Homework di Programmazione di Sistemi Multicore

di Valerio Baudo, matricola 1696903

Scelta dell'algoritmo

La risoluzione di un sudoku è classificata come un problema in NP. Questo vuol dire che sebbene la correttezza di una soluzione sia verificabile in tempo polinomiale, non esiste ancora un algoritmo in grado di generare una soluzione in tempo polinomiale – e quindi efficiente.

Recursive Backtracking. Si tratta dell'algoritmo più comunemente utilizzato per risolvere un sudoku. Esso consiste essenzialmente in una Depth-First Search nell'albero di tutte le possibili soluzioni legali del problema, considerando, man mano che si scende, solo i rami che contengono soluzioni legali. Poiché la consegna richiede che venga restituito in output il numero di tutte le possibili soluzioni del sudoku, l'algoritmo va modificato per permettere di contare le soluzioni ogni volta che ne trova una.

Dancing Links. Il problema della risoluzione di un sudoku è riducibile ad un problema di exact-cover, ossia ad un problema che, data una matrice composta di zeri ed uni, come soluzione ha un sottoinsieme delle sue righe in cui in ogni colonna 1 appare solo una volta. Una volta ridotto il sudoku ad un problema di exact-cover è possibile applicarvi l'algoritmo dei Dancing Links.

Vista la facilità di implementazione dell'algoritmo di recursive backtracking, nonché la sua predisposizione per essere parallelizzato, ho optato per implementare questo algoritmo.

L'algoritmo sequenziale

Il metodo che risolve il sudoku sequenzialmente (Figura 1) prende in input le coordinate dalle quali dovrà partire a risolvere il sudoku (quindi è assunto che le celle del sudoku precedenti a quelle coordinate siano già riempite).

```
// solver with backtracking algorithm
int seqSolver(int i, int j){
        int r = 0;

        if (i == DIM) {                         // if we got to the end of the column
                i = 0;                          // go to the first cell of the column
                if (++j == DIM)                 // and then to the next column; if next column is above the limit
                        return 1;               // then we got a new solution
        }

        if(getValue(i, j) != 0)                         // skip filled cells
                return seqSolver( i+1, j);              // recursive call to the next cell

        for(int val = 1; val <= 9; val++) {             // cycle through all possible values
                if(isLegal(i, j, val)){                 // if val is legal
                        setValue(i, j, val);            // then set val as the cell's value
                        r += seqSolver( i+1, j);        // add additional solutions
                }
        }
        reset(i, j);            // and reset on backtrack

        return r;

}
```

Figura 1: Metodo che risolve il sudoku sequenzialmente, con l'algoritmo di recursive backtracking

Come prima cosa, controlla se l'indice di riga sia corretto e, se non lo è, vuol dire che o bisogna procedere alla colonna successiva, o, se non esiste una colonna successiva, abbiamo trovato una soluzione.
Dopo ciò, il metodo controlla se la cella su cui si trova in questo momento è piena; nel caso lo sia, fa ripartire la ricorsione andando avanti di un indice.
Una volta che l'indice è giusto e abbiamo stabilito che ci troviamo in una cella vuota, possiamo entrare nel ciclo *for* principale. In esso cicliamo per tutti i valori inseribili in una qualunque cella (quindi i numeri da 1 a 9) e, se si tratta di una inserzione legale (ossia che rispetta le regole del sudoku), allora inseriamo tale valore e sommiamo alla variabile intera *r* il risultato della ricorsione che parte dall'indice successivo.
Ciclati tutti i numeri, resettiamo la cella (quindi la svuotiamo) e ritorniamo il valore di *r* (la cui funzione è memorizzare il numero di soluzioni presenti nel suo sotto-albero di ricorsione) che abbiamo calcolato fino ad ora.

Essendo sequenziale, questo algoritmo andrà ad utilizzare un solo core nella nostra CPU, non sfruttando quindi il pieno potenziale delle moderne CPU a più core.

L'algoritmo parallelo e le ottimizzazioni implementatevi

Per implementare l'algoritmo parallelo (Figura 2) sono partito dall'idea che c'è dietro algoritmo sequenziale, ossia quella di un DFS ricorsivo. Utilizzando il paradigma Fork-Join, come consigliato nella consegna, mi è subito venuto in mente di basare il tutto su *RecursiveTask*, che serve, essenzialmente, per implementare algoritmi ricorsivi che alla fine devono ritornare un qualche valore.
Il metodo compute, ossia dove si trova l'algoritmo parallelo vero e proprio, inizia subito con una ottimizzazione, detta "cutoff sequenziale". Nel caso in cui la complessità del sudoku sia inferiore a quella del cutoff, il sudoku viene eseguito sequenzialmente. Questo permette di tagliare più del 99% dei thread (ossia le foglie dell'albero di ricorsione) in quanto queste verranno invece eseguite sequenzialmente.
Successivamente andiamo ad eseguire l'algoritmo nella stessa identica maniera a quello sequenziale, con l'unica differenza che invece di fare una chiamata ricorsiva quando troviamo un valore legale, andiamo a creare un task per ogni possibile valore legale.
Alla fine andiamo quindi a fare la *fork()* di tutti i task generati in precedenza. Mentre i task stanno girando, applichiamo anche l'ottimizzazione del cosiddetto "dimezzamento dei thread" (o dei "task" in questo caso), ossia andiamo ad eseguire uno dei task sul task padre, permettendoci quindi di diminuire il numero di task creati (e diminuendo quindi l'overhead che genererebbe la creazione di un task aggiuntivo).
Viene infine ritornato *r*, che è la somma di tutti i *RecursiveTask* che hanno completato l'esecuzione.

```java
@Override
protected Integer compute() {
        if (searchSpace.compareTo(cutoff) < 0)
        //if (soodoku.computeEmptyCells() < cutoff.intValue())
                return soodoku.seqSolver(i, j);
        int r = 0;

        List<ParallelSolver> tasks = new ArrayList<>();

        if (i == Sudoku.DIM) {
                i = 0;
                if (++j == Sudoku.DIM)
                        return 1;
        }
        if (this.soodoku.getValue(i, j) != 0) {
                ++i;
                return compute();
        }
        for (int val = 1; val <= Sudoku.DIM; ++val) {
                if (soodoku.isLegal(i, j, val)) {
                        soodoku.setValue(i, j, val);
                        ParallelSolver recursiveTask =
                                        new ParallelSolver (
                                                        i + 1, j,
                                                cloneSudoku(soodoku),
                                                soodoku.computeSearchSpace(), cutoff
                                        );
                        tasks.add(recursiveTask);
                }
        }

        // Removing a task from the arraylist to run it on the main thread
        // shortens the wallclock time by ~10%
        if (tasks.size() > 0) {
                ParallelSolver threadHalving = tasks.remove( 0);

                for (ParallelSolver t : tasks)
                        t.fork();

                r += threadHalving.compute();

                for (ParallelSolver t : tasks)
                        r += t.join();
        }
        return r;
}
```

Figura 2: Metodo compute(), che esegue l'algoritmo parallelo

Specifiche delle piattaforme sulle quali sono stati effettuati i test

I test sono stati eseguiti su 3 macchine differenti, per sperimentare varie configurazioni.

Fisso: OS: Linux 64bit versione 4.20.0
CPU: Intel i5 8600k con 6 core a 4.5GHz
RAM: 16GB DDR4 a 3.0GHz

Portatile: OS: Linux 64bit versione 4.20.0
CPU: Intel i5 7300HQ con 4 core a 4.5GHz
RAM: 8GB DDR4 a 2.4GHz

Raspberry Pi 3b: OS: Linux 32bit versione 4.9.35 per ARMv8
CPU: ARM Cortex-A53 con 4 core a 1.2GHz
RAM: 1GB LPDDR2 a 900MHz

Risultati sperimentali ottenuti

Ogni tempo riportato in questo paragrafo è frutto di una media di almeno 5 esecuzioni con gli stessi parametri.
Tutti i tempi sono riportati in millisecondi.

Come prima prova ho fatto girare il test “test1_a.txt” (che ha una sola soluzione) su tutt’e tre le macchine, ottenendo i seguenti risultati:

test1_a.txt	Fisso	Portatile	Raspberry Pi 3
T Parallelo	3	4	19
T Sequenziale	1	1	5
Speedup	0.333	0.250	0.263

Chiaramente non è possibile apprezzare nessun tipo di speedup in questo caso, poiché il sudoku non offre la possibilità di esplorare un albero di ricorsione abbastanza vasto e l’overhead dell’algoritmo parallelo grava troppo sulla performance.

Ho quindi cercato un test più interessante per misurare effettivamente le potenzialità dell’algoritmo (utilizzato con il cutoff di default, ossia 10^{30}). Un buon test con cui ho avuto risultati affascinanti è stato il test “test1_d.txt”:

test1_d.txt	Fisso	Portatile	Raspberry Pi 3
T Parallelo	5527	12444	501751
T Sequenziale	31263	36772	1658373
Speedup	5.656	2.954	3.305

Qui si può veramente apprezzare uno speedup significativo, tutto grazie a ForkJoinPool che riesce a bilanciare il carico di lavoro dei task su tutti i processori disponibili, sfruttando cosi al meglio le macchine su cui è stato effettuato il test.

Ho anche sperimentato con diversi valori per il cutoff sequenziale (basato sullo spazio delle soluzioni) sul Fisso, utilizzando nuovamente il test “test1_d.txt”:

test1_d.txt	0	1	10^3	10^6	10^{12}	10^{20}	10^{30}	**10^{31}**	10^{40}	10^{41}
T Parallelo	187762	80375	67349	55205	19154	6418	5527	**5450**	6890	17148
Speedup	0.166	0.388	0.464	0.566	1.632	4.871	5.656	**5.736**	4.537	1.823

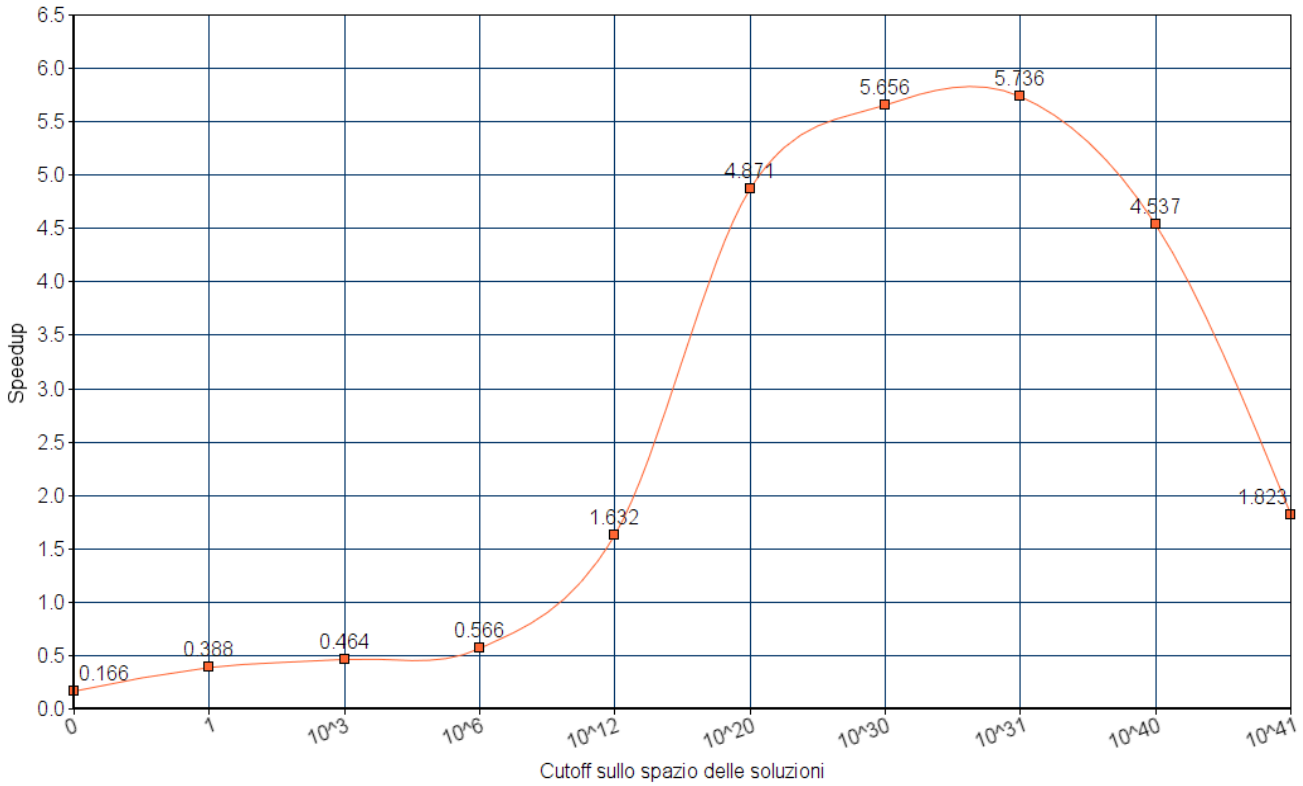

Figura 3: Grafico della relazione tra speedup e cutoff basato sullo spazio delle soluzioni

Poiché questo particolare sudoku ha uno spazio delle soluzioni di circa $4*10^{41}$, qualunque cutoff superiore a $4*10^{41}$ risulterebbe nell'esecuzione seriale dell'algoritmo. Come si può evincere dal grafico (Figura 3), lo speedup migliore viene raggiunto quando il cutoff è 10^{31}, mentre per valori troppo bassi del cutoff lo speedup è minore di 1.

Come sperimentazione finale ho provato ad utilizzare un cutoff differente, ossia uno basato sul numero di celle vuote rimanenti nel sudoku. Sebbene questo tipo di cutoff non ci dica molto su quanto ancora deve andare avanti la ricorsione, essendo più facile da calcolare (O(9*9) contro O(9*9*9) del cutoff basato sullo spazio delle soluzioni) la sua implementazione potrebbe impattare positivamente sulla performance dell'algoritmo:

test1_d.txt	15	30	45	50	**52**	54	56	58	60	62
T Parallelo	69792	25561	5722	5510	**5383**	5558	5541	5943	6836	17029
Speedup	0.447	1.223	5.463	5.673	**5.807**	5.624	5.642	5.260	4.573	1.835

Dai risultati si evince che, sebbene utilizzando questo cutoff si riesca a raggiungere uno speedup più alto, si tratta comunque di un incremento di grandezza trascurabile, quindi posso concludere che i due cutoff, se usati correttamente, sono equivalenti.

Considerazioni finali

Lo speedup è sempre maggiore di 1? Perché?
Lo speedup è maggiore di 1 solo nel caso in cui il sudoku accetti più di una soluzione, perché nel caso in cui è possibile solo una soluzione, lo spazio delle soluzioni da esplorare è troppo più piccolo dell'overhead generato dall'algoritmo parallelo.

Quali istanze richiedono più tempo? Esiste una correlazione tra fattore di riempimento, spazio delle soluzioni e tempo di esecuzione?
Le istanze che richiedono più tempo sono quelle con uno spazio delle soluzioni più grande e, in linea di massima, corrispondono a quelle che hanno più celle vuote all'inizio.

Modalità per compilare ed eseguire il programma

Per compilare il programma bisogna aprire da terminale la directory in cui si trova il file "make.sh"

Quindi bisogna eseguire:

```
$ bash make.sh
```

Per eseguire il programma con il test "test1_d.txt":

```
$ java -jar parallelsudoku.jar test1/test1_d.txt
```

Il programma accetta anche un cutoff come secondo argomento (opzionale). Ad esempio:

```
$ java -jar parallelsudoku.jar test1/test1_d.txt 1E31
```