Relazione Homework 01 – PSMC 2016/2017 – Alessandro Manfredi

Approccio al problema e caratteristiche: Il primo approccio effettuato per risolvere il problema dato è stato utilizzare un backtracking che prendesse man mano la cella successiva e verificasse la legalità di un valore, avendo precedentemente creato uno spazio dei possibili valori per quella cella. Il programma creato, però, non mi dava risultati corretti, quindi ho deciso di cambiare approccio e di ricorrere ad un bruteforce.

Il nuovo algoritmo progettato crea un albero partendo dalla prima cella e per ogni soluzione valida crea un sottoalbero. Procedendo di cella in cella, l'albero s'infittirà con tutte le possibili soluzioni, scartando in tempo reale quelle che non sono corrette. Tuttavia, durante la progettazione, mi sono reso conto che la memoria utilizzata era eccessiva quindi per evitare l'overhead, l'algoritmo si salverà tutte le N soluzioni valide delle prime tre righe, per poi riutilizzarle come radici per altri N alberi che verranno risolti uno ad uno tenendo il conto (al fine di verificare quante sono) delle soluzioni corrette.

Nel momento in cui ho dovuto usare il parallelismo avevo già in mente una possibile soluzione, ovvero utilizzare le N soluzioni delle prime righe e creare un thread ("SudokuThread") per risolvere in parallelo gli N alberi, riutilizzando lo stesso algoritmo pensato per il sequenziale.

Il risultato ottenuto è stato positivo, non avendo mai realizzato un progetto in parallelo non mi ero reso conto a livello pratico del tempo effettivo risparmiato, specialmente per calcoli molto grandi.

A seguire elencherò i dati ottenuti e le risposte che ho dedotto dal codice che ho realizzato.

Caratteristiche della piattaforma: I test sono stati effettuati tutti sul mio unico pc:

SO: Windows 10 64bit – CPU: Intel Core i5 2500 @3.30GHz – 8,0 GB RAM - IDE: Eclipse Java Neon

Esperimenti e risposte:

Lo speedup è sempre maggiore di 1? Perché?

Possiamo affermare che lo SpeedUp è sempre **maggiore o uguale** a 1. Per calcoli molto piccoli il parallelismo non da nessun beneficio evidente, anzi si attesta identico al calcolo sequenziale, stabilendo uno SpeedUp fisso a 1. Man mano che lo spazio delle soluzioni si ingrandisce e quindi di conseguenza anche i calcoli effettuati dal calcolatore si nota un netto miglioramento nei tempi di esecuzione in parallelo rispetto ai tempi sequenziali.

Quali istanze richiedono più tempo?

Secondo i risultati ottenuti, le istanze che richiedono più tempo sono quelle con lo spazio di soluzioni più ampio ed un fattore di riempimento più basso.

Esiste una correlazione tra fattore di riempimento, spazio delle soluzioni e tempo di esecuzione?

Il fattore di riempimento ci dice essenzialmente quanto è "piena" la nostra griglia sudoku. Minore è il fattore di riempimento (quindi più la griglia si svuota) maggiore sarà il nostro spazio delle possibili soluzioni, d'altronde ci sono più celle variabili che possono dare origine a più soluzioni legali. Testando le varie griglie date si nota come il tempo di esecuzione aumenti in corrispondenza all'aumento dello spazio delle soluzioni e, quindi, al decremento del fattore di riempimento di una griglia.

Test, dati e grafici degli esperimenti:

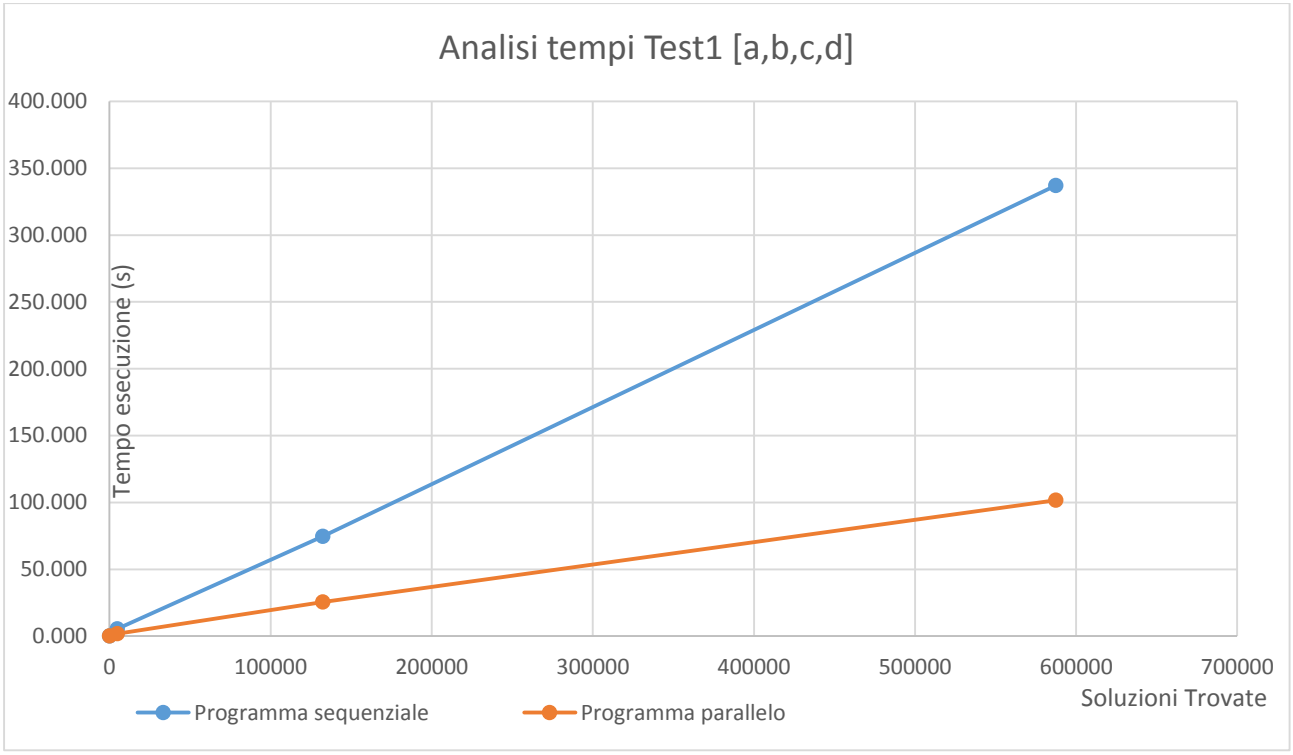
Questi di seguito sono tutti i dati ottenuti dai test effettuati. Se viene eseguito il programma per una di queste istanze i risultati saranno questi, fatta eccezione per i tempi che possono variare di qualche millisecondo.

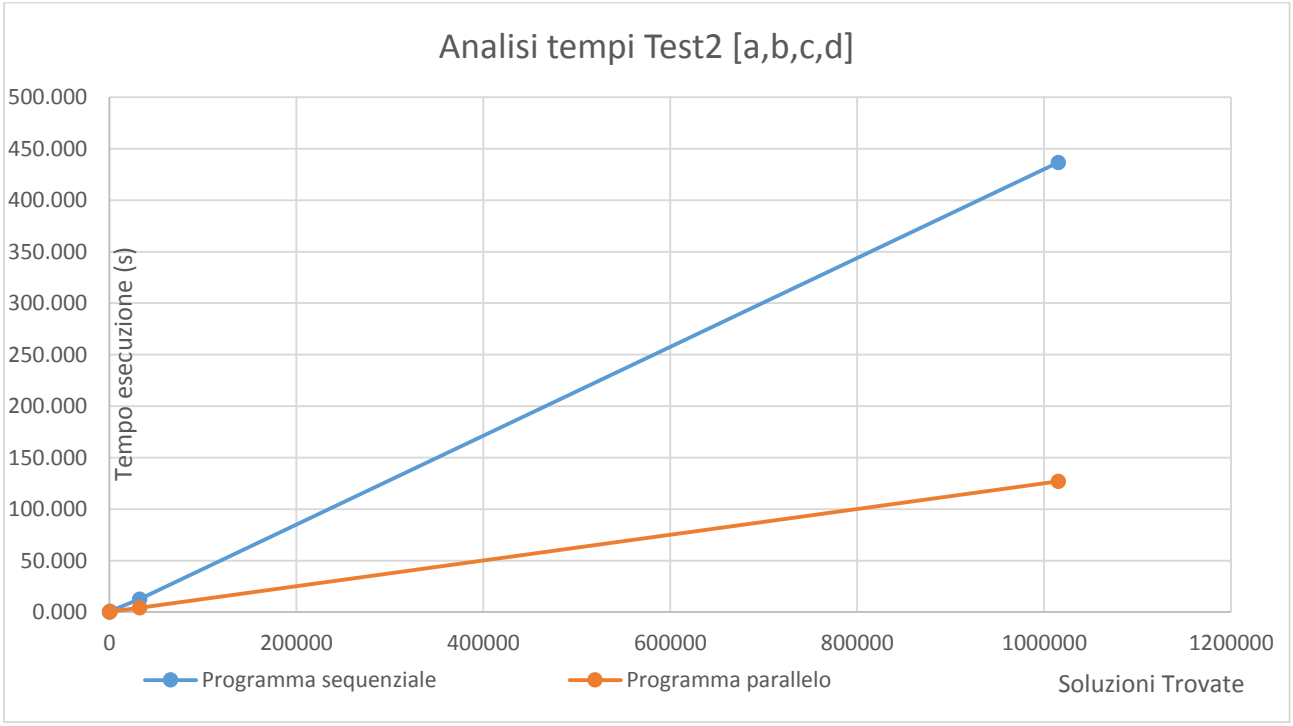
Sudoku in esame	Celle vuote	Fattore di riempimento (%)	Soluzioni trovate
game0	4	96	1
game1	45	45	1
game2	49	40	1
game3	59	28	50142
test1_a	53	35	1
test1_b	59	28	4715
test1_c	61	25	132271
test1_d	62	24	587264
test1_e	63	23	3151964
test2_a	58	29	1
test2_b	60	26	276
test2_c	62	24	32128
test2_d	64	21	1014785
test2_e	65	20	7388360

Sudoku in esame	Tempo di esecuzione (s)	Tempo di esecuzione parallelo (s)	SpeedUp
game0	0,001	0,001	1,000
game1	0,001	0,001	1,000
game2	0,001	0,001	1,000
game3	24,682	8,428	2,928
test1_a	0,028	0,027	1,000
test1_b	5,273	1,701	3,099
test1_c	74,602	25,493	2,926
test1_d	337,146	101,641	3,317
test1_e	1912,378	595,662	3,210
test2_a	0,193	0,099	1,949
test2_b	0,835	0,352	2,372
test2_c	12,638	4,074	3,102
test2_d	436,724	127,094	3,436
test2_e	3589,254	1104,437	3,249

Ho segnato in rosso gli Speed Up più interessanti, come possiamo notare dai dati, più svuotiamo la nostra griglia più il nostro programma in parallelo produce risultati migliori.

Questi due grafici sotto mostrano effettivamente le affermazioni fatte sopra: si può osservare infatti che per i primi test effettuati, i punti sul grafico dei due programmi quasi coincidono mentre se ci spostiamo sui test più difficili la differenza di tempi tra i due programmi è evidente.





Nota: ho omesso il test1_e e il test2_e per motivi di compattezza del grafico, tuttavia tutti i dati sono presenti nella tabella sopra indicata.

Modalità di compilazione:

- Per cambiare l'istanza in esame si deve andare su *Constants.java* e modificare il contenuto della variabile statica *fileInput* con il file di testo che si vuole esaminare. Al momento della consegna è settata l'esecuzione di "game0.txt".
- Per eseguire il programma bisogna semplicemente lanciare la classe *Main*.