Relazione di progetto
Corso di High Performance Computing
A.A. 2024/25

Alessandro Valmori
Matricola: 0001089308
February 15, 2025

1. Introduzione

Si sceglie di implementare la parallelizzazione dell'algoritmo skyline in due diverse versioni, utilizzando prima OpenMP e poi MPI, due tecniche di parallelizzazione rispettivamente a memoria condivisa e a memoria distribuita. L'approccio alla parallelizzazione utilizzando i due paradigmi condivide la fase di partizionamento del dominio iniziale fra i processi, utilizzando in particolare un partizionamento statico a grana grossa.

2. Versione OpenMP

Per la versione a memoria condivisa utilizzando OpenMP, si sceglie di parallellizzare il confronto fra un punto e tutti gli altri. In particolare il ciclo esterno, che itera attraverso i punti da esaminare, è stato parallellizzato utilizzando la direttiva `#pragma omp parallel for`. Per calcolare il numero di punti nello skyline set (l'insieme di punti non dominati), è stato applicato il pattern di riduzione. Nella riduzione, il valore di `r` (il contatore che tiene traccia dei punti nello skyline) viene modificato da più thread senza conflitti grazie alla direttiva `reduction(-:r)`. Ogni volta che un punto è trovato dominante su un altro, il punto dominato viene segnato come non appartenente allo skyline (modificando l'array `s`), e il contatore `r` viene decrementato. Per evitare il controllo ripetuto di punti già rimossi dallo skyline, ogni punto viene esaminato una sola volta, e non appena si trova un punto che domina un altro, quest'ultimo viene rimosso.

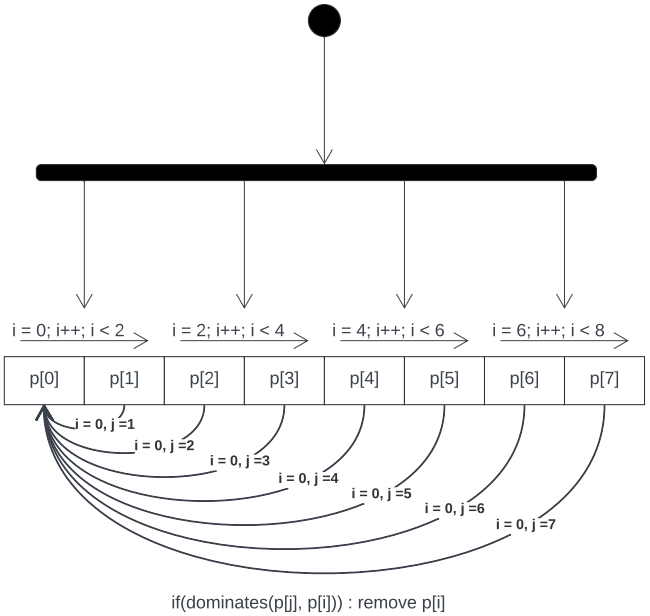


FIGURE 1. Schema concettuale del processo svolto dall'algoritmo parallelo con OpenMP

3. Versione MPI

Per la versione a memoria distribuita utilizzando MPI è stata implementata una soluzione che presenta un processo più scandito e verboso, che comincia con la suddivisione del dominio iniziale in parti il quanto più possibile eguali fra i processi. Questo partizionamento avviene staticamente dopo che il processo `master` ha letto l'insieme di punti da standard input.

Una volta che ciascun processo detiene la propria partizione dell'input set originale, ognuno di essi applica la versione seriale dell'algoritmo skyline a quest'ultima, ottenendo quindi un sottoinsieme di tale partizione costituito dai punti che non sono dominati da alcun altro punto all'interno della partizione.

In seguito, il processo `master` esegue una `MPI_Gatherv` di ciascuno skyline set locale in un unico array, di cui ne viene poi distribuita una copia a ciascun processo, in modo da consentire il confronto fra lo skyline set locale e tale array. Infine, dopo che ciascun processo ha rimosso dalla propria partizione tutti i punti dominati, avviene una chiamata alla funzione `MPI_Reduce` per ottenere il numero totale di punti presenti nello skyline finale, seguita da un'ulteriore chiamata a `MPI_Gatherv` per concatenare tutti i singoli array locali nello skyline finale.

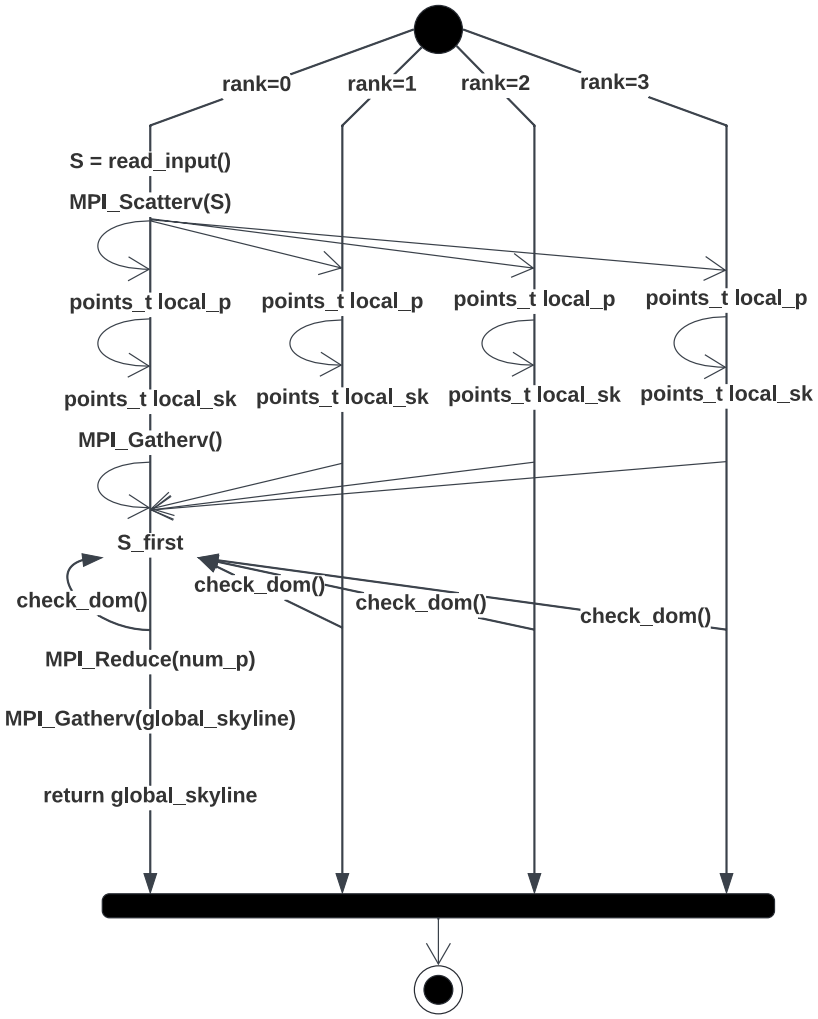


FIGURE 2. Schema concettuale del processo svolto dall'algoritmo parallelo con MPI

4. Prestazioni

Per valutare l'efficacia dell'implementazione parallela dell'algoritmo skyline, sono stati condotti test mirati all'analisi dello strong scaling e del weak scaling. L'obiettivo è verificare come il tempo di esecuzione varia al variare del numero di processi, sia mantenendo fisso il carico di lavoro totale (strong scaling) sia mantenendo costante il carico per processo (weak scaling).

I test sono stati eseguiti su una macchina con processore Intel Core i7-1165G7 di undicesima generazione (4 core fisici, 8 thread) e 16GB di RAM. L'algoritmo è stato eseguito 20 volte per ciascuna configurazione sperimentale, e il valore finale riportato corrisponde alla media dei tempi di esecuzione ottenuti, riducendo così l'effetto di eventuali variazioni dovute a fattori esterni, come il sistema operativo o altri processi in esecuzione.

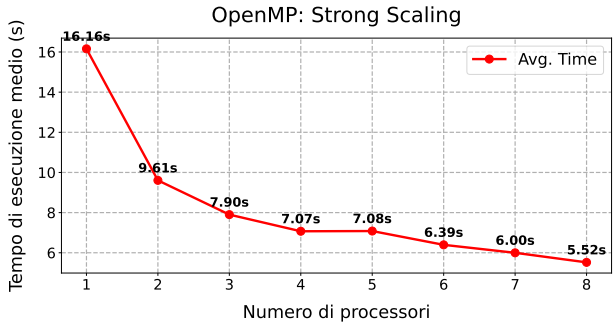
Per raccogliere i dati, è stato realizzato uno script bash che automatizza l'esecuzione delle prove. Lo script si occupa di lanciare il programma con diverse configurazioni di numero di processi e dataset di input, redirezionando l'output standard e gli errori in un file di log per facilitare l'analisi successiva. Inoltre, un insieme di script Python è stato utilizzato per elaborare i risultati e generare grafici rappresentativi delle prestazioni ottenute.

4.1. Strong Scaling

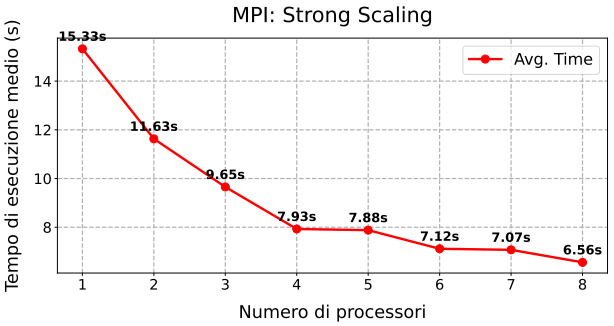
Durante la valutazione dello strong scaling si analizza il comportamento dell'algoritmo mantenendo costante la dimensione del problema e aumentando progressivamente il numero di processi utilizzati per l'esecuzione. L'obiettivo è osservare come varia il tempo di esecuzione e verificare se l'algoritmo è in grado di sfruttare in modo efficiente un numero crescente di processi.

Per questi test, è stato scelto un dataset di dimensione fissa, ovvero 50000 punti di 20 dimensioni dapprima costituenti uno Skyline set valido, rappresentativo di un caso d'uso realistico che corrisponde anche al caso pessimo. Il numero di processi è stato variato da 1 a 8, in modo da valutare il comportamento dell'algoritmo sia in condizioni di esecuzione seriale (1 processo) sia quando si sfruttano tutte le risorse disponibili sulla macchina di test (8 processi, pari al numero massimo di thread del processore).

Idealmente, in uno scenario di perfetto strong scaling, il tempo di esecuzione dovrebbe ridursi proporzionalmente all'aumento del numero di processi. Tuttavia, in una situazione reale, il comportamento può essere influenzato da diversi fattori, tra cui l'overhead della comunicazione tra processi, il bilanciamento del carico e le latenze introdotte dalla gestione della memoria.



A. Tempo di esecuzione in funzione del numero di processi (strong scaling) di OpenMP



B. Tempo di esecuzione in funzione del numero di processi (strong scaling) di MPI

FIGURE 3. Analisi dello strong scaling dell'algoritmo

4.2. Weak Scaling

Il weak scaling valuta come varia il tempo di esecuzione dell'algoritmo quando sia il numero di processi (o thread) che la dimensione del problema aumentano. L'obiettivo è determinare se l'efficienza parallela rimane stabile con l'aggiunta di risorse computazionali.

4.2.1. Metodologia di Scalabilità

La complessità parallela dell'algoritmo skyline può essere espressa come:

$$O\left(\frac{N^2 D}{p}\right)$$

dove:

- N = numero di punti,
- D = numero di dimensioni (assunto costante),
- p = numero di processi (o thread).

Ogni thread è responsabile di un sottoinsieme dei confronti totali N^2, dato da:

$$\frac{N^2}{p}$$

Per mantenere il carico di lavoro per processo costante, è necessario garantire che questo termine rimanga invariato all'aumentare di p. Ciò porta alla condizione:

$$\frac{(N')^2}{p'} = \frac{N^2}{p}$$

dove:

- N' è la nuova dimensione del problema con p' thread,
- N è la dimensione originale con p thread.

4.2.2. Derivazione del Fattore di Scalabilità

Riorganizzando la formula per N', si ottiene:

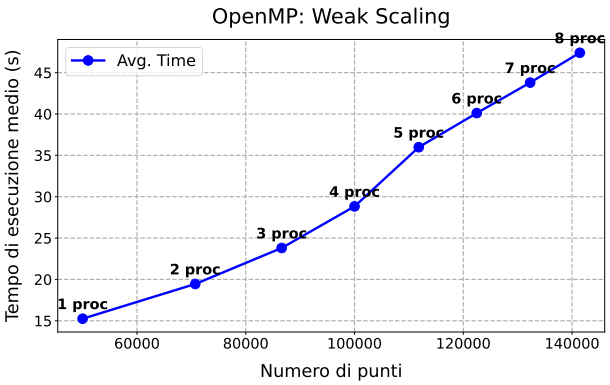
$$N' = N \times \sqrt{\frac{p'}{p}}$$

Questo significa che, per mantenere il carico di lavoro per processo costante, la dimensione del problema N deve essere aumentata di un fattore:

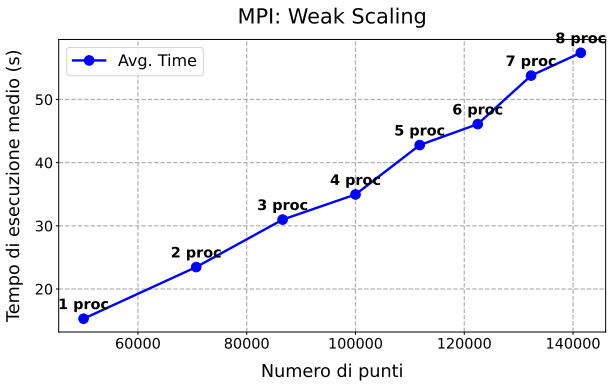
$$\sqrt{p}$$

4.2.3. Risultati Sperimentali

Per analizzare il weak scaling, si sono eseguiti test in cui sia il numero di thread p che la dimensione del dataset N sono stati aumentati secondo la regola di scalabilità derivata sopra.



A. Risultati del weak scaling con OpenMP.



B. Risultati del weak scaling con MPI.

FIGURE 4. Analisi del weak scaling per le implementazioni OpenMP e MPI.

I risultati in Figura 4 mostrano che il tempo di esecuzione, per entrambe le tecniche utilizzate, cresce approssimativamente linearmente al variare del numero dei processi e del numero di punti nel dataset in input. Ciò è legato all'overhead di accesso alla memoria per OpenMP, mentre per quando riguarda MPI esso è causato dall'overhead di comunicazione tra i processi, che aumenta all'aumentare del numero di processi utilizzati.

Si nota inoltre l'aumento improvviso del tempo di esecuzione quando si passa da 4 a 5 processi, ciò è dovuto al fatto che la macchina su cui sono stati eseguiti i test dispone di soli 4 core fisici, perciò utilizzare un numero di processi superiore a tale quantità porta a un incremento dell'overhead di gestione dei thread e della competizione per le risorse hardware.

Quando il numero di processi supera il numero di core fisici disponibili, i processi in eccesso vengono eseguiti sui thread logici (grazie all'hyper-threading), che condividono le unità di esecuzione dei core fisici. Questo causa un degrado delle prestazioni dovuto a diversi fattori. In primo luogo, vi è concorrenza per le unità di calcolo, poiché i processi devono condividere le stesse ALU e FPU, riducendo così l'efficienza di esecuzione. Inoltre, si verifica una maggiore pressione sulla cache: un numero elevato di processi attivi comporta un incremento nei cache miss, aumentando il numero di accessi alla memoria principale, che risulta significativamente più lenta. Infine, lo scheduling del sistema operativo introduce ulteriore overhead, poiché con più processi rispetto ai core fisici disponibili, il sistema deve gestire il contesto di esecuzione in modo più aggressivo, con un impatto negativo sulle prestazioni.

Nel caso di OpenMP, questo aumento del tempo di esecuzione è principalmente legato alla competizione per la memoria condivisa, poiché tutti i thread accedono agli stessi dati e devono sincronizzarsi frequentemente. Per quanto riguarda MPI, invece, il problema principale è l'aumento della latenza di comunicazione: con più processi distribuiti tra thread logici, i ritardi nella trasmissione dei messaggi tra i processi aumentano, degradando le prestazioni complessive.

In uno scenario ideale di weak scaling, il tempo di esecuzione dovrebbe rimanere costante con l'aumento simultaneo della dimensione del problema e delle risorse computazionali. Tuttavia le limitazioni reali contribuiscono alle deviazioni da questo comportamento ideale.

4.3. Speedup

I risultati sperimentali per lo strong scaling sono stati elaborati per ottenere lo speedup e l'efficienza per ciascun numero di processori. I calcoli sono stati effettuati utilizzando la formula:

$$\text{Speedup}(p) = \frac{T(1)}{T(p)} \quad \text{e} \quad \text{Efficienza}(p) = \frac{\text{Speedup}(p)}{p},$$

dove $T(1)$ è il tempo medio di esecuzione su 1 processore e $T(p)$ è il tempo medio di esecuzione su p processori.

Di seguito vengono riportate due tabelle riassuntive (con valori espressi in secondi per $T(p)$):

TABLE 1. Speedup e Efficienza per lo strong scaling: confronto tra OpenMP e MPI

a. OpenMP

N. Proc	$T(p)$ (s)	Speedup	Efficienza
1	16.159216	1.000	1.000
2	9.605886	1.682	0.841
3	7.900391	2.045	0.682
4	7.070698	2.285	0.571
5	7.081560	2.282	0.456
6	6.395000	2.527	0.421
7	5.998044	2.694	0.385
8	5.522114	2.926	0.366

b. MPI

N. Proc	$T(p)$ (s)	Speedup	Efficienza
1	15.327342	1.000	1.000
2	11.630479	1.318	0.659
3	9.654584	1.588	0.529
4	7.928183	1.933	0.483
5	7.880903	1.945	0.389
6	7.116237	2.154	0.359
7	7.072699	2.167	0.310
8	6.558208	2.337	0.292

Analisi dei risultati. Dall'analisi delle Tabelle 1a e 1b, emergono alcune osservazioni importanti riguardo lo scaling delle implementazioni OpenMP e MPI.

Innanzitutto, si nota che per entrambi i paradigmi di parallelismo lo speedup cresce all'aumentare del numero di processori, ma con un'efficienza decrescente. Questo comportamento è atteso, poiché con un numero crescente di processori l'overhead della sincronizzazione e della comunicazione diventa sempre più significativo.

Per quanto riguarda OpenMP, si osserva un buon incremento di speedup fino a circa 4 processori, dopodiché l'aumento diventa più contenuto. L'efficienza scende rapidamente, arrivando a valori inferiori al 50% quando si utilizzano più di 4 processori. Questo suggerisce che l'overhead della gestione dei thread e della memoria influisce negativamente sulle prestazioni.

Nell'implementazione MPI, lo speedup cresce più lentamente rispetto a OpenMP, con un'efficienza che cala più rapidamente all'aumentare dei processori. Questo è dovuto al maggiore overhead della comunicazione tra i processi MPI, specialmente nelle fasi di gathering e reduce. In particolare, si nota che oltre i 4 processori il guadagno prestazionale diventa sempre meno significativo, suggerendo che la comunicazione tra i nodi limita la scalabilità.

Infine, entrambi i metodi mostrano un limite nella scalabilità oltre un certo numero di processori, come osservato precedentemente nell'analisi del weak scaling, il che potrebbe indicare che l'algoritmo stesso introduce overhead di sincronizzazione e comunicazione difficili da mitigare. Inoltre, il calo di efficienza osservato oltre i 4 processori può essere attribuito anche al fatto che la macchina di test dispone di soli 4 core fisici. Quando il numero di processori supera questa soglia, i processi iniziano a essere eseguiti sui thread logici piuttosto che su core dedicati, introducendo ulteriore competizione per le risorse di calcolo e memoria. Questo comporta un aumento della latenza nell'accesso alla memoria e una riduzione dell'efficienza del parallelismo, limitando così il miglioramento dello speedup.

5. Conclusioni

Dai risultati ottenuti si deduce che, per il problema analizzato, OpenMP tende a essere più efficiente di MPI in termini di speedup ed efficienza per un numero limitato di processori, grazie al minore overhead di comunicazione. Tuttavia, l'efficienza scende rapidamente all'aumentare del numero di thread.

L'implementazione MPI, pur essendo più scalabile in teoria, soffre di overhead comunicativi che ne limitano il guadagno prestazionale oltre un certo numero di processori. Per migliorare la scalabilità, sarebbe utile valutare tecniche di riduzione delle comunicazioni inter-processo e ottimizzazione delle operazioni di gathering e reduce.

In sintesi, per sistemi con un numero ridotto di core, OpenMP sembra essere una scelta più vantaggiosa, mentre MPI potrebbe risultare più efficace per problemi più grandi e distribuiti su cluster con una migliore gestione della comunicazione.