Homework 1

Luigi Berducci, Angelo Di Mambro, Karim Mejri

Dicembre 2016

Idea algoritmica

Per implementare l'algoritmo si è usata un'apposita classe ***SudokuMatrix*** che, tralasciando alcuni attributi minori, è composta da 3 strutture dati:

- **valoriCelle**, array di interi di dimensione 81. Ogni cella contiene un valore $[1, 9]$ o *null* nel caso sia vuota;
- **candidatiScartati**, matrice di boolean $81 * 9$. Ogni cella i contiene un array booleano di dimensione 9 tale che il *j-esimo* booleano rappresenta la possibilità di scegliere il *j-esimo* valore come candidato legale della cella *valoriCella[i]*;
- **contatoreCandidatiScartati**, array di interi di dimensione 81. Per ogni cella conta il numero di candidati non permessi.

E che prevede l'implememementazione dei seguenti metodi:

- **checkValidita**, controlla che la configurazione corrente sia ancora valida ovvero che non esista una cella vuota per cui non esista un candidato legale;
- **checkSolution**, controlla se la configurazione corrente è una soluzione verificando che tutte le celle sono state valorizzate;
- **minPossibilityCell**, restituisce la cella con minor numero di candidati possibili.
- **riempiSingoletti**, valorizza tutte le celle per cui esiste un unico candidato legale;
- **aggiornaCandidati**, a seguito della valorizzazione della cella *valoriCelle[i]* effettua un aggiornamento dei candidati per ciascun elemento che si trova nella stessa riga, colonna o box della cella i.

Implementazione sequenziale

L'implementazione sequenziale prova ricorsivamente a valorizzare le celle vuote della griglia in modo coerente con i vincoli del gioco Sudoku.
Ogni inserimento in una data cella è legale in quanto viene effettuato in base alla scelta fatta tra ai suoi possibili candidati. Cosi, una volta raggiunta la completa valorizzazione della griglia, il programma provvederà ad aumentare il contatore del numero di soluzioni trovate.

Nello specifico, l'algoritmo riceve in input un file *.txt* e crea un apposito oggetto *SudokuMatrix*, procedendo poi con l'esecuzione delle seguenti fasi:

1. controlla che la configurazione corrente sia valida (*checkValidita*);

2. verifica il raggiungimento della soluzione (*checkSolution*);
3. valorizza tutte le celle contenenti un solo candidato disponibile (*riempiSingoletti*);
4. estrae la cella i = *minPossibilityCell()*;
5. per ciascun candidato c legale di i,crea una copia dell'oggetto SudokuMatrix modificato , valorizzando la cella i con c;
6. viene mantenuta la consistenza degli inserimenti (*aggiornaCandidati*).
7. eseguito ricorsivamente l'algoritmo sui nuovi oggetti SudokuMatrix ottenuti.

Implementazione parallela

L'idea è quella di ripetere il ragionamento sequenziale applicando il paradigma ForkJoin al posto di effettuare le chiamate ricorsive. Dopo aver effettuato i controlli iniziali, le vengono effettuate seguenti operazioni :

1. Viene estratta la cella (i, j) con minor numero di candidati;
2. Per ogni valore valido v all'interno della cella (i, j), viene creata una copia della SudokuMatrix M ,impostato $copyM[i, j] = v$ ed effettuata la fase di fork creando un thread che prosegue l'esecuzione su tale matrice;
3. Infine vi è la fase di join nella quale viene calcolato il numero di soluzioni ritornate e incrementato il contatore.

Ottimizzazioni

Per ridurre il numero di threads creati e diminuire i tempi di esecuzione, sono state effettuare le seguenti ottimizzazioni:

- **Cutoff sequenziale:** Per ottimizzare la parallelizzazione si è scelto di applicare un cutoff, un soglia sul riempimento della griglia oltre la quale viene eseguito l'algoritmo sequenziale. La soglia da noi impostata è pari a 30 celle riempite, tale valore è stato scelto in base a test sperimentali effettuati e di seguito ne riportiamo una breve parte.
  La tabella riporta i tempi di esecuzione ottenuti al variare della soglia del cutoff. Questo variazione è dovuta al forte abbassamento del numero di thread creati, ai quali viene affidato un maggior carico di lavoro.
  Si può inoltre osservare che l'eccessivo abbassamento della soglia comporta, oltre che ad una diminuzione dei thread creati, un aumento dei tempi di esecuzione in quanto si è reso l'algoritmo più sequenziale, perdendo i vantaggi del parallelismo.

N Thread	CUTOFF	Tempo di esecuzione
21621	35	0m 26.235s
3903	30	0m 22.008s
373	25	0m 23.647s

Table 1: Confronto cutoff eseguiti su *test1_e.txt*

- **Riuso del thread padre:** Dato che il thread dopo aver effettuato le operazioni di *fork()* resta inattivo fino alle relative operazioni di *join()*, si è scelto di effettuare un numero di *fork* pari a $NumeroCandidati - 1$ e di richiamare per l'ultimo candidato *this.compute()* cosicché la computazione

venga eseguita sul thread corrente.
Di seguito riportiamo brevemente alcuni dati che mostrano un dimezzamento del numero di threads creati, in seguito alla suddetta ottimizzazione, e la conseguente diminuzione dei tempi di esecuzione.

N Thread	Ottimizzazione	Tempo di esecuzione
3903	false	0m 22.008s
1888	true	0m 20.617s

Table 2: Esecuzione con/senza ottimizzazione eseguita su *test1_e.txt*

Caratteristiche piattaforma

I test sperimentali sono stati eseguite su diverse piattaforme, di seguito le specifiche tecniche:

Piattaforma	CPU	Freq.CPU	Cores	Hyperthreading	RAM	Sistema Operativo
A	Intel Core2 Q8200	2.34 GHz	4	No	4GB	Windows 7 Home Premium
B	Intel Core i7-2630QM	2.00 GHz	4	Si	4GB	Ubuntu 16.0.4
C	Intel Core i5	1.7 GHz	4	No	4GB	macOS Sierra

Table 3: Piattaforme di testing

Nota: I test sono stati eseguiti su tutte le piattaforme a disposizione ma per questioni di spazio si è scelto di riportare solo i dati dei test eseguiti sulla piattaforma B, sulla quale sono stati più evidenti gli effetti della programmazione parallela.

Fase sperimentale

Piattaforma	Filename	Tempo sequenziale(sec)	Tempo parallelo(sec)	Speedup
B	game0	0.004	0.003	1.333
B	game1	0.001	0.000	—
B	game2	0.001	0.000	—
B	game3	1.152	0.274	4.204
B	test1a	0.001	0.000	—
B	test1b	0.208	0.061	3.409
B	test1c	4.335	0.981	4.439
B	test1d	15.536	3.871	4.013
B	test1e	92.741	34.478	2.689
B	test2a	0.004	0.004	1.000
B	test2b	0.060	0.017	3.529
B	test2c	2.410	0.377	6.392
B	test2d	32.815	7.610	4.312
B	test2e	226.260	84.786	2.668

Table 4: Tabella complessiva dei tempi ottenuti dai due algoritmi eseguiti sulla piattaforma *B*

- Lo speedup è sempre maggiore di 1? Perchè?
  Si lo speedup risulta essere in ogni caso maggiore di 1, salvo nei casi banali nei quali il tempo di esecuzione è prossimo allo zero.
  l'algoritmo da noi sviluppato prevede l'esecuzione in parallelo di più possibili soluzioni , altrimenti eseguite in sequenziale . Pertanto disponendo di 2 o più core è sempre possibile parallelizzare più soluzioni riducendo il tempo di esecuzione e permettendo di avere uno speedup maggiore di 1.

- Quali istanze richiedono più tempo?
  test1e, test2d, e test2e risultano essere le tre istanze che richiedono maggiore tempo di esecuzione sia mediante l'algoritmo parallelo che tramite quello sequenziale, visto l'elevato spazio di soluzioni di questi sudoku.

- Esiste una correlazione tra fattore di riempimento, spazio delle soluzioni e tempo di esecuzione?
  Come viene evidenziato nei grafici sottostanti esiste una correlazione tra tali metriche, in particolare all'aumentare del fattore di riempimento lo spazio delle soluzioni tende a diminuire e di conseguenza anche i tempi di esecuzione. Tuttavia, abbiamo notato come la disposizione dei numeri all'interno della matrice influisca sullo spazio di soluzioni e sui tempi di esecuzione, come mostrano i test sottostanti.

Grafici

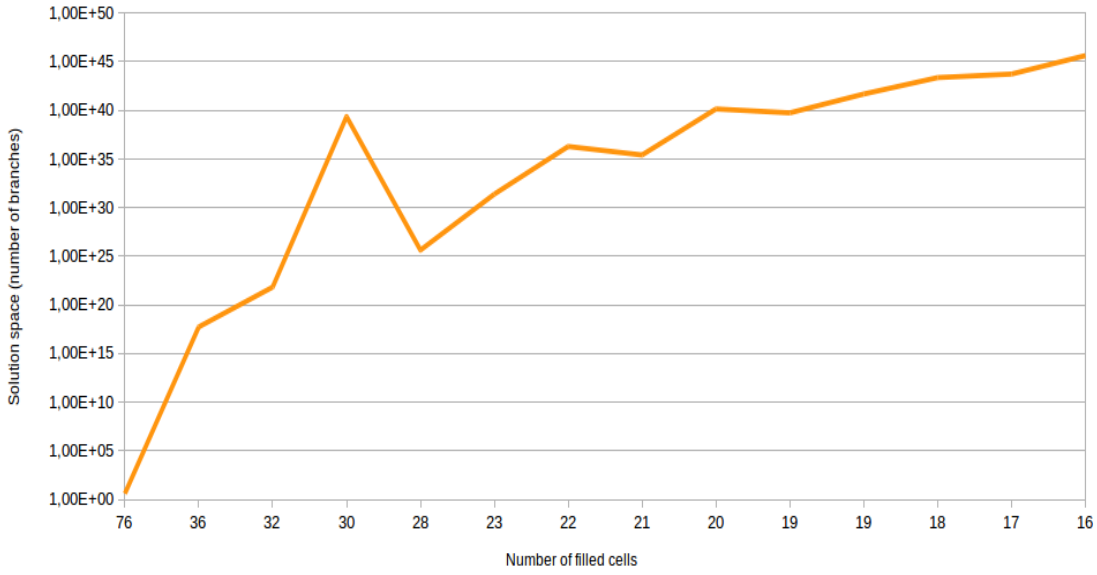

Figure 1: Grafico della crescita dello spazio di soluzioni in correlazione con il numero di celle vuote.
Si tenga conto che, vista l'entità degli spazi delle soluzioni, è stata utilizzata una scala logaritmica sull'asse delle ordinate per garantire maggiore comprensibilità del grafico.
Nota bene: Osservando l'andamento rispetto ai test che presentano fattore di riempimento 32,30 e 28, si può osservare un picco anomalo. Questo mette perfettamente in luce che, a differenza di quanto si possa pensare, per determinare la dimensione dello spazio di soluzioni non basta tener conto del fattore di riempimento. Gioca un ruolo altrettanto importante la disposizione dei valori nella griglia.

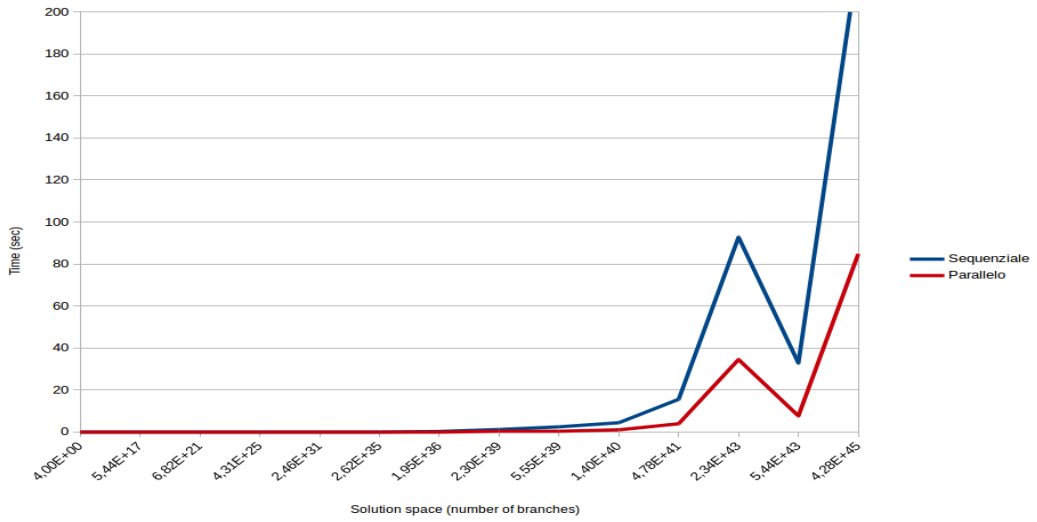

Figure 2: Grafico dei tempi di esecuzione dell'algoritmo sequenziale e parallelo in relazione alla dimensione dello spazio di soluzioni, ottenuti sulla Piattaforma B.
Nota bene: Osservando l'andamento di alcune esecuzioni con spazio di soluzioni differenti, è possibile notare un andamento anomalo. Alcuni test infatti, nonostante debbano esplorare uno spazio di soluzioni più ampio, hanno tempi di esecuzione inferiori.
Questo fenomeno è motivato dalla disposizione dei valori nella griglia e dall'implementazione dell'algoritmo che esegue le computazioni a partire dalla cella con minor numero di candidati.

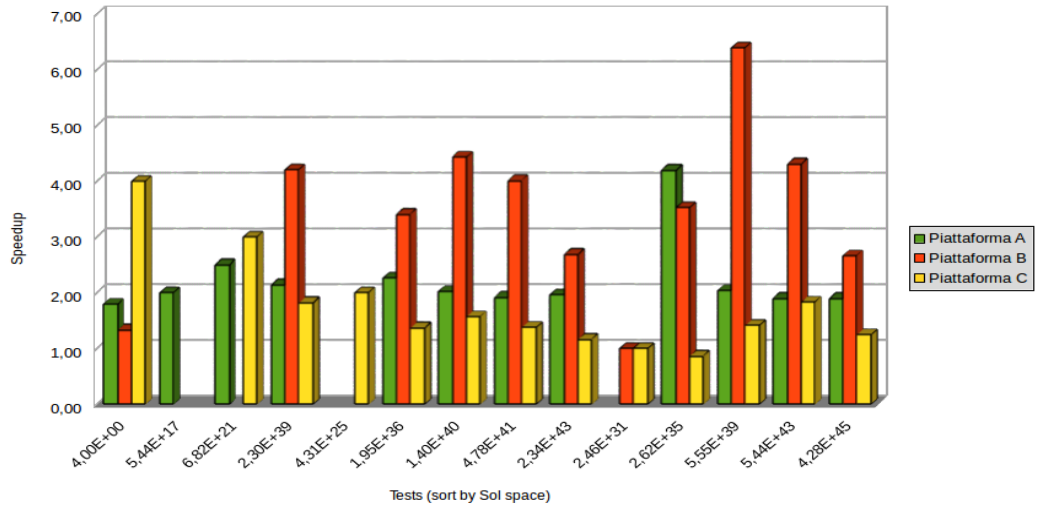

Figure 3: Grafico a colonne degli speedup sulle diverse piattaforme A, B, C.
Alcune colonne sono state omesse in quanto vi erano speedup eccessivamente grandi (infinito) ottenuti in test banali. Il grafico evidenzia come le differenze nel numero di core e nella frequenza di essi abbia un impatto significativo sugli speedup ottenuti.

Istruzioni

1. Posizionarsi mediante terminale nella cartella contenente il file *hw1.tar.gz*;
2. eseguire il comando: *tar -zxvf hw1.tar.gz*;
3. posizionarsi all'interno della cartella *hw1*: *cd hw1*;
4. proseguire alla compilazione dei file *.java* mediante: *javac *.java*
5. eseguire la classe *Main* in una delle seguenti modalità:
   - esecuzione di un singolo file:
     java Main <file_path>
   - esecuzione multipla di tutti i file contenuti nella directory *dir*:
     java Main <directory_path>