1 - Approccio algoritmico

Federico Bednarski, 1589903

Il programma usa un algoritmo fork-join iterativo ed uno divide-et-impera. Non sono state usate librerie esterne. L'approccio risolutivo e' inconsueto ma molto efficiente. L'unita' base dell'algoritmo non e' la cella ma bensi' la sottogriglia 3x3 del sudoku che chiamero' d'ora in poi quadrato. Altre parole e concetti chiave sono: **riga-quadrato e colonna-quadrato**, ovvero le righe o le colonne fatte da 3 quadrati 3x3 in una griglia sudoku. La **compatibilita' tra 2 quadrati**: 2 quadrati si dicono compatibili per riga se possono stare nella stessa riga-quadrato di un sudoku (nessun numero in comune nella stessa riga). Quando parlo delle **permutazioni di un quadrato**, intendo l'insieme di tutti i quadrati distinti senza numeri ripetuti (ovvero i quadrati legali del sudoku). Quando parlero' di **complessita'** cubica intendo n*m*o anziche' n^3 (cio' vale anche per le altre complessita'), inoltre essendo le complessita' date da cicli annidati, nel caso medio sono meno complesse per via dei cortocircuiti (break/continue).
Il programma e' complesso e tale complessita' non risiede negli algoritmi di parallellizzazione, se il lettore non fosse interessato nelle ragioni delle scelte e nel funzionamento delle ottimizzazioni puo' saltare tutti i punti in "Dettagli dell'implementazione e delle ottimizzazioni" tranne il primo punto "Gli algoritmi paralleli". Il programma calcola le soluzioni nella seguente maniera:

1. Calcolo tutte le permutazioni possibili (tutti i quadrati riempiti legalmente) di un quadrato vuoto (PQV).
2. Leggo il file con il sudoku iniziale e converto il quadrati in un formato altamente ottimizzato per i confronti tra quadrati (vedere "Dettagli dell'implementazione e delle ottimizzazioni").
3. Genero una copia di PQV per ognuno dei 9 quadrati filtrandone le permutazioni non compatibili con il quadrato. Ottengo quindi 9 liste, ognuna delle quali corrisponde a tutte le permutazioni di un quadrato.
4. Filtro le permutazioni di ogni quadrato in base ai quadrati nella stessa riga-quadrato e colonna-quadrato (ogni quadrato ha esattamente 4 vicini). Ora le permutazioni di un quadrato sono compatibili con i vicini.
5. Trovo le permutazioni di ogni riga-quadrato con un semplice algoritmo con caso peggiore cubico, ovvero 3 cicli for annidati dove il primo ciclo itera le permutazioni del primo quadrato di riga, il secondo il secondo e il terzo il terzo. Sfrutto nuovamente i confronti tra quadrati (in questo caso verifico la compatibilita' di riga) per trovare le terne che compongono una riga-quadrato legale. Alla fine di questa fase ho per ognuna delle 3 righe-quadrato tutte le permutazioni legali. Fino ad ora tutto e' stato computato serialmente poiche' anche nei casi piu' impegnativi il tempo dall'inizio del programma alla fine di questa fase non supera i 94 millisecondi (contro i 30 secondi dell'intero processo). I promissimi passi saranno molto piu' computazionalmente impegnativi e sono infatti parallelizzati.
6. Come abbiamo "fuso" i quadrati in righe ora "fondiamo" le righe tra loro. L'algoritmo e' nel caso migliore n^6 (vedere "Dettagli dell'implementazione e delle ottimizzazioni" per dettagli), ma l'approccio e' sostanzialmente lo stesso. In questa "fusione", anziche' confrontare le righe, richiedamo che 2 permutazioni di riga-quadrato adicenti abbiamo tutti e tre i propri quadrati compatibili per colonna. Fondiamo le permutazioni della prima riga con quelle della seconda e otteniamo permutazioni composte da due righe (6*9 celle). Le permutazioni su cui itera il ciclo for piu' esterno vengono divise tra i thread, i thread vengono spawnati dal thread principale che poi si unisce nella computazione. Quando dico che sono stati usati 4 thread intendo 3 + main thread. Dopo che il join tra thread principale ha computato e ha eseguito il join con i figli unisco le permutazioni risultanti in una un'unica lista. Infine fondiamo le permutazioni 6*9 con quelle della 3 riga parallelizzando nella stessa maniera iterativa ma questa volta ogni thread eseguira' un algoritmo fork-join divide-et-impera (stavolta la complessita' e' log9(n)*n^5), la ragione e' indicata in "Dettagli dell'implementazione e delle ottimizzazioni". Le permutazioni risultanti sono esattamente tutte le possibili soluzioni valide del sudoku!

2 - Dettagli dell'implementazione e delle ottimizzazioni

2.1 - Gli algoritmi paralleli

Il *primo algoritmo* parallelo e' di tipo iterativo. La ragione dell'utilizzo di questa tecnica sta nel fatto che l'output dell'algoritmo sia una lista molto grande, se generassimo un "albero di task parallele" creeremo diverse liste intermedie rendendo il processo molto piu' memory-bound e spenderemmo molto tempo a fondere le liste. Un aspetto positivo e' anche la facilita' di ripartire il ciclo for esterno tra i thread. Il numero di thread generati (compreso il main thread) e' pari al numero di processori, questo e' il numero piu' efficiente per l'algoritmo sul mio computer, tuttavia, essendo pochi (4), l'algoritmo e' suscettibile ad anomalie di scheduling.

Il *secondo algoritmo* ha il compito di esplorare un albero binario, per cui l'approccio divide-et-impera e' facilmente applicabile. Mentre un ramo viene eseguito in parallelo, l'altro e' eseguito sul thread corrente, e a loro volta ricorsivamente, il procedimento e' ripetuto per i figli dei 2 rami. Arrivati alle foglie vengono risolti 6 cicli for annidati per trovare le terne (righe) compatibili. Durante il percorrimento dell'albero le liste di permutazioni vengono ridotte diminuendo il tempo di risoluzione dei cicli. Le riduzioni vengono effetuate con un ciclo for. Il ciclo for eseguito alla radice e' parallelizzato cosi'da iniziare piu' alberi di thread e liste piu' piccole. Il secondo algoritmo non ha il cutoff sul fork in ricorsione (sperimentalmente ho verificato che il cutoff degradi le prestazioni).

Ho anche verificato che: i thread del primo livello di ogni algoritmo ci mettono lo stesso tempo di computazione, dunque il lavoro e' ben distribuito; i thread creati iterativamente richiedono meno di un millisecondo per il fork, mentre il join non supera i 100 millisecondi nei test piu' lunghi (12 secondi per l'intero programma); infine tutti i processori sono usati al 100%.

2.2 - La rappresentazione e il confronto tra quadrati

Le ottime performance del programma si basano sul confronto efficiente tra quadrati. I quadrati (quando non sono fusi) sono memorizzati con 2 int, uno rappresenta quali numeri sono in quali righe del quadrato e l'altro quali numeri sono in quali colonne. Ogni int e' usato come un bitfield: l'int e' trattato come se fosse un array booleano dove i bit sono i valori booleani (1 rappresenta true, 0 false). Prendiamo come esempio il bitfield che rappresenta le 3 righe del quadrato (per la colonna il concetto e' simmetrico). I primi 9 bit rappresentano quali tra i 9 numeri del sudoku sono nella prima riga del quadrato 3x3: se l'i-esimo bit e' a 1 allora il numero 'i' e' nella riga (se i bit sono 010110000 allora nella riga vi e' il 2, il 4 e il 5). Il 10imo fino al 18esimo bit viene usato per i numeri nella seconda riga e i bit dal 18esimo al 27esimo per rappresentare la terza riga (usiamo 27 dei 32 bit dell'int). Due quadrati sono compatibili per riga se non hanno numeri in comune nella stessa riga, se usiamo i bitfield equivale a richiedere che non abbiano un bit a 1 nella stessa posizione. Per verificarlo basta calcolare (b1 & b2) == 0, dove b1 e b2 sono bitfields di riga. Se il bitwise-and tra i due da 0 vuole dire che non ci sono bit a 1 nella stessa posizione. Con sole due operazioni native possiamo verificare se 2 quadrati siano compatibili.

2.3 - La rappresentazione di quadrati fusi

Quando fondiamo due quadrati sulla stessa riga-quadrato (riga di quadrati 3x3) usiamo 3 bitfield per ogni permutazione: 2 bitfield per le colonne, sono quelli dei due quadrati originari, e uno per le righe, ottenuto facendo il bitwise-or tra i due bitfield di riga. Quando fondiamo il terzo quadrato di riga-quadrato ottenendo la riga-quadrato intera, teniamo solo i 3 bitfield di colonna perche' non avremmo piu' bisogno di verificare le compatibilita' per riga. Fondere due righe quadrato ci lascia sempre con 3 bitfield di colonna che sono il bitwise-or tra i bitfield delle due righe-quadrato (primo in bitwise-or col primo, scondo col secondo, terzo col terzo).

2.4 - Fusione con l'ultima riga

Quando fondo le soluzioni trovate con l'ultima riga, anziche' salvarle in una lista aumento un contatore. Il contatore finale avra' il numero di soluzioni del sudoku. L'uso del contatore aumenta di molto le prestazioni.

2.5 - La rappresentazione delle permutazioni

Abbiamo gia' detto che per rappresentare quadrati o righe-quadrato usiamo piu' bitfield per ogni elemento. Ebbene molte di queste stringhe di bitfields condividono i primi bitfields, per cui possiamo raggrupparli in alberi. La radice di un albero corrisponde al primo bitfield di tutte le permutazioni nell'albero, e se le permutazioni sono lunghe 3 bitfield ogni figlio della radice puo' raccogliere piu' permutazioni con il secondo bitfield in comune. Questa rappresentazione migliora esponenzialmente le prestazioni computazionali e riduce altrettanto gli sprechi di memoria.

2.6 - Ottimizzazione del ciclo in fusione delle righe

Nella fusione tra righe-quadrato (II Algoritmo Parallelo) le permutazioni di una delle righe-quadrato vengono ordinate in un albero. L'ordinamento avviene in base ai bit del primo bitfield e distribuisce le permutazioni tra le foglie dell'albero (non mi soffermo sui dettagli dell'ordinamento per non complicare ulteriormente). Questa ottimizzazione riduce la complessita' da n^6 a log(n)*n^5 e migliora di circa il 30% i tempi seriali dei suoduku con molte soluzioni (test 1/2 'd' ed 'e'), ma non migliora altrettanto bene i tempi in parallelo (probabilmente e' molto memory-bound). Riducendo il tempo seriale piu' che quello parallelo lo speedup peggiora. Senza ottimizzazione lo speedup e' intorno a 3, con l'ottimizzazione si aggira intorno a 2,6.

3 - Caratteristiche del computer

Processsore	AMD A10-7300 Radeon R6, 10 Compute Cores 4C + 6G 1.90 GHz, 4 processori
RAM	8 GB (6.95 GB usabili)
Sistema operativo	Windows 10 64bit

4 - Risultati

4.1 - Soluzioni, Spazio delle soluzioni, Fattore di Riempimento dei Test

Le soluzioni, lo spazio iniziale delle soluzioni e il fattore di riempimento dei sudoku testati.

Test	Soluzioni	Spazio Soluzioni	Riempimento
test1_a.txt	1	1.99946164690944E34	34.5%
test1_b.txt	4715	5.925444579849063E43	27.1%
test1_c.txt	132271	2.1511212563561017E47	24.6%
test1_d.txt	587264	8.78173741694815E48	23.4%
test1_e.txt	3151964	2.5301941845711E50	22.2%
test2_a.txt	1	1.5584944324853144E41	28.3%
test2_b.txt	276	1.2327934475713917E44	25.9%
test2_c.txt	32128	6.8835880203395215E47	23.4%
test2_d.txt	1014785	2.3824875376520945E51	20.9%
test2_e.txt	7388360	2.3824875376520945E51	19.7%

Dai risultati in tabella possiamo notare come il numero si soluzioni sia proporzionale allo spazio iniziale delle soluzioni e che questi siano inversamente proporzionali al fattore di riempimento.

4.2 - Tempi delle Fasi del Programma

Queste sono le descrizioni delle diverse fasi:

- **Fase 1:** leggo il file con il sudoku, costruisco tutte le permutazioni del quadrato vuoto e calcolo lo spazio delle soluzioni e il fattore di riempimento.

- **Fase 2:** trovo tutte le permutazioni delle 3 righe-quadrato.
- **Fase 3:** (I Algoritmo Parallelo) fondo le prime 2 righe-quadrato in un unica soluzione (tutte le permutazioni della parte 6*9 del sudoku).
- **Fase 4:** (II Algoritmo Parallelo) conto le soluzioni costruibili fondendo la parte superiore 6*9 con l'ultima riga-quadrato. Il risultato e' il numero di sudoku possibili con la configurazione di partenza.

I tempi in millisecondi delle diverse fasi del programma in parallelo dati dalla media di 10 iterazioni, ordinate per numero di soluzioni

Test	Soluzioni	Fase 1	Fase 2	Fase 3	Fase 4	Tempo Totale
test1_a.txt	1	28 ms	0 ms	0 ms	1 ms	31 ms
test2_a.txt	1	16 ms	0 ms	0 ms	0 ms	16 ms
test2_b.txt	276	22 ms	0 ms	1 ms	0 ms	24 ms
test1_b.txt	4715	33 ms	4 ms	15 ms	35 ms	89 ms
test2_c.txt	32128	23 ms	3 ms	23 ms	36 ms	86 ms
test1_c.txt	132271	23 ms	10 ms	35 ms	97 ms	167 ms
test1_d.txt	587264	21 ms	10 ms	309 ms	539 ms	880 ms
test2_d.txt	1014785	26 ms	25 ms	148 ms	1642 ms	1842 ms
test1_e.txt	3151964	26 ms	49 ms	221 ms	2566 ms	2863 ms
test2_e.txt	7388360	101 ms	63 ms	3387 ms	8725 ms	12278 ms

I tempi della fase 1 e 2 sono seriali, la 3 e la 4 parallele. La fase 1 e' a tempo costante, le fluttuazioni di tempo sono casuali.

I tempi in millisecondi di fase 3 e 4 seriale e parallelo con relativi speedup misurati in 5 iterazioni, ordinate per soluzioni:

Test	Soluzioni	Fase 3 seriale	Fase 3 parallela	Speedup Fase 3	Fase 4 seriale	Fase 4 parallela	Speedup Fase 4
test1_a.txt	1	0 ms	1 ms	--	0 ms	1 ms	--
test2_a.txt	1	0 ms	0 ms	--	0 ms	1 ms	--
test2_b.txt	276	0 ms	1 ms	--	1 ms	0 ms	--
test1_b.txt	4715	10 ms	28 ms	0.37	10 ms	40 ms	0.25
test2_c.txt	32128	44 ms	21 ms	2.03	100 ms	40 ms	2.5
test1_c.txt	132271	52 ms	35 ms	1.47	200 ms	107 ms	1.87
test1_d.txt	587264	504 ms	268 ms	1.88	1244 ms	559 ms	2.22
test2_d.txt	1014785	273 ms	93 ms	2.92	4700 ms	1762 ms	2.66
test1_e.txt	3151964	315 ms	236 ms	1.33	6207 ms	2552 ms	2.43
test2_e.txt	7388360	4700 ms	3815 ms	1.23	24952 ms	9481 ms	2.63

Possiamo vedere come lo speedup della 3a fase (I Algoritmo Parallelo Iterativo) migliori fino al 1.000.000 di soluzioni per poi iniziare a peggiorare, mentre la fase 4 (II Algoritmo Parallelo Divide-Et-Impera). Il problema non e' dovuto all'aumentare dell'input: nel test1_e la fase 3 trova 1.482.772 soluzioni elaborando 3 righe-quadrato con poche migliaia o centinaia di permutazioni, mentre la fase 4 ne trova 3.151.964 dall'unione delle 1.482.772 della fase precedente con poche migliaia della 3a riga-quadrato. La fase 3 deve aggiungere in una lista le soluzioni mentre la 4 deve aggiornare un contatore, probabilmente l'aggiunga di risultati in liste lo rende molto piu' memory bound. Nei test a, b, c lo speedup e' basso perche' ci sono pochi elementi nel ciclo da dividere tra i thread, alcuni sono meno dei thread.

4.3 - Tempi e Speedup

I tempi in millisecondi in seriale e parallelo con relativo speedup (senza iterazioni), ordinate per numero di soluzioni:

Test	Soluzioni	Tempo Seriale	Tempo Parallelo	Speedup
test1_a.txt	1	60 ms	30 ms	2.0
test2_a.txt	1	18 ms	17 ms	1.05
test2_b.txt	276	18 ms	33 ms	0.54
test1_b.txt	4715	105 ms	96 ms	1.09
test2_c.txt	32128	213 ms	77 ms	2.76
test1_c.txt	132271	429 ms	162 ms	2.64
test1_d.txt	587264	2612 ms	916 ms	2.85
test2_d.txt	1014785	5138 ms	1940 ms	2.64
test1_e.txt	3151964	7660 ms	3020 ms	2.53
test2_e.txt	7388360	32989 ms	12033 ms	2.74

Tutti i dati riportati fin'ora vano presi con un grano di sale, i tempi variano da esecuzione ad esecuzione. In particolare alcuni speedup che sono 1.3 saltano a 3.0 e viceversa. Inoltre le tabelle sulle fasi sono state calcolate facendo la media su iterazioni consecutive, questo potrebbe influenzare i risultati (la JVM puo' ottimizzare codice, la memoria e' gia' stata richiesta al sistema operativo ecc.). I tempi sono stati registrati avviando un jar ed eseguendo gli 8 test in un unico processo. Eseguendo un solo test da riga di comando il file .class o il jar, il test1_e.txt ci ha messo 9 secondi anziche' 3.

Conclusione Lo speedup purtroppo non e' perfetto, forse la causa e' nell'avere due algoritmi paralleli in successione anziche' un unico grande algoritmo parallelo e nel fatto che gli algoritmi siano memory-bound (la maggior parte del tempo viene passata a leggere e a scrivere). Pero' sono davvero sodisfato delle prestazioni: i tempi sono molto veloci, tutti i processori vengono usati al 100% e nel test piu' grosso (2e) la memoria usata e' meno di 512MB.

5 - Eseguire il programma

Compilare ed eseguire senza il jar file

Andate nella directory principale con la shell, dovrebbe esserci la directory sudokuhw, il pdf e il jar; e' importante eseguire tutti i comandi da questa directory:
I .java file sono gia' compilati, ma se si vuole li si puo' ricompilare eseguendo il seguente comando:
javac .\sudokuhw*.java
Ora le classi sono compilate, per eseguire il progetto digitate ed eseguite:
java -Xmx512m -cp . sudokuhw/SudokuHW *path/to/sudoku*
Dove "path/to/sudoku" e' il path del file contenente il sudoku.