Implementazione di algoritmi paralleli per la soluzione di Sudoku

Fantesini Alessandro 1884107
Carlini Lorenzo 1883140

*Il nome completo giapponese del sudoku è **Sū**ji wa **doku**shin ni kagiru, che in italiano vuol dire "sono consentiti solo numeri solitari"*

EXACT COVER (CON DANCING LINKS)

l'algoritmo si basa sull'exact cover e usa i dancing links

Idea dell'algoritmo

Viene costruito lo "scheletro" di una matrice, ovvero:
- per l'indice di ogni colonna abbiamo la struct constraint che indica quali regole deve seguire un elemento della matrice per stare al suo interno
- per l'indice di ogni riga abbiamo la struct possible_value che indica un possibile valore in una determinata casella del sudoku.

i valori all'interno della matrice sono collegati tra di loro tramite puntatori, in questo modo possiamo gestire più facilmente eliminazione e inserimento
a = elemento
R[a] = elemento a destra di a
L[a] = elemento a sinistra di a
per eliminare un elemento -> R[L[a]] = R[a] AND L[R[a]] = L[a]
per reinserirlo -> R[L[a]] = a AND L[R[a]] = a
elemento della matrice = struct matrix_value
la quantità di elementi all'interno della matrice varia in base al sudoku (più numeri sono già inseriti, meno elementi avrò nella matrice)

Una volta creato lo "scheletro" e riempito di valori:

- cerchiamo se esiste una colonna con elementi al suo interno minori o uguali a 1
  - se la colonna è vuota, devo tornare indietro e ricostruire la matrice fino ad uno step "split" e scegliere l'altra possibile soluzione, ma se torno allo step 0 non ho soluzione
  - se la colonna ha un elemento, lo aggiungo alla soluzione
- se la colonna non esiste, cerchiamo quella con il numero di elementi minore
  - scelgo un elemento della colonna
  - lo step viene definito "split"

quando non ho più colonne, l'algoritmo termina correttamente

tempo medio di esecuzione = 0.000523 secondi

NOTE

-quando aggiungo un elemento alla soluzione significa che:

elimino la sua colonna

elimino tutte le colonne che hanno un elemento della stessa riga di quello aggiunto alla soluzione

elimino ogni riga che ha un elemento dentro una colonna eliminata

-sono in "split" quando ad un passo possono scegliere tra 2 o più soluzioni

SCELTE PROGETTUALI

È stato scelto di cercare prima una colonna con elementi al suo interno minori o uguali a 1 e, in caso negativo, trovare quella con il minor numero di elementi perché:

- una colonna con un elemento solo è la soluzione preferibile da aggiungere perché non crea uno "split" (e quindi una possibilità di tornare indietro),
- nell'80-90% dei casi abbiamo una casella con un elemento solo
- trovare subito una colonna senza elementi ci permette di tornare subito indietro allo "split" (o a terminare l'algoritmo senza soluzione)

Ci siamo affidati ad una libreria esterna per la generazione di sudoku randomica

Per provare l'algoritmo, fare riferimento **compile.sh**

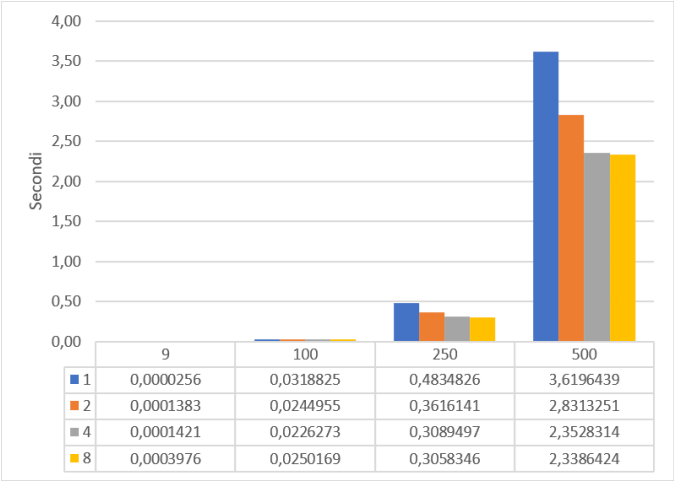
OpenMP

La versione openMP, per la legge di Amdahl, non ci permette di avere speedup effettivi a causa del dataset troppo piccolo che il sudoku 9x9 ci offre.
Sono stati quindi realizzati degli eseguibili che isolano parti di codice dell'algoritmo principale sulle quali era stato applicato openMP ampliandone il dataset e mostrando i vantaggi che la parallelizzazione porta.

Tempo medio di esecuzione con n threads = 0.004552 secondi
(contro i 0.000523 secondi della versione seriale)

ATTENZIONE!
Ogni dimensione dei seguenti grafici è uguale alla dimensione che il lato del sudoku dovrebbe avere

parallel.c
algoritmo che permette di creare gli indici delle righe e delle colonne della matrice

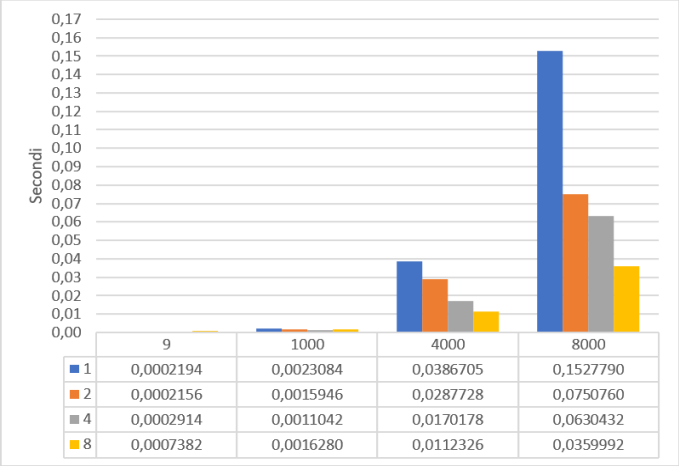


Thread/Size	9	100	250	500
1	0,0000256	0,0318825	0,4834826	3,6196439
2	0,0001383	0,0244955	0,3616141	2,8313251
4	0,0001421	0,0226273	0,3089497	2,3528314
8	0,0003976	0,0250169	0,3058346	2,3386424

Input	S/E	1	2	4	8
9	S	1.0	0,185105	0,180155	0,064386
	E	1.0	0,092552	0,045039	0,008048
100	S	1.0	1,301566	1,409028	1,274438
	E	1.0	0,650783	0,352257	0,159305
250	S	1.0	1,337013	1,564923	1,580863
	E	1.0	0,668506	0,391231	0,197608
	S	1.0	1,278428	1,53842	1,547754
500	E	1.0	0,639214	0,384605	0,193469

reduction.c

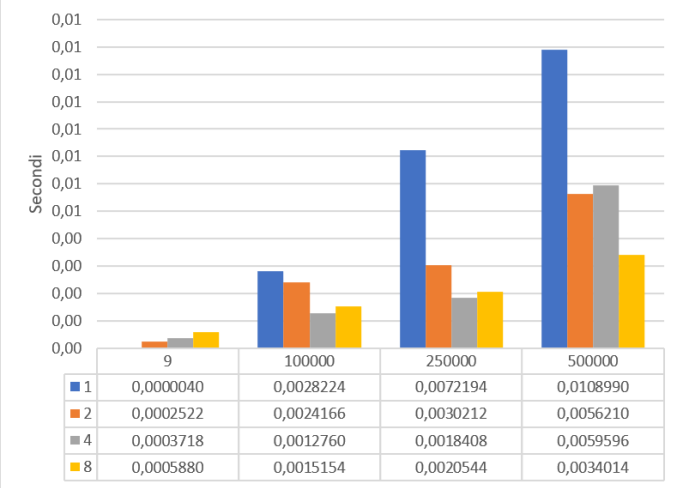
algoritmo di riduzione che permette di calcolare la dimensione che avrà la matrice a partire dal sudoku



Thread/Size	9	1000	4000	8000
1	0,0002194	0,0023084	0,0386705	0,1527790
2	0,0002156	0,0015946	0,0287728	0,0750760
4	0,0002914	0,0011042	0,0170178	0,0630432
8	0,0007382	0,0016280	0,0112326	0,0359992

Input	S/E	1	2	4	8
9	S	1.0	1,017625	0,752917	0,297209
	E	1.0	0,508813	0,188229	0,037151
1000	S	1.0	1,447636	2,090563	1,417936
	E	1.0	0,723818	0,522641	0,177242
4000	S	1.0	1,343995	2,272356	3,442702
	E	1.0	0,671998	0,568089	0,430338
	S	1.0	2,034991	2,423402	4,243955
8000	E	1.0	1,017496	0,60585	0,530494

smallest_col_finder.c
algoritmo che ricerca la colonna più piccola simulando un insieme di liste linkate



Thread/Size	9	100000	250000	500000
1	0,0000040	0,0028224	0,0072194	0,0108990
2	0,0002522	0,0024166	0,0030212	0,0056210
4	0,0003718	0,0012760	0,0018408	0,0059596
8	0,0005880	0,0015154	0,0020544	0,0034014

Input	S/E	1	2	4	8
9	S	1.0	0,01586	0,010758	0,006803
	E	1.0	0,00793	0,00269	0,00085
100000	S	1.0	1,167922	2,211912	1,862479
	E	1.0	0,583961	0,552978	0,23281
250000	S	1.0	2,38958	3,921882	3,514116
	E	1.0	1,19479	0,98047	0,439265
	S	1.0	1,938979	1,828814	3,204269
500000	E	1.0	0,969489	0,457204	0,400534

Simulated Annealing (SA)

Idea dell'algoritmo

Il Simulated Annealing è un algoritmo di ricerca che, partendo da un valore iniziale, è in grado di determinare valori successivi attraverso un processo randomico iterativo.
Questa sua caratteristica lo rende particolarmente adatto per funzioni con obbiettivi non-lineari, dove altri algoritmi di ricerca non sono particolarmente idonei.
L'algoritmo ha una "temperatura" che progressivamente diminuisce e, col diminuire del valore della temperatura, diminuisce il range di accettazione del prossimo valore (che inizialmente può essere peggiore di quello precedente ma progressivamente dovrà avere sempre di più un valore migliore) fino a quando non è più possibile ottenere nuove soluzioni per il valore precedente. A questo punto l'algoritmo si ripete aumentando la "temperatura" e ripartendo dall'ultimo valore trovato.

Il SA viene utilizzato per queste sue proprietà per la soluzione di un Sudoku.
(Basato sul seguente paper: https://link.springer.com/content/pdf/10.1007/s10732-007-9012-8.pdf)

Una volta ottenuti i valori di input di un Sudoku di dimensione LatoSudoku x LatoSudoku (i numeri statici), la restante parte del Sudoku sará riempita casualmente, per ogni settore del Sudoku (di dimensione sqr(LatoSudoku) x sqr(LatoSudoku)) con numeri "mobili" in modo tale che in ogni settore del Sudoku ci siano tutti i valori compresi nell'insieme {0,..,LatoSudoku}.
Si calcolerá a questo punto, per ogni riga e colonna del Sudoku, il numero di "ripetizioni" dei valori al loro interno (i.e. {1,1,3,5,6} contiene due 1 ed avrá punteggio 1) ed un punteggio totale del Sudoku, cioé la somma di ogni numero di "ripetizioni" per ogni riga e colonna, ottenendo quindi un Initial State per il nostro algoritmo di Annealing.
Scegliamo quindi una Starting Temperature calcolando la Standard Deviation del punteggio totale di ANNEAL_TEMP_SAMPLE possibili stati iniziali di questo Sudoku.
Si calcola quindi il numero di iterazioni per ogni temperatura, che sará LatoSudoku^2, cioé circa il numero di "numeri mobili" nel Sudoku (Homogenous Simulated Annealing).
Il cooling rate sará di 0.99 (scelto sperimentalmente, diminuendolo si avrá un algoritmo piú veloce ma con meno probabilitá di trovare una soluzione).

Viene creata una copia temperanea del Sudoku.
Verrá quindi, per il numero massimo di iterazioni,scelto casualmente uno dei LatoSudoku settori del Sudoku, al cui interno verrano a loro volta scelti due "numeri mobili" di cui invertire le posizioni; fatto ció viene ricalcolato il numero di ripetizioni all'interno del Sudoku ottenuto: se (rand()%sqr(LatoSudoku)) < (exp(nuovo_punteggio - punteggio precedente)/temperatura) allora questo risultante Sudoku diventerá il nuovo Sudoku temperaneo, altrimenti resterá al valore attuale.
Se il Sudoku temporaneo ottenuto dopo queste iterazioni risulta avere un punteggio migliore di quello precedente questo verrá aggiornato col suo valore.
Questo processo si ripeterá finché non si otterrá un Sudoku con punteggio totale = 0.

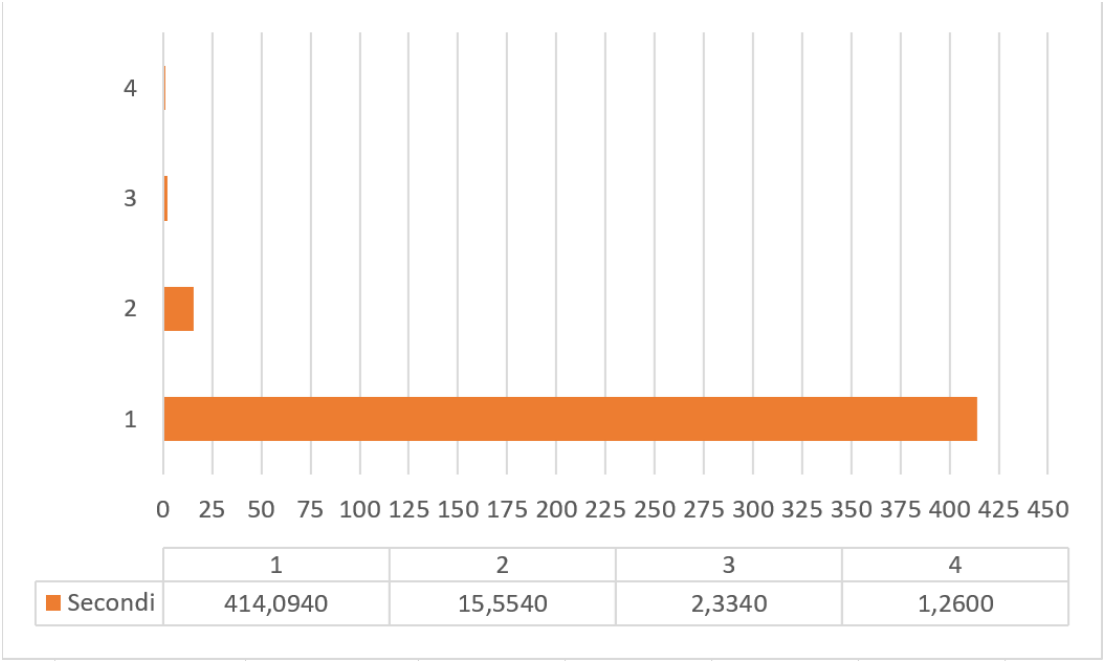
pThreads

La versione Multicore di questo algoritmo opera sullo stesso principio di quella Singlecore, rendendo peró concorrente tra piú Threads la ricerca di Sudoku con punteggio totale migliore di quello attuale.
Viene adoperato pthreads per, una volta inizializzati i dati, avviare piú Threads che avranno, oltre ai parametri discussi prima, accesso via puntatore all'array del Sudoku principale e ad una key, che verrá utilizzata per controllare l'accesso in scrittura a questa struttura (approccio Lettore/Scrittore).
Una volta che un Thread avrá trovato un Sudoku con punteggio totale minore, gli altri al controllo successivo sul Sudoku principale, aggiorneranno i propri array contenenenti i loro Sudoku temporanei con i suoi valori.

Test e note

Con l'aumento della complessitá ed il numero di "numeri statici" all'interno del Sudoku i tempi dell'algoritmo Single-Thread aumentano esponenzialmente rispetto a quelli Multi-Thread.
Per i seguenti test e tempi sono stati utilizzati Sudoku 3X3 di complessitá alta ottenuti con il generatore incluso.

Threads/Size	9
1	414,0940
2	15,5540
4	2,3340
8	1,2600



Input	S/E	1	2	4	8
9	S	1.0	26,62299	177,4182	328,646
	E	1.0	13,3115	44,35454	41,08075