Progetto di High Performance Computing 2024/2025

Enrico Marchionni
enrico.marchionni@studio.unibo.it

28 febbraio 2025

1 Introduzione

Progetto di High-Performance Computing - a.y. 2024-2025 sulla parallelizzazione di un algoritmo brute-force per il calcolo dello skyline.

2 Versione Seriale

L'algoritmo seriale da parallelizzare 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;
}
```

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

Quello descritto è l'algoritmo seriale proposto, tipico per soluzioni brute-force del problema dello skyline. Il costo computazionale dell'algoritmo nel Listato 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 $D >> N$ (oppure $D >> N^2$). 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 $O(N^2 D)$.

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

Tabella 1: Complessità computazionale.

Come si può vedere dalla Tabella 1 la complessità in realtà varia anche di molto in base all'input fornito. La complessità computazionale è un fattore importante nei test fatti di seguito, per esempio nei test sulla Weak Scaling Efficiency (OpenMP).

3 Versione OpenMP

Analizzando il codice nel Listato 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 di `s[j]` nel caso in cui `i = j`, questo può causare problemi di concorrenza. Inoltre anche la scrittura su `r` può portare ad una 'race condition' nella versione parallela. Quindi:
  - Per il primo problema ho scelto di parallelizzare solo il ciclo più interno. Il ciclo più esterno infatti non svolge il cuore della computazione, inoltre per poterlo parallelizzare andrebbe modificato il codice, cambiando l'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 potrebbe considerare che il lavoro nel ciclo più interno per il calcolo dello skyline non è necessariamente bilanciato se si fa un partizionamento a grana grossa. Data la diversità dei possibili input ed il possibile sbilanciamento del carico potrebbe convenire fare un partizionamento più fine o in caso estremo applicare anche il pattern *master-worker*, che nonostante l'overhead più alto favorisce un bilanciamento maggiore nei casi meno favorevoli. In questo caso particolare non sono stati notati miglioramenti applicando il partizionamento a grana fine, nè il paradigma *master-worker*, entrambi testati con varie dimensioni dei blocchi, quindi questo non è stato applicato nella versione proposta (si potrebbero fare input ad hoc, in particolare se $D << N$ non fosse un'ipotesi, per verificare questo sbilanciamento).

Aspetto importante da considerare è l'overhead eccessivo che si ha nelle consecutive creazioni e distruzioni del team di thread ogni volta che il ciclo interno al calcolo dello skyline viene eseguito in parallelo usando la clausola `#pragma omp parallel for`. I compilatori più recenti ottimizzano da soli evitando creazioni e distruzioni multiple in cicli, questo però non è detto e dipende completamente dalla macchina utilizzata. L'implementazione proposta fa la cosa giusta in ogni caso. Proprio per questo ho

scelto di usare la clausola `#pragma omp parallel` in un unico blocco che contiene i cicli di inizializzazione e di calcolo dello skyline e poi con l'altra clausola `#pragma omp for` di parallelizzare i cicli. L'utilizzo di `#pragma omp for` invece di `#pragma omp parallel for` garantisce una miglior gestione del team di thread, che nella pratica risulta in una concreta velocizzazione della versione non ottimizzata. Si noti che nel particolare algoritmo implementato `s[i]` è acceduto senza una race condition da ogni singolo thread il parallelo. Questo perché alla fine del `#pragma omp for` c'è una barriera implicita, così come all'inizio ve ne è una esplicita, guarda la direttiva `#pragma omp barrier`. In linea teorica si potrebbe pensare che in questo caso particolare la lettura di questa variabile anche se fatta senza la barriera esplicita, non dovrebbe comportare problemi. Tuttavia in questo caso potrebbe verificarsi che solo alcuni thread entrino nell'`if` e inizino ad eseguire il `#pragma omp for` non aspettando gli altri che stanno ancora facendo il check su `s[i]`. Questo, anche se non sono state trovate delucidazioni a riguardo nella documentazione, potrebbe portare problemi dato che dopo il `#pragma omp for` c'è una barriera di sincronizzazione implicita tra i thread. Quindi, in questo caso, sebbene nella pratica non sembri esserci un vero e proprio problema, la race condition è evitata per sicurezza usando una barriera di sincronizzazione `#pragma omp barrier`.

3.1 Strong Scaling Efficiency

All'aumentare del numero di core la dimensione totale del problema rimane fissa, di conseguenza quella locale, per ogni core, diminuisce. L'obiettivo è ridurre il tempo di esecuzione totale aggiungendo sempre più core. 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-N50000-D10.in` (file generato dal programma fornito `inputgen`), in cui $N = 50000$ e $D = 10$.

Di seguito i grafici con le statistiche ottenute dai test pratici sul server `isi-raptor03.csr.unibo.it`. La CPU del server, su cui sono stati effettuati i test, è un `Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz` con 12 core fisici (quindi senza Hyper-Threading). Tutti i dati successivamente riportati rispettano le predizioni teoriche che si possono fare. Unica eccezione è per i dati ottenuti con tutti i dodici core che sono abbastanza incoerenti anche tra di loro, perciò non riportati nel grafico. Nonostante non riportati in questa relazione, tali dati sono stati analizzati e possono essere trovati nei fogli di calcolo allegati contenenti i risultati completi relativi ai test effettuati. Probabilmente, per poter considerare anche l'ultimo core, andrebbero fatti più test (centinaia) e presi solo i dati migliori. Questo sbalzo è dovuto al fatto che un core è sempre più attivo degli altri perché nel server ci sono vari processi in background che impiegano in parte la CPU e per come il SO schedula i processi nei core che ha a disposizione.

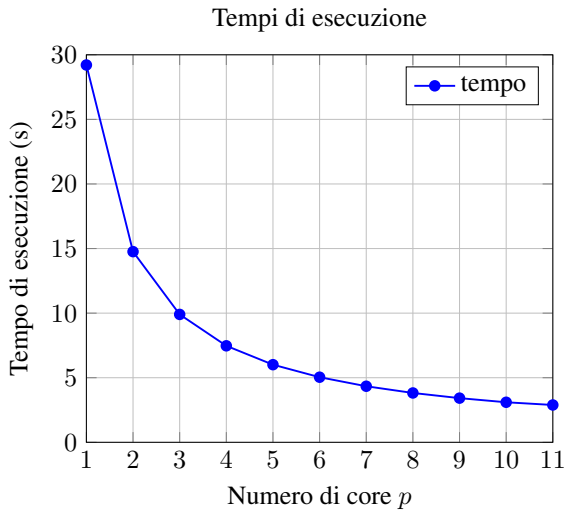
Nel grafico in Figura 1a viene mostrato il tempo di esecuzione impiegato nella computazione che cala sempre, in modo inversamente proporzionale, all'aumentare dei core. Inoltre nel grafico in Figura 1b si nota uno speedup quasi lineare.

Nel grafico in Figura 2 si può notare che l'efficienza, seppur lentamente diminuendo, rimane vicina ad 1.

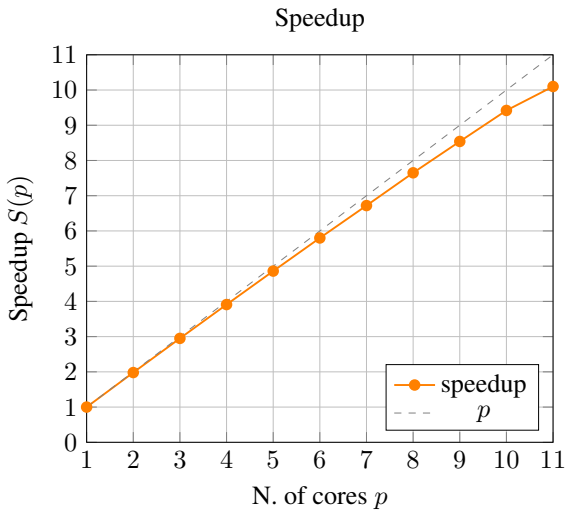
3.2 Weak Scaling Efficiency

All'aumentare del numero di core cambia la dimensione totale del problema ma quella locale, per ogni core, rimane il più costante possibile. L'obiettivo è risolvere problemi più grandi nello stesso tempo. 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. Quindi i file di input considerati sono del tipo `worst-NX-D10.in`, dove X è proporzionale al lavoro che viene fatto con $N = 50000$ con tutti e 12 i core del server.



(a) OpenMP: tempo di esecuzione.



(b) OpenMP: accelerazione.

Figura 1: OpenMP: tempi di esecuzione e speedup.

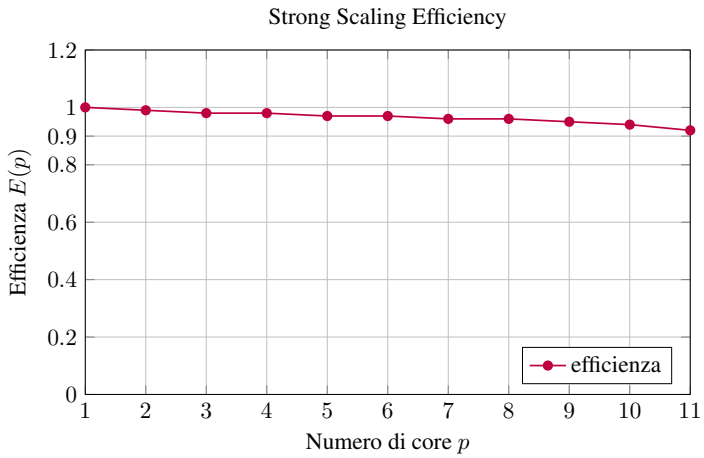


Figura 2: OpenMP: efficienza.

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, dall'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.

In base alle considerazioni fatte è stato scelto di calcolare N_0 come $N_0 = \lceil \frac{50000}{\sqrt{12}} \rceil = 14434$. N_0 potrebbe anche essere scelto a priori, però questa precisione ha permesso di favorire tempi ragionevoli di esecuzione, almeno qualche secondo, in modo da garantire una buona coerenza tra i risultati ed allo stesso tempo si è evitato di occupare la macchina per decine o centinaia di secondi. Da qui N nelle iterazioni successive (che sono 12) vale $N = \lceil \sqrt{p} \cdot N_0 \rceil$.

p	N	D
1	14434	10
2	20413	10
3	25000	10
4	28868	10
5	32275	10
6	35356	10
7	38189	10
8	40826	10
9	43302	10
10	45644	10
11	47872	10
12	50001	10

Tabella 2: Dimensione dell'input all'aumentare del numero di core.

Di seguito i grafici con le statistiche ottenute dai test pratici sul server `isi-raptor03.csr.unibo.it`. Come input sono state usate le dimensioni riportate in Tabella 2. Anche in questo caso sono stati omessi i risultati ottenuti con tutti i 12 core per lo stesso motivo già spiegato appena prima di riportare i dati della weak scaling efficiency.

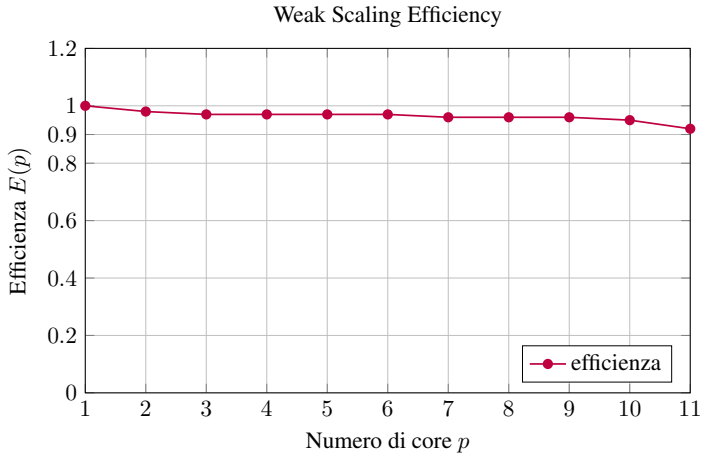


Figura 3: OpenMP: efficienza.

Nel grafico in Figura 3 si può notare che, anche in questo caso, l'efficienza, seppur lentamente diminuendo, rimane vicina ad 1. Questo conferma quanto ottenuto anche nella strong scaling efficiency.

4 Versione CUDA

Valgono le considerazioni già fatte per OpenMP. L'algoritmo di partenza è stato testato con varie soluzioni. Quella scelta permette di mantenere la struttura dell'algoritmo di partenza descritto in Listato 1. La versione implementata sfrutta il massimo numero di thread per blocco e non fa utilizzo della shared memory.

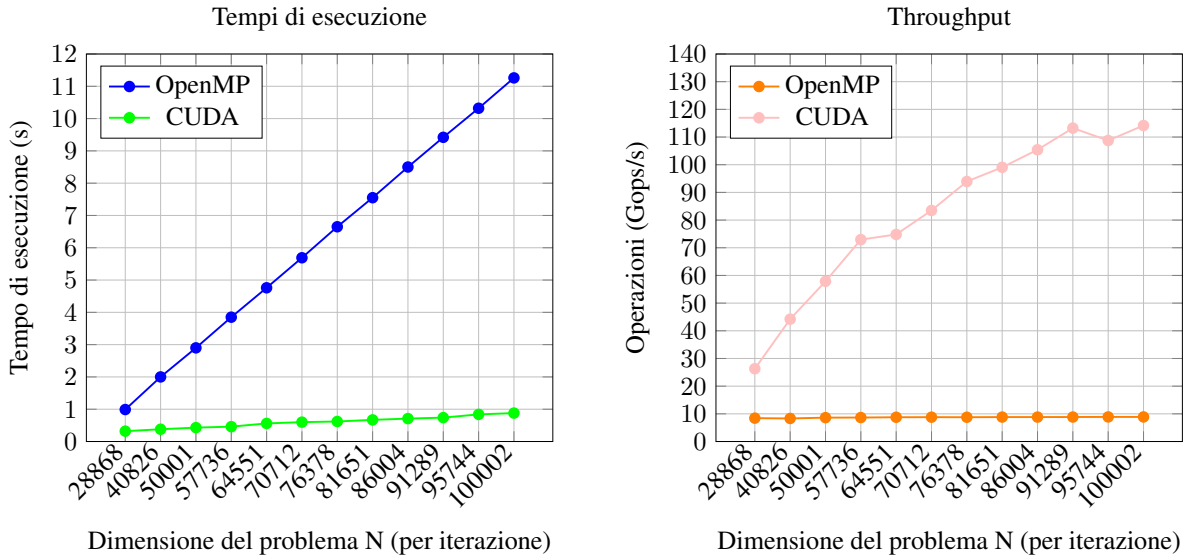
Il codice consiste nell'esecuzione di 2 kernel:

- Il primo svolge l'inizializzazione dell'array dei punti che sono nello skyline in concomitanza con la prima iterazione del ciclo che fa il vero e proprio calcolo dello skyline. In questo caso un kernel specifico è necessario perché altrimenti dovrebbe essere fatta una sincronizzazione tra tutti i thread della griglia (tra più blocchi) per l'accesso a `s[i]`.
- Nel secondo kernel ogni thread esegue un'iterazione del ciclo più interno del calcolo dello skyline. Tutti i thread eseguono il ciclo più esterno completamente. Nel codice è presente una possibile race condition sul check di `s[i]` che però è accettabile perché al massimo può comportare qualche iterazione in più che non sarebbe necessaria, ma che non ha effetti indesiderati. Si noti che `s[g_index]` non è stato settato a `false` in modo atomico perché questo non è necessario. Per la riduzione di `r`, invece, è stato scelto di eseguire il decremento atomicamente, invece di eseguire la vera e propria riduzione. Questo perché intuitivamente l'operazione viene fatta raramente da più thread nello stesso momento, inoltre nella pratica è stato verificato che il decremento atomico risulta più conveniente della riduzione vera e propria per i vari test sperimentati.

4.1 Statistiche

Dato che in questo caso il concetto di speedup non può essere considerato, vengono invece considerati tempo d'esecuzione, throughput e speedup di CUDA rispetto ad OpenMP.

Anche in questo caso i dati riportati su OpenMP si riferiscono alla CPU che usa undici dei dodici core a sua disposizione, che nella pratica risulta in valori attendibili e addirittura migliori dei test con i dodici core. Anche questi dati possono essere trovati nei fogli di calcolo presenti nella documentazione.



(a) Tempi di esecuzione.

(b) Throughput.

Figura 4: OpenMP e CUDA: tempi di esecuzione e throughput a confronto.

Per quanto riguarda i tempi di esecuzione confrontati in Figura 4a, si può notare come i valori di OpenMP incrementino molto più velocemente che quelli di CUDA, all'aumentare della dimensione del problema.

In Figura 4b, invece, si può notare come CUDA abbia un throughput che aumenta quasi linearmente, mentre per OpenMP questo rimane pressoché costante.

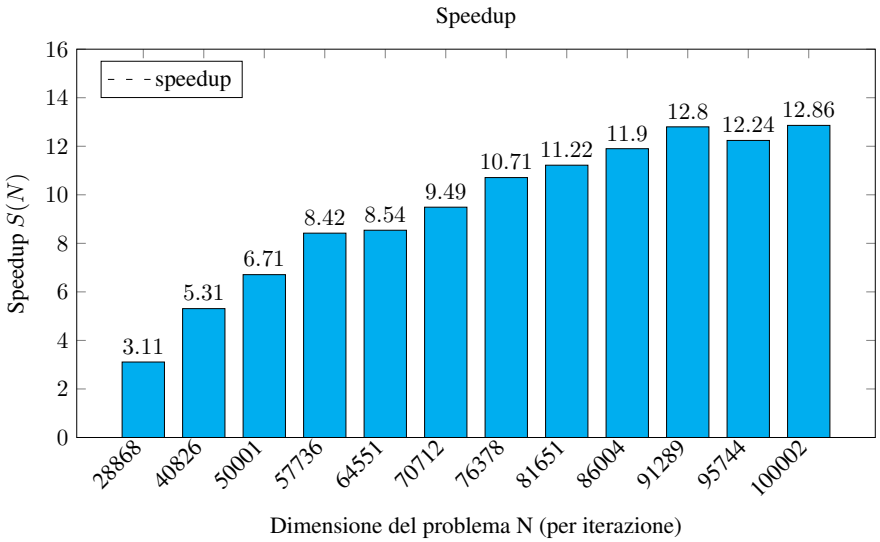


Figura 5: OpenMP e CUDA: accelerazione di CUDA rispetto a OpenMP.

In Figura 5, si può notare che lo speedup di CUDA rispetto ad OpenMP incrementa linearmente fino quasi a stabilizzarsi dopo una certa soglia ($N = 91289$).

5 Conclusioni

Per concludere, la parallelizzazione dell'algoritmo proposto nel Listato 1 è stata fatta in OpenMP e CUDA. Per entrambe sono state analizzate le relative versioni parallele. Alla fine, per comparare le due versioni, è stato riportato lo speedup di CUDA rispetto ad OpenMP.