Progetto di High Performance Computing 2022/2023

Lorenzo Drudi, matr. 0000969871
13/02/2023

Introduzione

In questa relazione viene presentato il lavoro svolto per il progetto del corso di High Performance Computing. Il compito assegnato era quello di realizzare due versioni parallele del modello SPH (Smoothed Particle Hydrodynamics) [1], un algoritmo utilizzato per descrivere il comportamento di corpi deformabili e, successivamente, esteso per simulare i fluidi.
In particolare, partendo dal codice seriale sviluppato da Lucas V. Schuermannm [2], lo scopo era quello di realizzare due versioni parallele utilizzando OpenMP per la prima versione e, a scelta, MPI oppure CUDA per la seconda versione. La mia scelta è ricaduta sull'utilizzo di MPI.
La parte più importante dell'implementazione e su cui ho effettuato un'analisi più approfondita per la parallelizzazione è costituita dalle seguenti quattro funzioni:

- *compute_density_pressure*: come prima cosa calcola la densità delle particelle e, successivamente, usando il dato precedentemente ottenuto, ne calcola la pressione;
- *compute_forces*: utilizza i valori aggiornati di densità e pressione per calcolare le forze che agiscono tra particelle vicine. A queste si aggiunge poi la forza di gravità che tende a spingere le particelle verso il basso;
- *integrate*: utilizza i valori aggiornati delle forze per determinare la nuova posizione e la nuova velocità;
- *avg_velocities*: calcola la velocità media delle particelle.

Versione OpenMP

La prima versione è stata realizzata sfruttando il parallelismo a memoria condivisa fornito da OpenMP.
Inizialmente è stata fatta un'analisi delle possibili funzioni da parallelizzare al fine di trovare possibili dipendenze e quindi capire la strategia più adatta. Ricordiamo infatti il "*Teorema fondamentale della dipendenza*": qualsiasi trasformazione di riordino che preservi ogni dipendenza di un programma preserva il significato di quel programma.
La prima osservazione da fare riguarda le similarità tra le prime due funzioni (*compute_density_pressure* e *compute_forces*). Entrambe, infatti, seguono il seguente pattern: iterano su ogni coppia di particelle ottenendo valori intermedi utilizzati poi per aggiornare le proprietà delle singole particelle.
La strategia adottata per la loro parallelizzazione è stata la seguente:

1. Sono stati creati degli array temporanei per densità e forze;
2. È stata effettuata una rifattorizzazione dei due cicli for annidati necessaria per poter applicare, insieme al pragma *#omp for,* il pattern *reduce* attraverso l'apposita clausola *reduction* al fine di calcolare i valori di densità e delle forze. Ciò è stato necessario per una corretta parallelizzazione dato che in ogni iterazione viene incrementata la stessa variabile. È stata poi utilizzata anche la clausola *collapse* che permette così di collassare i due cicli in una grande iterazione suddivisa poi tra i thread;
3. Infine, è stato aggiunto un ciclo for per aggiornare le proprietà delle particelle tramite i valori accumulati all'interno degli array. A questo è stato applicato un semplice pragma *#omp for.*

È poi importante parlare della gestione del pool di thread. Questa dipende strettamente dall'implementazione di OpenMP. Dalle ricerche effettuate, senza però trovare documentazioni ufficiali, già le implementazioni di GCC, ICC e MSVC dovrebbero effettuare una gestione efficiente delle risorse.
Data la mancanza di documentazioni ufficiali e con l'obiettivo di gestire in maniera efficiente il pool di thread a prescindere dall'implementazione di OpenMP è stato scelto di inserire, nelle prime due funzioni, tutti i cicli all'interno di un blocco strutturato a cui è stato applicato un *#pragma omp parallel*. Così facendo viene creato un pool di thread all'inizio del blocco e, successivamente, ogni qual volta venga chiamato un *#pragma omp for,* verranno utilizzati thread appartenenti ad esso.
Le successive due funzioni sono invece più semplici da parallelizzare. Per quanto riguarda *integrate* è stato utilizzato un semplice *#pragma omp parallel for* mentre per *avg_velocities* è stato utilizzato un *#pragma omp parallel for* con una clausola *reduction* con operatore di somma sulla variabile result visto appunto il suo incremento ogni iterazione.
Questa, come successo precedentemente per il calcolo della densità e delle forze, è la situazione ideale in cui applicare il *pattern reduce* al fine di evitare l'utilizzo di istruzioni atomiche o eventuali sezioni critiche.

Versione MPI

La seconda versione è stata realizzata sfruttando il parallelismo a memoria distribuita fornito da MPI.
Come prima cosa, per facilitare l'invio e la ricezione delle particelle, è stato definito un tipo di dato derivato. Visto che ogni particella è descritta da una struttura contenente otto float, il tipo di dato scelto è stato il vettore di dati contiguo (*type contiguous).*
È da sottolineare che questa è stata una scelta puramente implementativa al fine di poter inviare e ricevere particelle in maniera semplice e comoda. Infatti, le primitive messe a disposizione da MPI già consentono l'invio di dati contigui.
Inizialmente, dopo l'inizializzazione delle particelle a carico del processo master (quello con id uguale a zero), attraverso una primitiva di comunicazione collettiva broadcast, vengono inviati a tutti i processi l'intero array delle particelle e la lunghezza dell'array stesso. Quest'ultimo è necessario in quanto, essendo l'array delle particelle allocato dinamicamente, non è possibile ottenere la sua dimensione in altro modo.
Successivamente è stata effettuato il partizionamento calcolando gli array contenenti spostamento e dimensione di ogni partizione in modo tale da avere sbilanciamento massimo pari a uno. Questi valori sono fondamentali per consentire un corretto funzionamento con dimensioni arbitrarie del dominio.
Ogni processo modifica quindi solamente le particelle appartenenti alla propria partizione, ottenuta appunto utilizzando i rispettivi indici.
Per aggiornare l'array delle particelle di ogni processo vengono utilizzate delle *Allgatherv*, delle primitive di comunicazione collettiva che consentono di raccogliere tutte le partizioni e, successivamente, inviare l'array completo a tutti i processi. Da sottolineare che non vengono utilizzate delle *Allgather* normali bensì delle *Allgatherv* che, a differenza delle prime, consentono la raccolta di partizioni di dimensioni diverse tra loro. Anche questo è un punto fondamentale per consentire il corretto funzionamento con una dimensione arbitraria del dominio.
Analizzando in maggior dettaglio l'implementazione si può notare come i risultati locali siano raccolti successivamente a:

- *compute_density_pressure*: la successiva *compute_forces* richiede i valori delle proprietà aggiornati;
- *integrate*: quest'ultima richiede la conoscenza solo delle particelle appartenenti alla propria partizione e quindi non è necessario raccogliere i dati computati dagli altri

processi prima della sua esecuzione.
Infine, per quanto riguarda il calcolo della velocità media, viene eseguito il calcolo locale da ogni processo sulla propria partizione e poi, attraverso una *Allreduce* con operatore di somma, vengono uniti i risultati. In questo caso, nonostante al fine solamente dell'output sia sufficiente una *Reduce* su un solo processo senza l'invio del risultato a tutti, si è preferito mantenere un'esecuzione uniforme e quindi distribuire comunque il risultato a tutti i processi.

Risultati e Conclusioni

Tutti i test sono stati eseguiti sul server utilizzato per il corso. Le sue specifiche sono le seguenti:

- Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz
- RAM 64Gb

Il processore utilizzato non è dotato di Hyper-Threading.

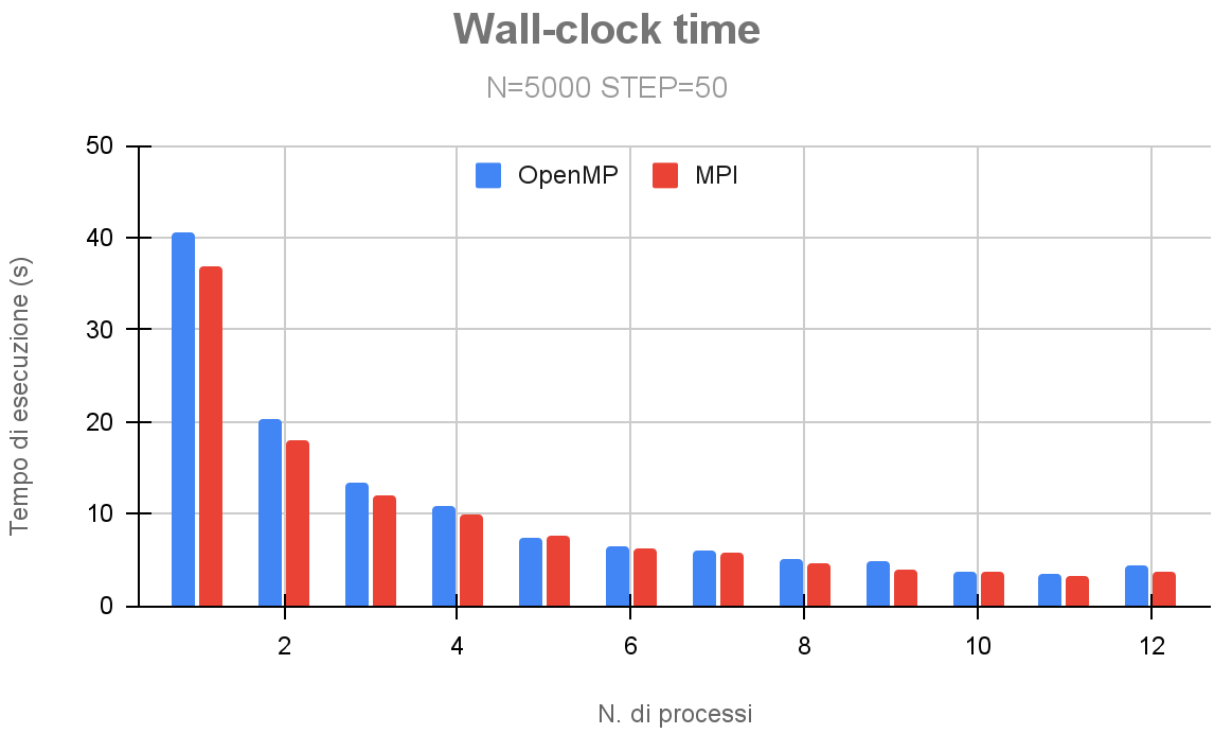

Figura 1: Tempi di esecuzione al variare del numero di processi (N=5000 e STEP=50)

In figura 1 possiamo notare il confronto dei tempi di esecuzione tra le due versioni all'aumentare del numero di processi.
È possibile notare come, soprattutto con un numero di processi piccolo, la versione realizzata sfruttando il parallelismo a memoria distribuita con MPI abbia prestazioni migliori. All'aumentare del numero di processi si può vedere come le differenze vadano a diminuire.
È importante anche porre attenzione all'esecuzione con 12 core, ossia il numero totale dei core disponibili sulla macchina sulla quale sono stati eseguiti i test. È possibile notare infatti come questa esecuzione abbia prestazioni peggiori rispetto a quella con 11 core. Questo è dato dal fatto che sulla macchina sono attivi anche altri processi.

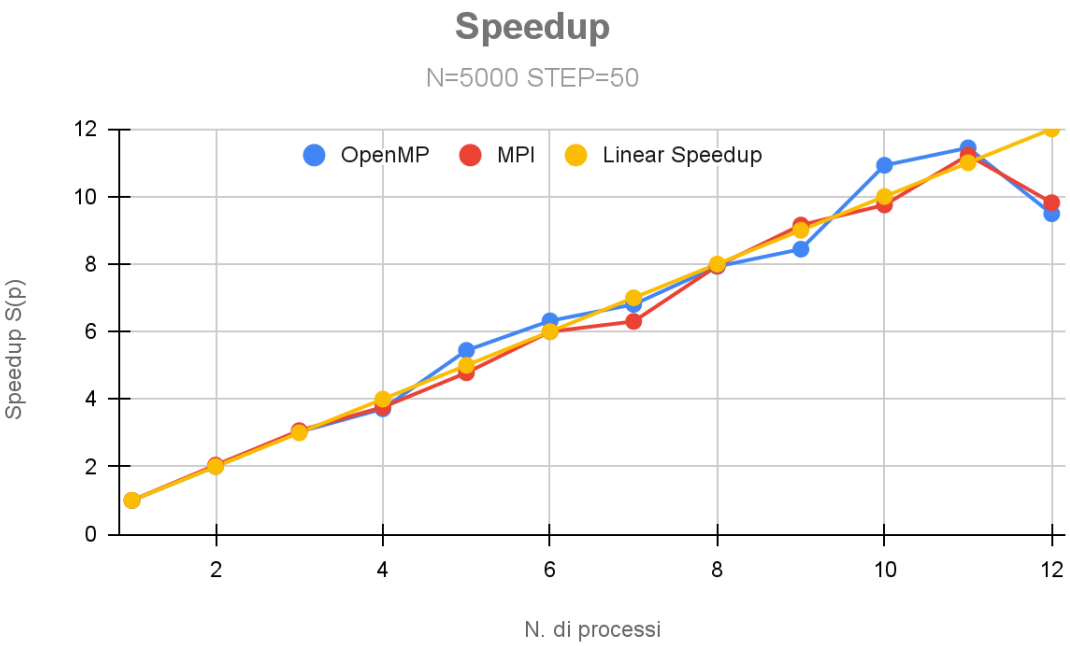

Figura 2: Speedup

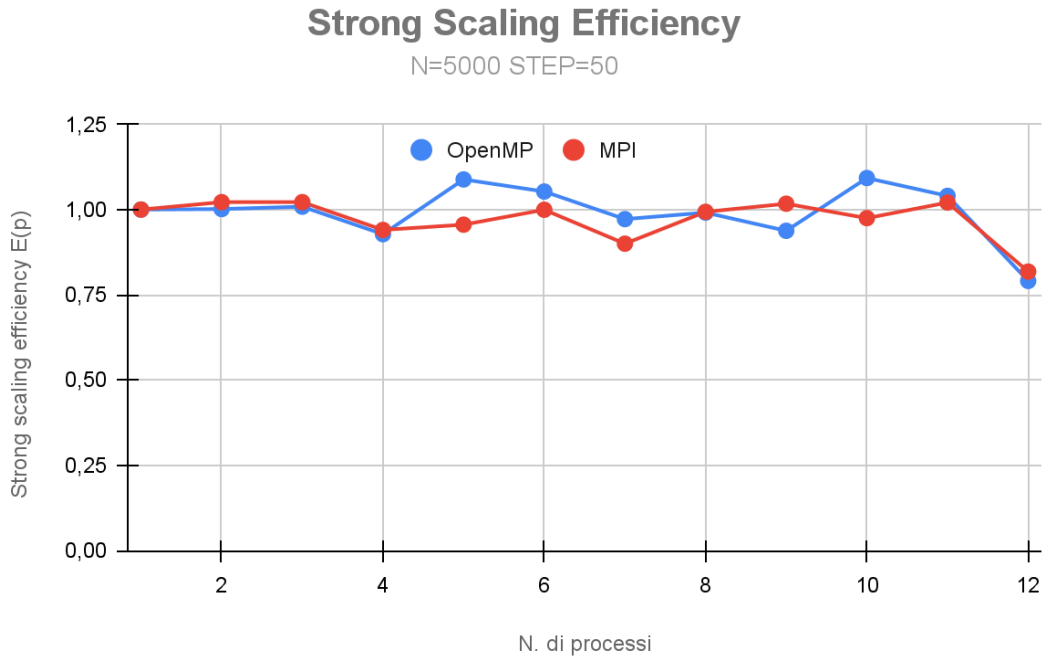

Figura 3: Strong scaling efficiency (N=5000 e STEP=50)

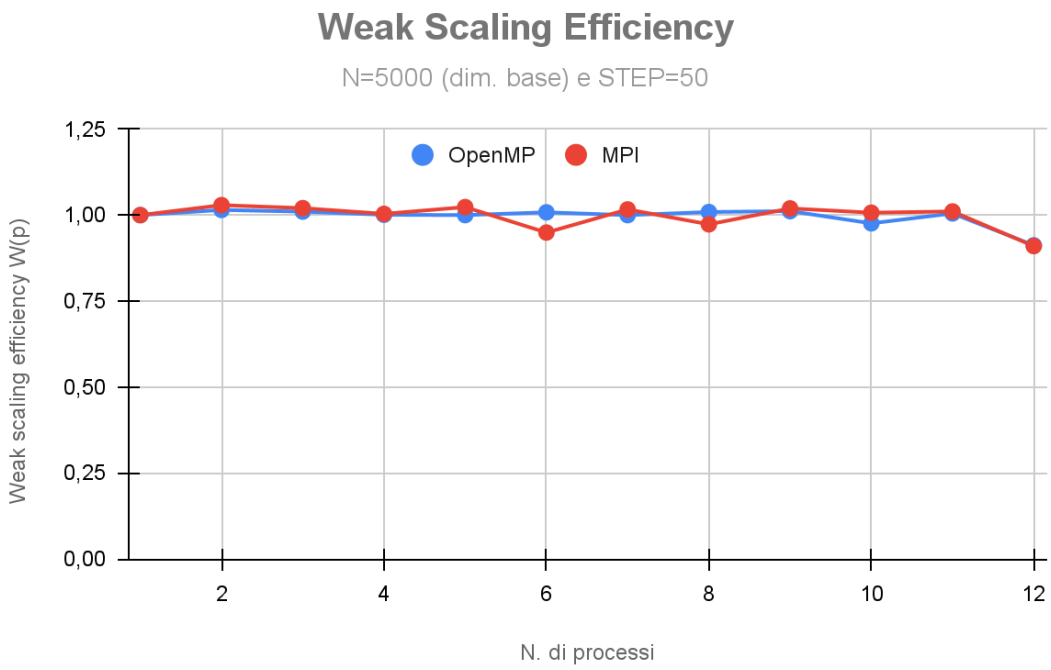

Figure 4: Weak scaling efficiency (N=5000 (dim. base) e STEP=50)

Dai precedenti grafici si può notare come lo speedup della versione realizzata con OpenMP, soprattutto all'aumentare del numero di processi, sia leggermente migliore rispetto a quello della versione realizzata utilizzando MPI.
Da evidenziare anche il grafico sulla strong scaling efficiency (figura 3): si può vedere come la versione OpenMP benefici maggiormente dell'aumentare del numero di processi. Questo è confermato dai tempi di esecuzione (figura 1): inizialmente la versione MPI ha tempi di esecuzione inferiori ma all'aumentare dei processi questi vanno quasi ad eguagliarsi.
Ciò è dato dal fatto che all'aumentare del numero di processi, sul modello di programmazione a memoria distribuita, saranno necessarie più comunicazioni che, come sappiamo, incidono molto sui tempi di esecuzione.
Riguardo alla versione MPI si può discutere sicuramente anche di possibili migliorie. Infatti, come detto precedentemente, nella versione realizzata, è stato utilizzato un tipo di dato derivato che consente, in maniera agile, l'invio e la ricezione di tutte le particelle. Così facendo però, ogni volta, vengono inviate anche proprietà delle particelle che non sono state aggiornate.
Nel caso in cui si cercasse la massima efficienza e si puntasse al raggiungimento di tempi di esecuzione inferiori sicuramente si potrebbe procedere con il seguente approccio:
non utilizzare un tipo di dato derivato e inviare, tra i processi, solamente le proprietà aggiornate. Ciò permetterebbe sicuramente di ridurre le comunicazioni effettuate e quindi ottenere tempi di esecuzione inferiori.
Per quanto riguarda gli ottimi tempi ottenuti con la versione realizzata con OpenMP questo è dato dall'utilizzo, all'interno delle funzioni, di thread appartenenti allo stesso pool evitando quindi possibili overhead dati da molteplici creazioni e distruzioni.

Infine, aggiungo che, nel caso si volesse provare ad ottenere un miglioramento di prestazioni ulteriore, si potrebbe pensare all'utilizzo di istruzioni *SIMD*. Si potrebbe, ad esempio, sfruttare la tecnica di *loop unrolling* ed i *vector data type* (essendo consapevole che non sono standard e che dipendono dal compilatore) al fine di poter eseguire più operazioni in parallelo e ridurre il numero di volte in cui le condizioni del ciclo sono valutate. È da sottolineare però che, prima della scelta definitiva, andrebbero fatte prove sperimentali in quanto le funzioni comprendono diversi branch. Andrebbe verificato che la scelta non porti all'esecuzione di molte istruzioni non necessarie. Per la verifica si potrebbe controllare quante volte, durante le esecuzioni, le condizioni dei branch sono vere e quindi quante volte le istruzioni contenute vengono eseguite. Nel caso risulti conveniente l'utilizzo di istruzioni *SIMD,* per trattare i branch, si potrebbe utilizzare la tecnica del *selection and masking*.

Riferimenti bibliografici

[1] Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA '03). Eurographics Association, Goslar, DEU, 154–159. https://dl.acm.org/doi/10.5555/846276.846298

[2] https://github.com/cerrno/mueller-sph