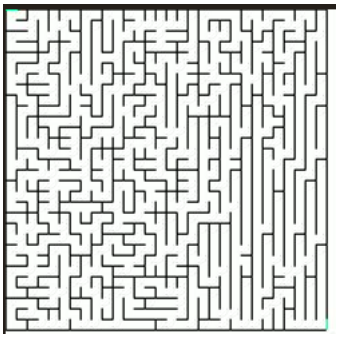
4. Prova su un altro tipo di testing

Abbiamo notato in un precedente esperimento che nel caso in cui si usano solo 2 thread, il programma in parallelo impiega più tempo rispetto a quello sequenziale. Questo come potrebbe essere possibile? Probabilmente ci siamo trovati in un caso sfortunato (ricordando che le particelle si

muovono in maniera casuale, potrebbe succedere che anche se ci sono più particelle che si spostano nella griglia, magari nessuna di queste riesce a trovare il percorso desiderato). Per ovviare a questo problema potremmo effettuare numerose prove dei vari tempi impiegati e successivamente fare una media aritmetica dei tempi. Abbiamo provato a fare questo tipo di test per i labirinti di dimensione 30x30, 10x10 e 15x15 per vedere se possiamo attribuire questo episodio per quanto riguarda il programma in parallelo per 2 thread ad un caso sfortunato oppure no.

4.1. Labirinto 30x30 per 1 thread e 2 thread

Nel caso di questo labirinto:



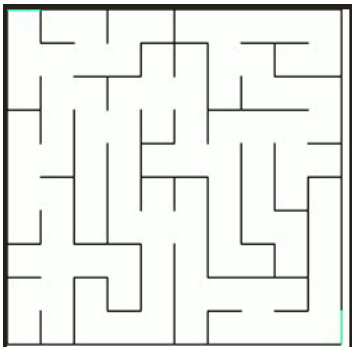
Abbiamo riscontrato i seguenti tempi:

30x30		
	1thread	2threads
time	189,62	51,12
		29,84
		118,19
		106,41
		21,07
		82,36
		626,53
		110,24
		50,32
	189,62	132,90

E' facile notare che rispetto al metodo sequenziale, nonostante ci sia un caso particolarmente sfortunato per 2 thread, si ha comunque uno speedup mediamente maggiore rispetto a quello sequenziale. In questo caso abbiamo uno speedup di circa 1.42. Adesso conduciamo un analisi completa di tempi in media per ogni thread per un labirinto di dimensioni più piccole, per vedere se persiste sempre il problema degli outliers.

4.2. Labirinto 10x10

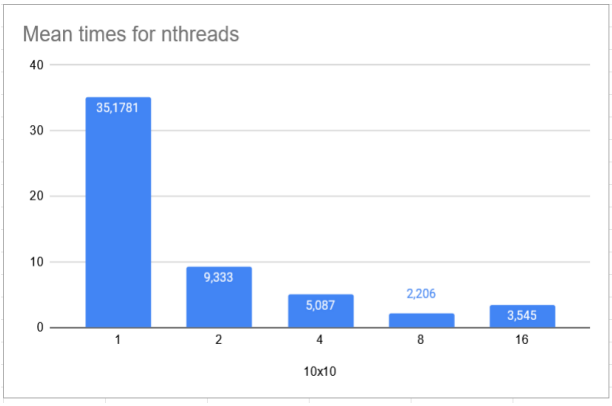
Per questo labirinto 10x10:



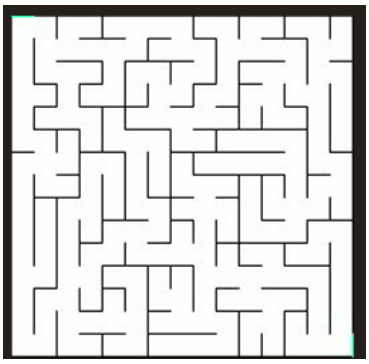
Abbiamo ottenuto questa tabella di tempi di esecuzione:

10x10	1	2	4	8	16
times	0,67	11,55	4,6	1,08	3,43
	11,19	8,7	10,03	6,67	6,17
	16,4	1,49	2,76	0,78	2,24
	135,92	15,29	8,04	3,37	1,2
	2,39	3,02	4,98	2,2	1,63
	18,676	12,57	0,98	1,91	1,78
	4,67	10,3	5,14	0,88	4,41
	148,74	18,78	5,35	3,17	3,4
	9,455	1,35	5,64	0,64	2,34
	3,67	10,28	3,35	1,36	8,85
	35,1781	9,333	5,087	2,206	3,545

Considerando la media di ogni tempo di esecuzione per ogni thread (scritta nell'ultima riga della tabella), abbiamo uno speedup di 6.97 rispetto al tempo sequenziale.

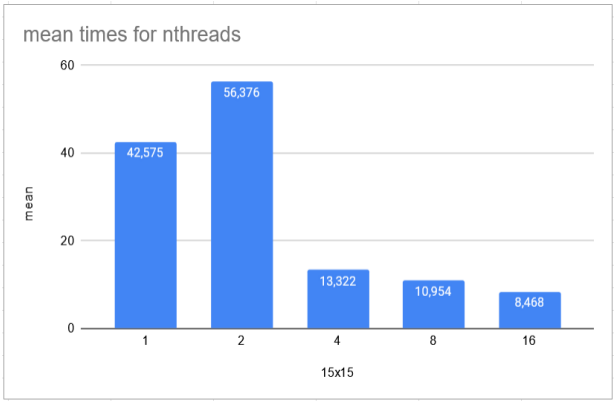


4.3. Labirinto 15x15



15x15	1	2	4	8	16
times	30,85	59,61	5,95	7,35	5,28
	28,98	201,44	6,83	10,99	5,06
	61,03	40,23	8,24	15,25	11,96
	49,33	18,47	5,39	7,97	21,02
	33,3	31,11	21,38	24,79	11,53
	9,93	47,67	7,84	7,23	4,03
	150,45	17,51	12,22	6,11	7,05
	7,24	37,59	6	12,98	4,45
	50,84	29,2	32,53	11,6	10,44
	3,8	80,93	26,84	5,27	3,86
mean	42,575	56,376	13,322	10,954	8,468

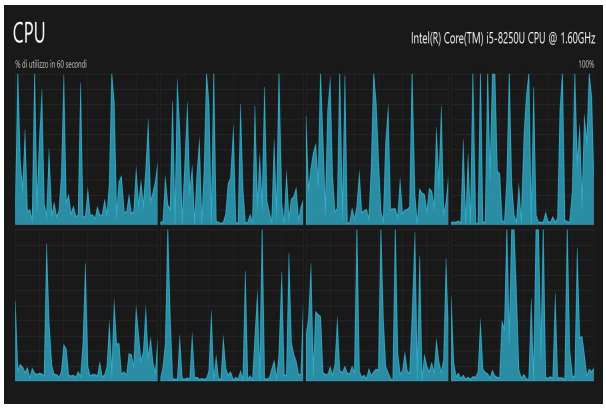
In questo caso, con due thread mediamente il programma parallelo performa peggio rispetto a quello sequenziale. Lo speedup medio generale del codice parallelo rispetto a quello sequenziale è di circa 1,34.



5. Conclusione

L'approccio multithreading è considerevolmente più veloce rispetto all'approccio sequenziale. Questo risulta ragionevole, in quanto più particelle sono implicate nel processo, e quindi più è plausibile supporre che l'algoritmo termini in minor tempo. Con gli ultimi test aggiuntivi abbiamo visto che mediamente performa in maniera migliore il codice parallelo, seppur con speedup differenti. Purtroppo però, con due thread soltanto, esiste la possibilità che il programma performi effettivamente peggio, anche eseguendo più prove ripetute. Solitamente però, con l'aumentare dei thread, questa possibilità non si presenta.

5.1. Esecuzione sequenziale



5.2. Esecuzione parallela