Progetto di High Performance Computing 2024/2025

Enrico Marchionni, matr. 0001032322

18 dicembre 2024

1 Introduzione

...

2 Versione Seriale

L'algoritmo implementato è:

```c
int skyline(const points_t *points, int *s)
{
  const int D = points->D;
  const int N = points->N;
  const float *P = points->P;
  int r = N;

  /* Inizializzazione */
  for (int i = 0; i < N; i++) {
    s[i] = 1;
  }

  /* Calcolo skyline */
  for (int i = 0; i < N; i++) {
    if (s[i]) {
      for (int j = 0; j < N; j++) {
        if (s[j] && dominates(&(P[i * D]), &(P[j * D]), D)) {
          s[j] = 0;
          r--;
        }
      }
    }
  }
  return r;
}
```

Listing 1: Algoritmo per il calcolo dello skyline in C.

È l'algoritmo seriale proposto è tipico per soluzioni brute-force del problema dello skyline. Il costo computazionale dell'algoritmo in Listing 1 è in generale $O(N^2D)$. Infatti il ciclo esterno viene percorso N volte, nelle quali, nel caso peggiore, vengono eseguite altrettante N operazioni di confronto tra D elementi (da cui $N \times N \times D$). Si può notare che D può influire anche molto nella complessità, in particolare se $D >> N^2$. Nel caso in cui $N^2 >> D$, si considera $O(N^2)$, nel caso in cui $D >> N^2$, $O(D)$, nell'ultimo caso in cui $D \approx N^2$ allora $O(N^2D)$.

Caso	**Complessità**	**Range**
Pessimo	$\Theta(N^2D)$	Tutti i punti sono dello skyline.
Medio	$\Theta(N^2D)$	La metà dei punti fanno parte dello skyline.
Ottimo	$\Theta(ND)$	Il primo punto analizzato domina tutti gli altri.

Tabella 1: Input data.

Come si può vedere dalla Tabella 1 la complessità in realtà varia anche di molto in base all'input fornito. Fattore importante nella misura della weak scaling efficiency per esempio.

3 Versione OpenMP

Analizzando in codice in Listing 1 si può notare che:

- Il ciclo di inizializzazione segue il pattern *embarrassingly parallel*. Quindi la sua computazione può essere svolta in modo indipendente da più processi. Inoltre, dato il perfetto bilanciamento del carico, il pattern *partition* può essere applicato con partizioni a grana grossa, per ridurre al minimo anche l'overhead per la gestione dei thread;
- Per quanto riguarda i due cicli che servono per il calcolo dello skyline invece, va considerato che c'è una dipendenza tra i dati, in particolare il check di `s[i]` dipende dalla possibile assegnazione a `s[j]` nel caso in cui `i = j` (oltre ai problemi di concorrenza). Inoltre anche l'accesso ad `r` sarebbe una 'race condition'. Quindi:
  - Per il primo problema ho scelto di parallelizzare solo il ciclo più interno. Il ciclo più esterno infatti viene svolto `N` volte senza fare operazioni, quindi ha costo trascurabile, inoltre per poterlo parallelizzare andrebbe modificato il codice, trovando per esempio un altro algoritmo;
  - Per il secondo, invece, si potrebbe usare la direttiva *atomic*, ma si può anche notare che per la variabile `r` può essere applicato il pattern *reduction* sull'operazione di somma.

  Per migliorare ulteriormente le prestazioni si deve considerare che il lavoro nel ciclo più interno per il calcolo dello skyline non è bilanciato se si fa un partizionamento a grana grossa. Data la diversità dei possibili input ed il possibile sbilanciamento del carico conviene fare partizionamento più fine ed applicare anche il pattern *master-worker*, che nonostante l'overhead più alto. Questo nella pratica favorisce il bilanciamento del carico in casi in cui l'input è per sua natura molto sbilanciato nella distribuzione del carico di lavoro.

3.1 Strong Scaling Efficiency

La dimensione del problema rimane fissa. Il numero di core varia in p da $1, \ldots, n$, dove n è il numero massimo di core logici nella macchina.

Per farlo ho considerato come input il file `worst-N100000-D10.in` (file generato dal programma fornito `inputgen`), in cui $N = 100000$ e $D = 10$.

3.2 Weak Scaling Efficiency

Cambia la dimensione 'globale' del problema ma quella locale, per ogni core, rimane il più costante possibile. Il numero di core varia in p da $1, \ldots, n$, dove n è il numero massimo di core logici nella macchina.

Per farlo ho considerato come input diversi file generati da `inputgen`, fissando $D = 10$ e variando N in modo opportuno, come considerato nel seguito.

In questo particolare caso pessimo, il costo computazionale è pari a $\Theta(N^2 D)$. Da qui posso ricavare che il lavoro per core è:

$$\frac{N^2 \cdot D}{p} = CONST \tag{1}$$

In particolare in Equazione 1 il lavoro viene posto costante per ipotesi della weak scaling efficiency. Dobbiamo ricavare il valore di N in funzione del numero di processi.

$$N = \sqrt{p} \cdot \frac{CONST}{\sqrt{D}} = \sqrt{p} \cdot CONST' \tag{2}$$

Quindi, da Equazione 2, supponendo D costante, N può essere in pratica calcolato come:

$$N = \sqrt{p} \cdot N_0 \tag{3}$$

Dove N_0 è il valore iniziale di N, infatti per un singolo core si ha $N = N_0$. In particolare per scegliere N_0 opportuno ho calcolato $N_0 = \frac{N}{\sqrt{p}}$, formula inversa da Equazione 3. A questo punto N_0 si calcola con N e p scelti uguali a quelli usati nell'ultimo test della strong scaling efficiency.

4 Versione MPI/CUDA

...

5 Conclusioni

...