Algoritmi Avanzati – Relazione di Progetto: "Skyline"

Fabio Biselli – Mat: 0000734326

1 Introduzione

Il progetto proposto riguarda l'implementazione di un algoritmo parallelo per determinare lo *skyline* di un isieme di punti nello spazio, su architetture a memoria distribuita utilizzando il linguaggio `C` e la libreria `MPI`. Il processo principale deve leggere le coordinate dei punti in input da un file di testo `in.txt` composto da $N + 1$ righe in cui la prima specifica il numero di punti (N), e le successive contengono le coordinate di ogni punto. Al termine dell'esecuzione l'insieme dello skyline deve essere salvato con le medesime caratteristiche nel file `out.txt`. Si assume che i punti in input appartengano allo spazio euclideo a tre dimensioni e che il loro numero sia sempre (molto) maggiore del numero di processi `MPI`.

Definizione 1.1. Sia $P = \{p_0, p_1, \ldots, p_{N-1}\}$ un insieme di punti in tre dimensioni ($p_i = \langle x_i, y_i, z_i \rangle \in \mathbb{R}^3$). Si dice che p_i *domina* p_j, e scriveremo $p_i \succ p_j$, se si verificano le seguenti condizioni:

- $x_i \geq x_j, y_i \geq y_j, z_i \geq z_j$;
- Esiste almeno una coordinata per cui la disuguaglianza valga in senso stretto.

Proposizione 1.2. *Transitività della relazione* $\succ$.

$$\forall p, q, r \in P, \quad p \succ q \ e \ q \succ r \Rightarrow p \succ r.$$

Definizione 1.3. Sia P un insieme di punti, si definisce lo *skyline* di P l'insieme $Sk(P)$ composto da tutti i punti di P che non sono dominati da alcun altro punto di P.

$$Sk(P) := \{s \in P \colon \neg\exists\ t \in P \colon t \not\succ s\}\,.$$

Si noti che per definizione nessun punto domina se stesso.

2 Implementazione

L'implementazione sequenziale (descritta nello pseudocodice 1) suggerita per questo algoritmo prevede l'utilizzo di un array di interi S di lunghezza pari al numero dei punti in input. Questo viene inizializzato impostando tutti i valori ad 1. In seguito vengono confrontati tutti i punti mediante un operatore binario $\succ$ (domina), ogni punto viene confrontato con tutti gli altri per un totale di $O(N^2)$ chiamate. Se $p_i \succ p_j$, il relativo valore di p_j in S ($S[j]$) viene impostato a 0. Al termine della procedura l'array S conterrà le posizioni della skyline: se $S[i] = 1$ allora $p_i \in Sk(P)$.

Pseudocodice 1 Skyline sequenziale.

```
function SKYLINE(x[N], y[N], z[N])
    int S[N];
    for i ← 0 to N − 1 do
        S[i] ← 1;
    end for
    for i ← 0 to N − 1 do
        for j ← 0 to N − 1 do
            if ⟨x[i], y[i], z[i]⟩ ≻ ⟨x[j], y[j], z[j]⟩ then
                S[j] ← 0;
            end if
        end for
    end for
end function
```

La soluzione parallela implementata sfrutta la transitività della relazione *domina*, in sostanza l'algoritmo è suddiviso in due fasi. Nella prima fase i dati in input vengono partizionati in base al numero di processi disponibili ed inviati ai processi client (server compreso), in ogni partizione è quindi utilizzato l'algoritmo sequenziale proposto. Ogni processo invia la propria parte di array S (di dimesione fissata) al processo "server".

Nella seconda fase, che riguarda il solo processo server, vengono creati gli array dei punti ottenuti dalla prima, a cui vengono eventualmente aggiunti quelli in avanzo dalla procedura di partizione dei dati in input. Infatti se il numero di processi p non divide il numero di punti in input N si ottiene un insieme di punti di "resto" che devono essere aggiunti in questa fase. Tale numero è comunque compreso tra 1 e $p - 1$, quindi possiamo assumere che sia irrilevante in termini di complessità computazionale. A questo punto si utilizza nuovamente l'algoritmo sequenziale sull'insieme di punti per ottenere la skyline finale.

Si può vedere questo algoritmo come una specie di torneo, in cui si hanno un numero p di gironi di qualificazione (in cui però non è dato sapere a priori

il numero di vincitori) ed un girone finale in cui partecipano tutti i qualificati più le "teste di serie" (l'eventuale resto).

3 Analisi

Il punto di forza di questo algoritmo è dato dalla transitività della relazione *domina* ($\succ$) che consente di ridurre notevolmente il numero totale di confronti. Questo approccio funziona molto bene con insiemi di punti densi, in cui il numero di "vincitori" per ogni partizione è basso rispetto al numero totale, e quindi si comporta molto bene nel caso medio. D'altra parte, nel caso pessimo (quello in cui tutti i punti in input appartengono alla skyline) si comporta peggio di quello sequenziale, dovendo aggiungere al tempo sequenziale il tempo relativo alla prima fase.

3.1 Scalabilità

L'algoritmo parallelo proposto ed implementato riduce drasticamente i tempi di calcolo rispetto alla versione sequenziale proposta. Questo è dovuto al fatto che il secondo agisce "a forza bruta", mentre il primo, oltre a suddividere il carico della computazione su più processi, sfrutta la transitività di $\succ$. Utilizzando l'algoritmo `MPI` implementato con $\mathbf{N} = 1$ per l'analisi infatti si ottengono risultati non obiettivi, con *speedup* ed *efficienza* esageratamente alte. Al fine di generare dati compatibili si è implementato un algoritmo sequenziale che potesse sfruttare il partizionamento. I dati riportati nelle tabelle 3 e 4 sono stati ottenuti confrontando i tempi dell'algoritmo seriale riportati nella tabella 1, in cui $\mathbf{N}$ rappresenta il numero di partizioni, ed i tempi dell'algoritmo parallelo riportati nella tabella 2, in cui `comm_sz` rappresenta il numero di processi.

La tabella 3 mostra la *speedup* ottenuta. Si può notare che, fissato il numero dei processi, questa aumenti all'aumentare dei dati in input e quindi all'aumentare della dimensione delle partizioni. A prima vista potrebbe sembrare un comportamento anomalo, in realtà ciò è dovuto all'aumento della densità delle partizioni. Ovviamente questo comportamento è strettamente legato alla "qualità" dei punti in input, mi aspetto che per insiemi di punti meno densi il fenomeno sia minore e che comunque, all'aumentare dei dati si superi la soglia ottimale e lo speedup dimunisca.

Fissando ora il numero di dati input si può notare un comportamento che avvalora la tesi di cui sopra. In generale all'aumento dei processi corrisponde un aumento della speedup, ma abbiamo un'eccezione nella prima colonna (3500) in cui lo speedup diminusice utilizzando 8 processi, in questo caso probabilmente la soglia ottimale del rapporto "Numero di punti" / `comm_sz` è stata superata.

N	Numero di punti			
	3500	7500	15000	25000
2	0.11	0.46	1.81	5.00
3	0.07	0.31	1.21	3.33
4	0.05	0.23	0.91	2.50
8	0.03	0.12	0.46	1.25

Tabella 1: Esecuzione seriale

comm_sz	Numero di punti			
	3500	7500	15000	25000
2	0.05	0.21	0.82	2.22
3	0.03	0.10	0.38	1.04
4	0.02	0.06	0.22	0.59
8	0.014	0.03	0.10	0.25

Tabella 2: Esecuzione parallela

comm_sz	Numero di punti			
	3500	7500	15000	25000
2	2.2	2.2	2.2	2.25
3	2.3	3.1	3.18	3.20
4	2.5	3.83	4.13	4.23
8	2.14	4.0	4.6	5.0

Tabella 3: Speedup

comm_sz	Numero di punti			
	3500	7500	15000	25000
2	1.10	1.10	1.10	1.12
3	0.77	1.03	1.06	1.06
4	0.63	0.95	1.03	1.05
8	0.26	0.50	0.57	0.62

Tabella 4: Efficienza

Ulteriori test con `comm_sz` > 8 sono stati eseguiti, ed in tutti i casi si è notato una stabilizzazione dei tempi di risposta ed un corrispondente degrado della speedup, ma questo potrebbe essere dovuto anche alla macchina su cui sono stati effettuati i test (Intel® Core™ i7-3630QM CPU @ 2.40GHz×8).

Infine la tabella 4 mostra l'efficienza dell'algoritmo, evidenziando il fatto che questa aumenti con l'aumento del rapporto "Numero di punti" / `comm_sz`.

3.2 Speedup superlineare

Un dato che balza all'occhio guardando le tabelle è la presenza di alcuni casi di speedup superlienare e la relativa efficienza > 1. Solitamente, dati p processi, la speedup $S(p)$ è minore o uguale a p, il caso ottimo si ha quando $S(p) = p$ e di conseguenza l'efficienza $E(p) = 1$. In genere questo fenomeno si verifica conseguentemente all'utilizzo della cache di memoria, tuttavia osservando i dati sembra che anche la densità dei punti, in questo specifico caso, possa dare il proprio contributo.