30/12/2020

Generazione e

Progetto Parallel Algorithm A.A 20/21

Loris Cino

Indice

Introduzione al problema

L'Hitori è un rompicapo giapponese per certi versi simile al Sudoku. Alla sua base ci sono sostanzialmente tre regole:

- Non possono esserci numeri ripetuti in ciascuna riga o colonna.
- Non possono esserci due caselle annerite adiacenti.
- I bianchi devono essere connessi tra di loro. Non possono esserci bianchi isolati.

Lo scopo del gioco è annerire delle caselle per fare in modo che queste tre regole siano rispettate.

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 1: esempio di puzzle Hitori

I numeri ammessi sono da 1 sino alla dimensione del puzzle, nell'esempio 5.

Come usare il programma

Un esempio di utilizzo del programma è il seguente:

mpirun -n 4 ./hitori_par -f <file> -s <dimensione>

Il programma può essere lanciato solo attraverso *mpirun* specificando un numero di processori che sia un quadrato perfetto. Il programma riceve in input due argomenti:

- *-f* (opzionale) attraverso il quale è possibile specificare il percorso del file dal quale prendere la matrice da risolvere. Se non è specificato il programma aspetterà l'inserimento della matrice da *stdin*.
- *-s* (obbligatorio) attraverso il quale è possibile specificare la dimensione del puzzle.

Per maggiori informazioni riguardanti gli altri file presenti nel progetto leggere il file *readme*.

Tecnica usata

Per risolvere il problema è stata utilizzata una tecnica di programmazione chiamata **Backtracking**. Questa tecnica consiste nel rappresentare tutte le possibili soluzioni in un albero e attraversarle fino ad ottenere una soluzione eliminando i rami che non rispettano i vincoli del problema.

```
procedure EXPLORE(node n)
    if REJECT(n) then return
    if COMPLETE(n) then
        OUTPUT(n)
    for n_i : CHILDREN(n) do EXPLORE(n_i)
```

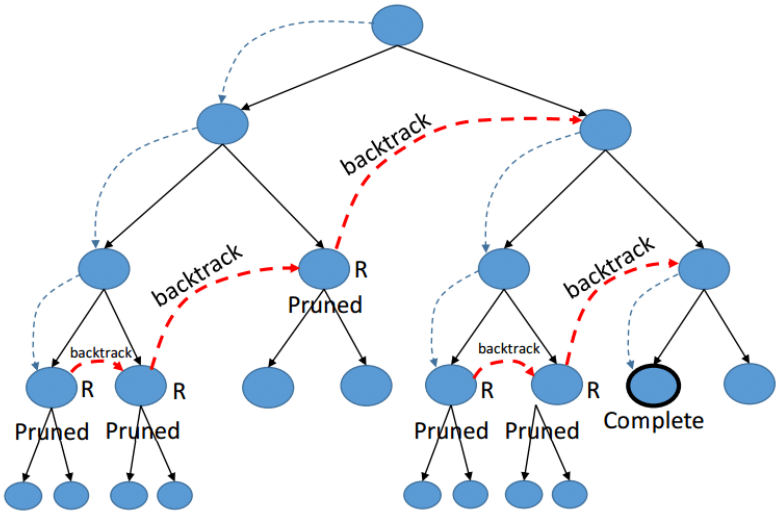


Figura 2: Pseudocodice e grafo backtracking preso da https://www.researchgate.net/figure/Backtracking-algorithm-taken-from-1_fig2_316610194

Questa tecnica è stata utilizzata in entrambi i casi, sia per la verifica che per la generazione di un puzzle risolvibile. Nel primo caso mi limito ad applicare le regole per trovare la soluzione. Nel secondo caso, invece, modifico anche i numeri presenti all'interno del puzzle per trovare un puzzle risolvibile secondo i vincoli del gioco.

Per generare un puzzle Hitori ogni nodo padre avrà 2xN nodi figli, dove N è la dimensione del puzzle in quanto in ogni casella è possibile avere N numeri ed ognuno di questi numeri può essere, nella soluzione, annerito o meno. Mentre nella verifica ogni nodo padre avrà solo due nodi figli in quanto il numero all'interno delle caselle è fissato.

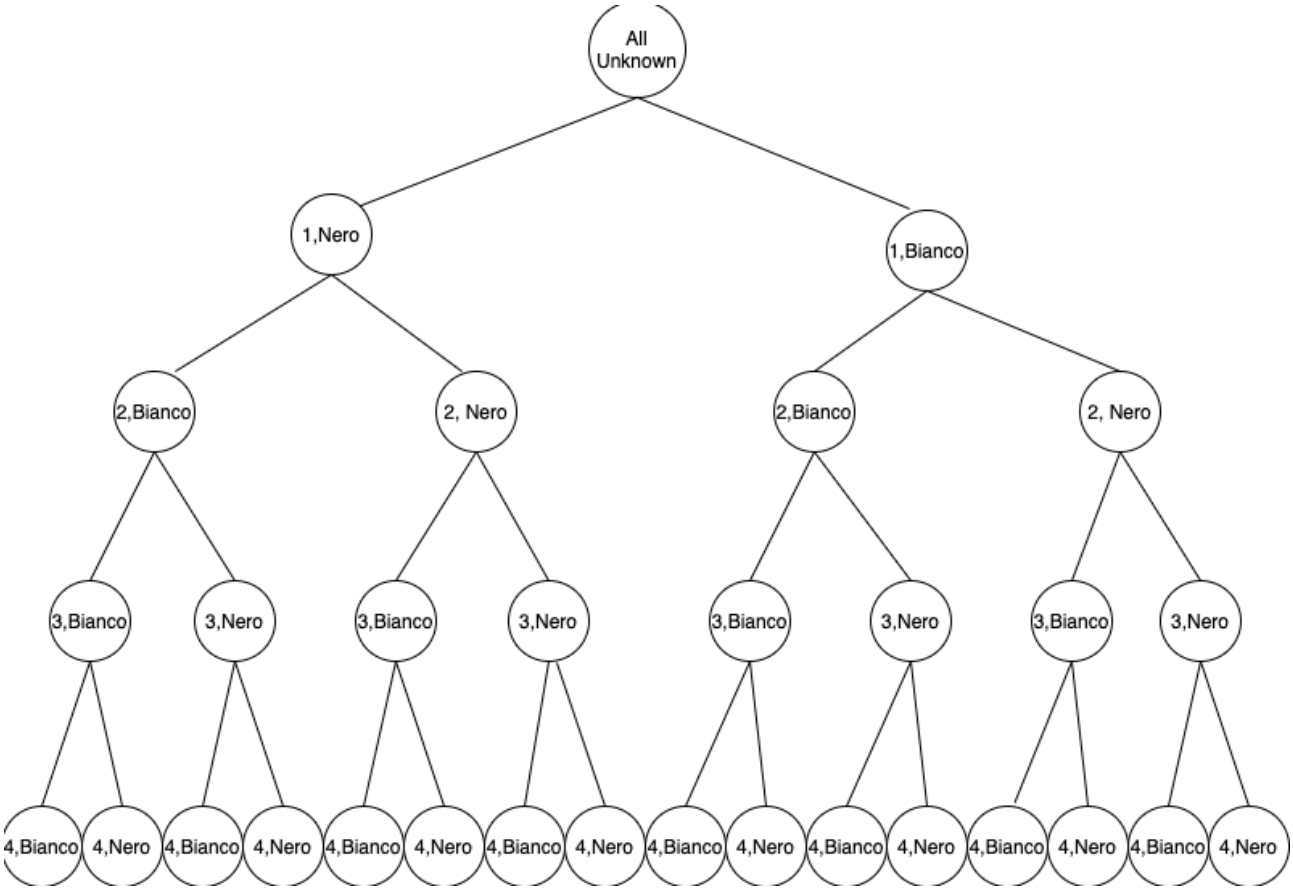


Figura 3: Esempio di un albero per la verifica di un puzzle 2x2

Come si può già dedurre il numero di nodi è enorme in entrambi i casi, è un numero esponenziale rispetto alla grandezza del puzzle. Nel primo caso ci saranno 2^{N^2} nodi generati ma applicando le regole del gioco, la maggior parte dei rami verrà potato e di conseguenza, per quanto scritto nell'articolo *V. N. Rao and V. Kumar, "On the efficiency of parallel backtracking," in IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, pp. 427-437, April 1993, doi: 10.1109/71.219757*, la complessità dell'algoritmo risulterà essere polinomiale.

Per quanto riguarda l'occupazione di memoria questo algoritmo risulta essere molto efficiente, con una implementazione ricorsiva viene impiegata solo una quantità di memoria lineare rispetto alla profondità del grafo. Anche dal punto di vista del calcolo, matrici di dimensione 100x100 ed anche oltre, possono essere generate e anche risolte in pochi secondi, come si vedrà dopo.

Implementazione

Il puzzle è stato modellato come una matrice di *block*, cioè un tipo di dato composto da due valori, uno *short int* e *char*. Il primo contiene il numero della casella corrispondente mentre il secondo conterrà lo stato della cella, cioè *b* se la casella sarà annerita, *w* se la casella è bianca, *u* se non si sa ancora lo stato della casella.

Funzione che verifica l'unicità del numero nelle righe/colonne

Questa funzione si occupa di verificare che non vi siano due bianchi nella stessa riga -o colonna- con lo stesso valore. Inoltre, se vi è una casella bianca con un determinato valore, annerisce tutte le caselle con lo stesso numero che hanno stato sconosciuto.

Funzione che verifica le regole di adiacenza

Semplicemente verifica che non ci siano due neri adiacenti e richiama un'altra funzione che si occupa di verificare che tutti i bianchi siano collegati.

La funzione che verifica che tutte le celle bianche siano connesse è stata risolta immaginando di affrontare il problema di verifica che un grafo -composto solo dalle celle bianche- sia connesso. Si utilizza una ricerca del tipo DFS partendo dalla prima cella bianca che si trova, una volta esplorato l'intero grafo, se non tutti i bianchi sono stati raggiunti da questa esplorazione, l'algoritmo ritornerà -1.

Funzione di verifica delle regole

Questa funzione si occupa di verificare che tutte le regole siano rispettate e itera l'applicazione delle regole fino a quando non vi sono più variazioni nella tabella.

Tutte le funzioni viste fino ad ora ritornano -1 nel caso in cui una o più regole non siano rispettate.

Funzione di backtracking

Semplicemente implementa la tecnica di backtracking e si occupa di tenere il conto di quante caselle con stato sconosciuto sono rimaste, senza doverle contare ogni volta.

Parallelizzazione

Sono stati implementati due approcci, uno che rendeva parallela la ricerca nell'albero tramite backtracking (suggerito dal libro) ed un altro basato sulla suddivisione in blocchi e righe della matrice.

Il suggerimento del libro di testo era quello di effettuare il calcolo in maniera uguale fino ad un certo punto e successivamente dividere i rami tra i vari processi.

L'approccio seguito per la generazione è leggermente diverso in quanto, senza effettuare nessun calcolo, sono state fissati i valori di un certo numero di caselle e i diversi rami assegnati ai processi differiscono nei valori presenti in queste caselle.

1	4	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 4Rami diverso nella generazione

Per il caso della verifica l'approccio seguito è molto simile solo che non si bloccano i valori delle celle ma si blocca il colore della cella: se bianco o nero.

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

1	5	3	1	2
5	4	1	3	4
3	4	3	1	5
4	4	2	3	3
2	1	5	4	4

Figura 5Esempi di rami differenti per la verifica

Per evitare che il programma trovi sempre lo stesso puzzle i valori delle caselle bloccati (o lo stato delle caselle bloccate) sono state codificate in un intero. Ognuno di questi numeri rappresenta un sotto-ramo da analizzare. Tutti questi interi sono stati inseriti in un vettore e vengono estratti in ordine casuale.

Per evitare una distribuzione non uniforme del carico di lavoro dovuto ad una diversa complessità dei rami, si è assegnato un numero molto grande di rami per processo.

Questo metodo purtroppo ha fallito in quanto i rami non conformi alle regole venivano subito potati mentre il ramo con la soluzione richiedeva lo stesso numero di calcoli.

Successivamente si è applicata una parallelizzazione differente basata sulla suddivisione della matrice. Per applicare la regola delle righe si è effettuata una suddivisione a righe della matrice, assegnando circa n/p righe ad ogni processo.

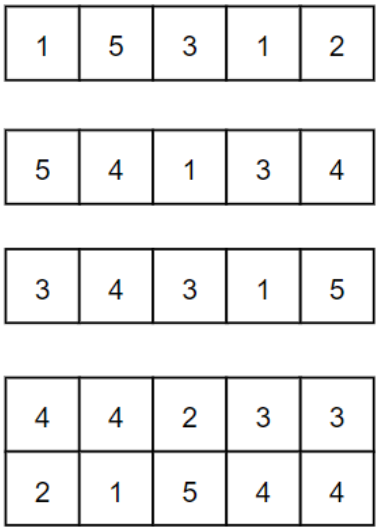


Figura 6Suddivisione esempio per righe

Per quanto riguarda le colonne si è fatta la stessa cosa ma i vari processi, quando vanno a ricevere le proprie colonne, le salvano come se fossero righe, ciò ha permesso di aumentare il **cache hit rate** e di avere performance, come si vedrà, **superlineari**.

1	5	3	4	2

5	4	4	4	1

3	1	3	2	5

1	3	1	3	4
2	4	5	3	4

Figura 7Suddivisione esempio per colonne

Per quanto riguarda la applicazione delle regole di adiacenza si è preferito optare per una suddivisione a blocchi. I blocchi sono stati creati con due righe e due colonne in più, per permettere lo scambio dei bordi.

0	0	0	0
0	1	5	3
0	5	4	1
0	3	4	0

0	0	0	0	0
5	3	1	2	0
4	1	3	4	0
0	3	1	5	0

0	5	4	0
0	3	4	3
0	4	4	2
0	2	1	5
0	0	0	0

0	1	3	4	0
4	3	1	5	0
4	2	3	3	0
1	5	4	4	0
0	0	0	0	0

Figura 8 Suddivisione a blocchi, bordi in blu e bordi in verde

Mentre la funzione che verifica l'adiacenza delle celle bianche è eseguita in sequenziale, solo dal primo processo, in quanto non è stata trovata nessuna parallelizzazione efficiente per risolvere questo problema.

Terminazione (parallel backtracking)

Nel primo caso, quello suggerito dal libro, quando un processo trova una soluzione deve avvertire gli altri processi che possono terminare i loro calcoli in quanto il problema è stato risolto.

Nel caso in cui due processi trovino in tempi molto ravvicinati la soluzione può succedere che un processo, che ha in coda dei messaggi non letti, invochi la funzione *MPI_finalize* e ciò porta ad una terminazione del programma con errori. Per evitare ció è stato utilizzato *l'anello Dijkstra* in una versione leggermente semplificata

Analisi

L'analisi verrà fatta algoritmo per algoritmo poiché il loro funzionamento è completamente separato. Per quanto il programma generale basta tenere conto che, oltre a queste tre funzioni, il resto del codice è eseguito in sequenziale dal processo 0 ma poiché la complessità complessiva delle altre operazioni è costante e quindi possono essere trascurate.

Applicazione regola righe/colonne

In sequenziale, per la aplicazione della regola di unicità la complessità è:

$$\theta(n^3) = n \cdot n \cdot 2(n-1)$$

In quanto ci sono *n* righe/colonne con *n* caselle ciascuna e, per ogni casella, dovranno essere fatti 2(*n* -1) confronti (il due poiché i confronti vanno fatti sia per le righe che per le colonne). Nell'algoritmo in parallelo abbiamo invece:

$$\mathrm{T_p} = \left\lceil \frac{\mathrm{n}}{\mathrm{p}} \right\rceil \cdot n^2$$

Questo poiché è stata scelta una decomposizione a righe. Ogni processo dovrà analizzare al più $\left\lceil \frac{\mathrm{n}}{\mathrm{p}} \right\rceil$ righe/colonne. Poiché, una volta hanno ricevuto la loro parte di puzzle, che i vari processi possono effettuare i calcoli in maniera indipendente, l'algoritmo risulta essere **embarrassingly parallel.**

Dalle formule scritte è facile che, poiché l'overhead è nullo, il programma sarà altamente scalabile.

Applicazione regola di adiacenza

Per applicare questa regola in sequenziale la complessità computazionale è:

$$\theta(n^2) = n^2 \cdot 4$$

Poiché il controllo va effettuato per ogni cella e ci sono n^2 celle. Ogni cella richiede di effettuare 4 controlli, ognuno per ogni vicino. In questo caso, invece si è scelta una decomposizione a blocchi, il tempo parallelo sarà invece dato da:

$$T_p = \frac{n}{\sqrt{p}} \cdot \frac{n}{\sqrt{p}} + 4\frac{n}{\sqrt{p}}$$

Il primo termine è dovuto al calcolo, mentre, il secondo termine invece è dovuto allo scambio dei bordi tra i vari processi.

Di conseguenza l'overhead sarà:

$$T_0 = n^2 + \sqrt{p}n - n^2 = \sqrt{p}n$$

Di conseguenza l'isoefficienza sarà data da:

$$M\left(\frac{C^2\sqrt{p}^2}{p}\right) = C^2$$

$$n^2 \geq C\ p$$

La prima è la funzione di isoefficienza riguardante la memoria mentre la seconda è la vera funzione di isoefficienza. Per entrambe l'algoritmo risulta essere perfettamente scalabile.

Verifica regola di connessione dei bianchi

Per questa parte dell'algoritmo non è stato possibile trovare un algoritmo in grado di avere buone performance. In sequenziale il problema è stato risolto immaginandolo come una DFS in un grafo. Esistono algoritmi in grado di risolvere questo tipo di problema in parallelo, anche con discrete performance, ma funzionano bene quando il grafo è di dimensioni molto maggiori rispetto a quelli presi in considerazione in questi progetti.

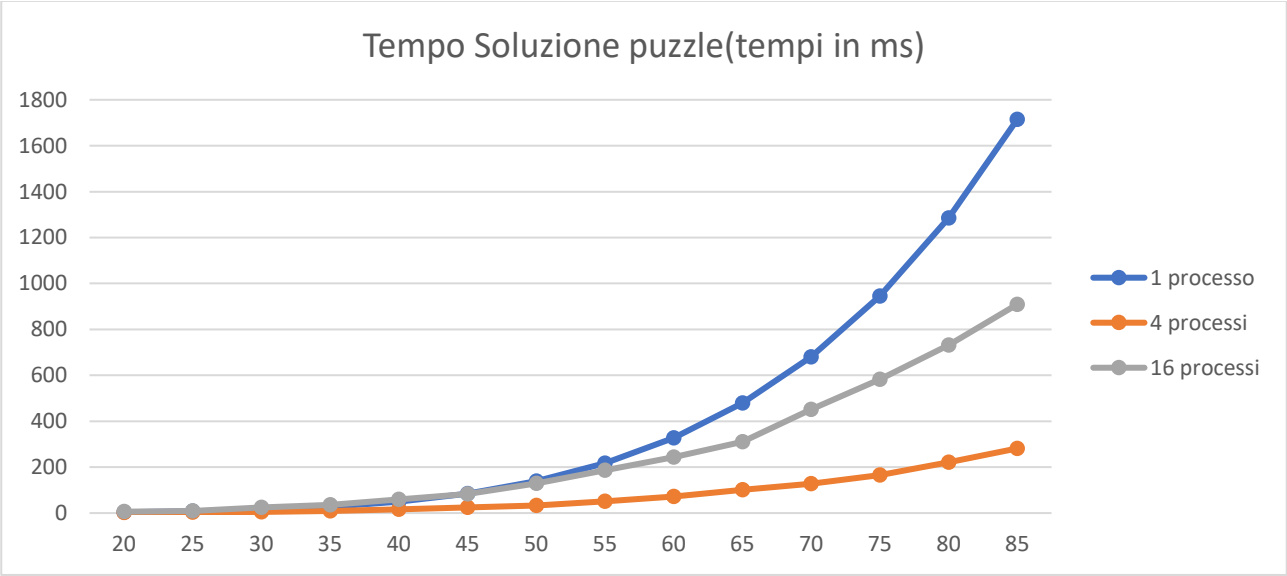
Conclusioni

Come già detto, una parte del codice viene eseguito in sequenziale quindi aumentando i processori senza aumentare la dimensione della matrice -**strong scaling-** le performance peggioreranno inesorabilmente.

Quanto appena detto può sembrare in contrasto con i risultati trovati sperimentalmente ma così non è , i risultati sono buoni solo perché il numero di processi utilizzati è piccolo.

Risultati

Come si è già detto si sono avuti risultati superlineari grazie ad una migliore ottimizzazione della cache.



I risultati riportano la media di 10 esecuzioni con 1, 4, 16 processori nella soluzione di puzzle di varie dimensioni da 20x20 fino a 85x85. Fino a matrici 40x40 non vi sono grosse differenze, si ha comunque uno speedup lineare ma, poiché tempi di calcolo sono di pochi millisecondi, non si apprezza nessuna differenza significativa, mentre per matrici di dimensioni 85x85 si ha addirittura uno speedup doppio rispetto a quello lineare.

La macchina su cui sono stati effettuati i test ha meno di 16 processori ed è per questo che si ha un peggioramento delle performance dell'algoritmo in quelle esecuzioni. Proprio per i pochi processori a disposizione della macchina su cui sono stati effettuati i test non si è verificato nessun fenomeno di decadimento delle performance all'aumentare dei processi utilizzati dovuta al fatto che, indipendentemente dal numero di processi utilizzati, la funzione che verifica la terza regola del gioco -connessione dei bianchi- è eseguita in sequenziale.

Probabilmente, aumentando il numero di processori, le performance peggiorerebbero

Lavori futuri

Ci sono vari possibili miglioramenti da poter fare al programma. Primo tra tutti trovare un algoritmo parallelo capace di verificare in maniera efficiente che tutti i bianchi siano connessi.

Effettuare uno *scaling down* dei task per poter avere un numero di processi che non sia obbligatoriamente un quadrato perfetto.