Progetto di High Performance Computing 2022/2023

Fabio Veroli, matr. 0000970669

01/08/2023

Introduzione

Per la realizzazione del progetto, sono state implementate due versioni parallele dell'algoritmo **Smoothed Particle Hydrodynamics** (**SPH**). Una versione è stata sviluppata sfruttando il parallelismo a memoria condivisa fornito da **OpenMP**, mentre l'altra versione sfruttando il parallelismo a memoria distribuita fornito da **MPI**. Le funzioni chiave dell'algoritmo, `compute_density_pressure()`, `compute_forces()`, `integrate()` e `avg_velocities()` sono state parallelizzate per entrambe le versioni utilizzando i meccanismi specifici offerti dalle due tecnologie impiegate.
I due programmi sono stati compilati ed eseguiti sul server di laboratorio equipaggiato con un processore Xeon E5-2603 con 12 core a 1.70 GHz, 64GB di RAM e 3 GPU Nvidia GTX 1070. Il server è stato utilizzato anche per condurre i test al fine di valutare le prestazioni dei programmi. In particolare, per ciascun programma, sono stati effettuati test utilizzando un numero di core compreso tra 1 e 12, eseguendo 5 misurazioni del tempo di esecuzione per ciascuna configurazione, utilizzando 7500 particelle e 100 step. Dai 5 valori ottenuti per ogni numero di core, è stato selezionato il tempo di esecuzione minimo come valore rappresentativo. Questa scelta è stata preferita rispetto all'utilizzo della media o della mediana dei valori, poiché ciò ha consentito di evitare l'inclusione di valori non veritieri causati da fluttuazioni nella misurazione dei tempi di esecuzione, che potrebbero essere influenzate da fattori esterni al programma (approccio suggerito da [1] capitolo 2.6.4). Utilizzando i tempi di esecuzione ottenuti, sono stati poi calcolati lo *Speedup*, la *Strong Scaling Efficiency* e la *Weak Scaling Efficiency* per valutare le prestazioni dei programmi e analizzare come si comportano in termini di scalabilità.

Versione OpenMP

Nella parallelizzazione utilizzando OpenMP sono state adottate diverse clausole per le quattro funzioni, tenendo conto delle dipendenze sui dati presenti nei cicli. Inizialmente, è stata eseguita un'analisi della struttura dei cicli presenti in ogni funzione, al fine di evitare accessi concorrenti alla stessa risorsa o dipendenze tra le iterazioni dei cicli.
Per quanto riguarda la funzione `integrate()`, questa si presta ad essere parallelizzata in modo *embarrassingly parallel*, poiché le sue iterazioni sono indipendenti tra loro e possono essere eseguite in qualsiasi ordine. Pertanto, è stata suddivisa equamente tra i vari thread.
Nella funzione `avg_velocities()`, invece, è presente un ciclo in cui ogni iterazione dipende dalla precedente, rendendo impossibile una parallelizzazione diretta. Tuttavia, è stato possibile parallelizzarla utilizzando il pattern **reduction** e la relativa clausola fornita da OpenMP.
Le funzioni `compute_density_pressure()` e `compute_forces()` presentano invece una struttura simile, entrambe con due cicli annidati. Il ciclo esterno può essere parallelizzato in maniera banale, in quanto le interazioni riguardano particelle diverse e non dipendono l'una dall'altra. Mentre il ciclo interno aggiorna le caratteristiche di una particella basandosi sui valori di tutte le altre e quindi non può essere parallelizzato in maniera banale.
Per la parallelizzazione delle due funzioni sono state utilizzate due strategie differenti. Nella funzione `compute_density_pressure()`, i due cicli sono stati parallelizzati collassandoli in un'unica iterazione, utilizzando la clausola `collapse`, ed utilizzando la clausola `atomic` per evitare *race condition*. Successivamente, è stato aggiunto un ciclo aggiuntivo parallelizzato in modo *embarrassingly parallel* per calcolare la pressione di ogni particella. Nel caso della funzione `compute_forces()`, si è deciso invece di parallelizzare soltanto il ciclo esterno, dividendo l'interazione tra i vari thread. Questa soluzione ha mostrato prestazio-

ni migliori rispetto a quella adottata per la funzione precedente, poiché probabilmente evita un overhead maggiore causato dalla sincronizzazione dei thread necessaria per evitare le *race condition*.

Analisi prestazioni versione OpenMP

In seguito ai test effettuati sui tempi di esecuzione della versione parallela OpenMP, è stato calcolato lo **Speedup relativo**, definito come $S(p) = \frac{T_{parallelo}(1)}{T_{parallelo}(p)}$, dove p rappresenta il numero di thread utilizzati nella versione parallela. Dalla Tabella 1 emerge che lo Speedup ottenuto è sub-lineare. Questa tendenza è evidenziata in modo più chiaro nel grafico riportato in Figura 1, dove si osserva un andamento pressoché lineare per valori di p inferiori a 11, mentre per p uguale a 12 si osserva un cambiamento di tendenza. Questo andamento potrebbe essere influenzato da vari fattori. Innanzitutto, l'overhead necessario per la creazione e la sincronizzazione dei thread aumenta all'aumentare del numero dei thread utilizzati, risultando particolarmente rilevante nella funzione `compute_density_pressure()`, in cui i thread devono sincronizzarsi per evitare *race condition*. Un altro fattore è la presenza di alcune parti di codice eseguite in modo seriale, che come stabilito dalla legge di Amdahl, pongono un limite superiore al massimo Speedup ottenibile. Queste sezioni seriali includono la fase di sincronizzazione nella funzione `compute_density_pressure()` e il ciclo interno della funzione `compute_forces()`, in cui ogni thread scorre tutte le particelle per calcolare le caratteristiche della particella corrente, comportando un degrado delle prestazioni all'aumentare del numero di particelle.

OMP Speedup & Strong Scaling Efficiency												
Num. Thread	1	2	3	4	5	6	7	8	9	10	11	12
Tempo	172,970	86,910	57,994	43,554	34,832	29,059	24,921	21,801	19,413	17,456	15,999	16,137
Speedup	1,000	1,990	2,983	3,971	4,966	5,952	6,941	7,934	8,910	9,909	10,811	10,719
Strong Scaling Efficiency	1,000	0,995	0,994	0,993	0,993	0,992	0,992	0,992	0,990	0,991	0,983	0,893

Tabella 1: Speedup e Strong Scaling Efficiency versione parallela OpenMP

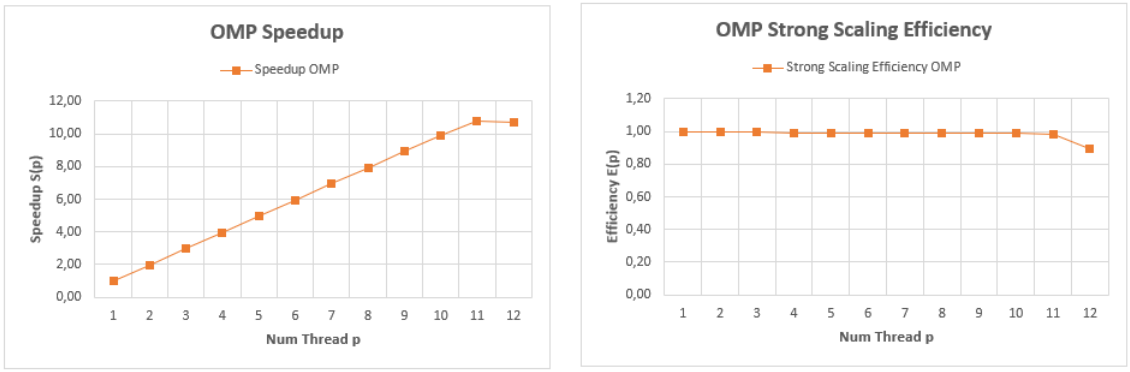


Figura 1: Grafico Speedup e Strong Scaling Efficiency per 1-12 thread nella versione OpenMP

Per valutare la scalabilità del programma, sono stati presi in considerazione la Strong Scaling Efficiency e la Weak Scaling Efficiency.
Come mostrato nella Tabella 1 e nella Figura 1, aumentando il numero di thread e mantenendo costante il numero di particelle, il tempo di esecuzione totale si riduce, portando ad una **Strong Scaling Efficiency** all'incirca sempre maggiore di 0.9. Anche in questo caso, come per lo Speedup, ci si aspetta che per un elevato numero di thread l'efficienza diminuisca a causa dell'overhead dovuto alla sincronizzazione.
Per la valutazione della **Weak Scaling Efficiency**, il numero di particelle è stato aumentato secondo la

formula: $n_p = const \cdot \sqrt{p}$, dove p rappresenta il numero di thread e const è stato impostato a 2165. Come mostrato nella Tabella 2 e nella Figura 2, all'aumentare del numero di thread e del numero di particelle, il tempo di esecuzione rimane pressoché invariato, come indicato dai valori ottenuti per la Weak Scaling Efficiency, che risultano sempre maggiori di 0.9 ad indicare la scalabilità del programma.

OMP Weak Scaling Efficiency												
Num. Thread	1	2	3	4	5	6	7	8	9	10	11	12
Tempo	14,556	14,808	14,662	14,641	14,639	14,624	14,622	14,611	14,577	14,588	14,669	16,119
Weak Scaling Efficiency	1,000	0,983	0,993	0,994	0,994	0,995	0,995	0,996	0,999	0,998	0,992	0,903

Tabella 2: Weak Scaling Efficiency versione OpenMP

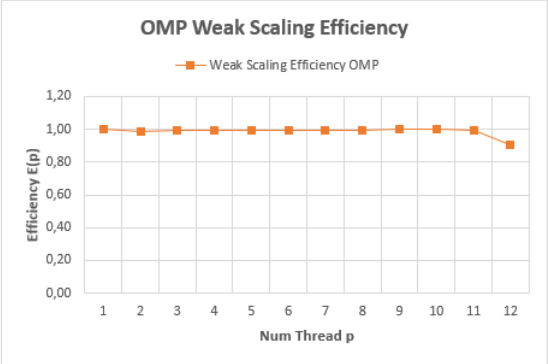


Figura 2: Grafico Weak Scaling Efficiency versione OpenMP

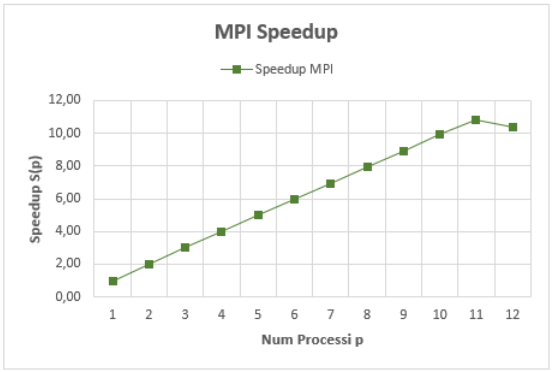
Versione MPI

Nella parallelizzazione utilizzando MPI, si è scelto di ripartire le particelle tra i vari processi, in modo che ogni processo sia responsabile del calcolo di una sotto-porzione delle particelle originarie. Questo approccio è stato reso possibile grazie all'utilizzo delle funzioni di comunicazione collettiva offerte da MPI.
Le particelle vengono inizialmente ripartite tra i vari processi utilizzando la funzione `MPI_Scatterv()`, che consente di suddividere le particelle tra i processi, utilizzando blocchi di dati di lunghezza diversa se il numero delle particelle non è divisibile per il numero di processi. Ogni processo esegue poi le quattro funzioni utilizzando la propria sotto-porzione di particelle. Fra l'esecuzione di una funzione e quella successiva è necessario ridistribuire i valori aggiornati di ogni sotto-porzione di particelle a tutti i processi, per questo scopo è stata utilizzata la funzione `MPI_Allgatherv()`. Ogni processo, quindi, oltre a mantenere un vettore contenete la propria sotto-porzione di particelle, avrà anche un vettore contenente tutte le particelle, che deve essere aggiornato dopo ogni funzione.
Per la funzione `avg_velocities()`, ciascun processo calcola la velocità per la propria sotto-porzione di particelle e successivamente viene utilizzata la funzione `MPI_Reduce()` per ottenere il risultato finale aggregando i risultati locali di tutte le sotto-porzioni.
Al fine di consentire la corretta comunicazione tra i processi, è stato necessario definire un nuovo tipo di dato per rappresentare una particella. Questo nuovo tipo di dato, creato utilizzando la funzione `MPI_Type_contiguous()`, è composto da un numero specifico di elementi float, corrispondente ai campi della struttura che rappresenta una particella. In questo modo, è stato possibile definire un formato standardizzato per le comunicazioni relative alle particelle, garantendo la coerenza dei dati scambiati tra i processi.

Analisi prestazioni versione MPI

MPI Speedup & Strong Scaling Efficiency												
Num. Processi	1	2	3	4	5	6	7	8	9	10	11	12
Tempo	167,946	84,140	56,155	42,101	33,728	28,136	24,154	21,151	18,823	16,946	15,525	16,233
Speedup	1,000	1,996	2,991	3,989	4,979	5,969	6,953	7,940	8,922	9,911	10,818	10,346
Strong Scaling Efficiency	1,000	0,998	0,997	0,997	0,996	0,995	0,993	0,993	0,991	0,991	0,983	0,862

Tabella 3: Speedup e Strong Scaling Efficiency versione parallela MPI



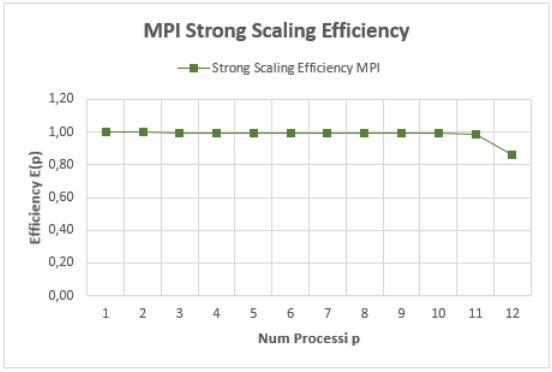


Figura 3: Grafico Speedup e Strong Scaling Efficiency per 1-12 processi nella versione MPI

Come mostrato dalla Tabella 3, anche nella parallelizzazione effettuata con MPI, come per quella con OpenMP, si è ottenuto uno **Speedup** sub-lineare. Anche in questo caso, come mostrato nel grafico riportato in Figura 3, lo Speedup mostra un andamento pressoché lineare per un numero di processi inferiore a 11, per poi avere un cambiamento di tendenza utilizzando 12 processi. In questo caso, l'overhead introdotto dalla versione parallela dipende dall'inizializzazione e dall'utilizzo delle funzioni di comunicazione. Questo overhead aumenta all'aumentare del numero di processi che necessitano di comunicare; è inoltre influenzato dalla necessità di far comunicare tra loro i processi dopo ogni funzione per ridistribuire le particelle aggiornate.

MPI Weak Scaling Efficiency												
Num. Processo	1	2	3	4	5	6	7	8	9	10	11	12
Tempo	14,153	14,119	14,111	14,107	14,127	14,130	14,130	14,144	14,155	14,152	14,264	16,234
Weak Scaling Efficiency	1,000	1,002	1,003	1,003	1,002	1,002	1,002	1,001	1,000	1,000	0,992	0,872

Tabella 4: Weak Scaling Efficiency versione MPI

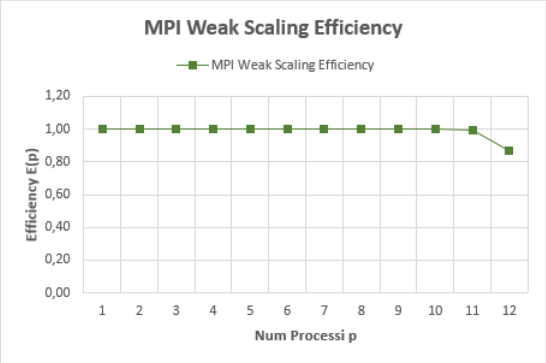


Figura 4: Grafico Weak Scaling Efficiency versione MPI

Per la valutazione della **Strong Scaling Efficiency** e **Weak Scaling Efficiency**, sono state seguite le stesse modalità illustrate per la versione OpenMP.
Come mostrato nella Figura 3 e nella Tabella 3, i valori della **Strong Scaling Efficiency** dimostrano una buona scalabilità del programma per un numero ridotti di processi, mentre l'efficienza peggiora all'aumentare del numero di processi. Questo risultato potrebbe essere una conseguenza della diminuzione della granularità della partizione delle particelle, che causa uno sbilanciamento tra comunicazione e computazione, rendendo la comunicazione tra processi più onerosa della computazione.
Anche i risultati della **Weak Scaling Efficiency**, come mostrato nella Figura 4 e nella Tabella 4, suggeriscono una buona scalabilità per un numero ridotto di processi, ma questa peggiora all'aumentare del numero di processi. In questo caso, la diminuzione dell'efficienza è causata dalla struttura del programma; sebbene all'aumentare del numero di processi e della dimensione delle particelle queste siano comunque equamente distribuite tra i processi; quindi, il carico di lavoro rimane bilanciato. Tuttavia, ogni processo deve mantenere ed elaborare anche l'intero vettore delle particelle, causando un aumento dei costi di computazione, specialmente nelle funzioni `compute_density_pressure()` e `compute_forces()`, in cui ogni processo deve scorrere l'intero vettore delle particelle. Questo comporta anche un aumento dei costi di comunicazione, poiché è necessario alla fine di ogni funzione ridistribuire l'intero vettore delle particelle aggiornate. Questi fattori causano quindi una scarsa scalabilità del programma all'aumentare del numero di particelle.

Conclusioni

I risultati ottenuti per entrambe le versioni parallele mostrano buone prestazioni all'aumentare del numero di thread/processi. Lo **Speedup** sub-lineare, in particolare, evidenzia l'efficienza dei due programmi paralleli, ma allo stesso tempo mette in luce un'inversione di tendenza al crescere del numero di thread/processi. Come già discusso in precedenza, questo andamento è dovuto all'aumento dell'overhead dovuto alla sincronizzazione dei thread nella versione OpenMP e alla comunicazione dei processi nella versione MPI.
Dalle metriche utilizzate per misurare la scalabilità emerge che entrambi i programmi dimostrano una buona scalabilità per un numero ridotto di thread/processi, ma presentano un cambio di tendenza all'aumentare del numero di thread/processi.
Nel caso della **Strong Scaling Efficiency**, questa tendenza è dovuta dalla diminuzione della granularità della partizione delle particelle, che, nonostante riduca il lavoro per thread/processo, comporta una sincronizzazione/comunicazione tra i thread/processi più onerosa della computazione.
Per la **Weak Scaling Efficiency**, invece, questa tendenza è causata dalle sezioni seriali non parallelizzate all'interno di entrambi i programmi, che all'aumentare del numero di particelle causano un degrado delle prestazioni. In particolare, queste sezioni includono il ciclo interno della funzione `compute_-`

`density_pressure()` nella versione OpenMP e i cicli interni delle funzioni `compute_density_pressure()` e `compute_forces()` nella versione MPI.
In conclusione, i due programmi paralleli hanno dimostrato l'efficacia delle strategie di parallelizzazione con OpenMP e MPI nell'accelerare l'esecuzione dell'algoritmo SPH, in particolare trovando il giusto numero di thread/processi che massimizzi le prestazioni dei due programmi. In futuro, potrebbero essere valutate altre strategie per ottenere una parallelizzazione anche delle parti di codice che si sono dimostrate critiche per le soluzioni fornite. L'ottimizzazione di queste sezioni potrebbe portare ad ulteriori miglioramenti delle prestazioni complessive dei programmi paralleli.

Riferimenti bibliografici

[1] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 1 edition, 2011.