Progetto di *High Performance Computing* 2022/2023

Stefano Furi, matr. 000987426

20/02/2023

Introduzione

L'algoritmo SPH (*Smoothed Particle Hydrodynamics*) fornito in versione seriale, è stato parallelizzato mediante il parallelismo a memoria condivisa tramite **OpenMP** e il paradigma di programmazione a memoria distribuita con **MPI**. In particolare, sono state parallelizzate le 4 funzioni del calcolo dell'interazione tra le particelle, ovvero: `compute_density_pressure()`, `compute_forces()`, `integrate()` e `avg_velocities()`.

I test sono stati condotti sul Server di laboratorio dotato di due CPU Intel Xeon E5-2603, con un totale di 12 core, MacBook Pro dotato di chip Apple Silicon M1 Pro con 8 core effettivi (2 *power efficiency core* e 8 *performance core* con compilatore Homebrew GCC 12.2.0 (non è stata utilizzata la versione di `clang` nativa MacOS poiché OpenMP non è più supportato) e infine una Macchina virtuale su Macbook precedente, emulando processore Cortex-A72 (ARM) con 8 core su distribuzione Linux Debian 11, compilatore GCC (Debian 10.2.1-6).

Per ogni core, sono state effettuate 5 misurazioni, estraendone la media utilizzata per i calcoli sulle prestazioni. Tutti i test sono stati effettuati considerando i tempi di esecuzione da 1 a 8 core con dimensione dell'input pari a 5.000 particelle effettuando 100 step.

Versione OpenMP

La parallelizzazione attraverso **OpenMP** è stata effettuata in modo tale da non provocare *data race* e massimizzare le prestazioni attraverso le direttive OpenMP. In particlare, nelle funzioni `compute_forces()` e `compute_density_pressure()` la presenza di due cicli `for` annidati suggerisce l'uso di una direttiva `collapse`, in modo tale da parallelizzare entrambi i cicli efficientemente. Questa procedura comporta la nascita di una corsa critica, poiché l'aggiornamento delle proprietà della particella i-esima può essere effettuato da più processi contemporaneamente, ad un passo j-esimo diverso (Figura 1). Inoltre per poter applicare la clausola `collapse`, è stato necessario eliminare il codice tra l'intestazione dei due cicli `for` e il codice dopo la terminazione del ciclo interno. Questi requisiti hanno portato all'adozione di un array di appoggio, mediante il quale è possibile evitare l'aggiornamento delle proprietà delle particelle al termine del ciclo interno, su cui è possible applicare una riduzione al termine di entrambi

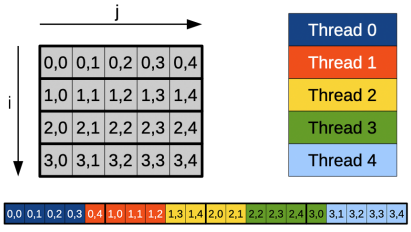


Figure 1: Suddivisione delle iterazioni di due cicli `for` annidati mediante la direttiva **OpenMP** `collapse(2)`.

i cicli evitando la corsa critica sopra descritta e massimizzando la cooperazione fra processi. All'interno di questo vettore quindi, sono contenuti i valori aggiornati delle proprietà, dove al termine della loro computazione è possibile assegnarli alle rispettive particelle mediante un ciclo parallelizzabile in modo *embarassingly parallel*. Per quanto riguarda le funzioni `integrate()` e `avg_velocities()`, la prima risulta parallelizzabile in modo *embarassingly parallel*, mentre la seconda rappresenta appieno il pattern di programmazione parallela *Reduction*.

Analisi versione Parallela

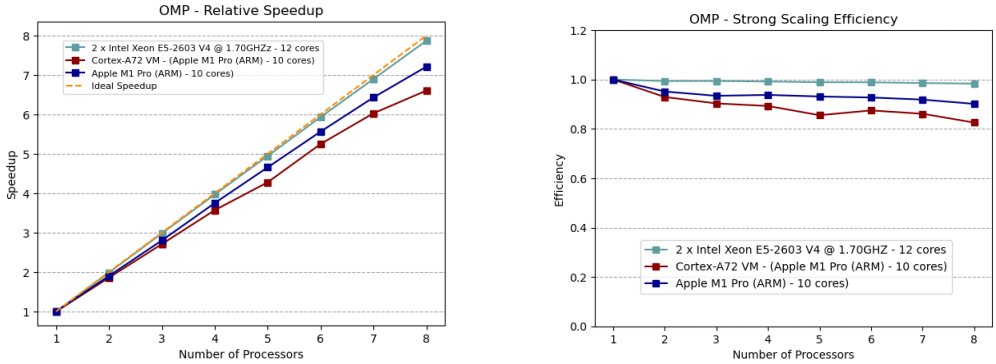
I risultati dei test sulla versione parallelizzata attraverso OpenMP sono riportati nella Tabella 1. Lo **Speedup** medio ottenuto nelle tre piattaforme mostra come (inevitabilmente) esista un certo *overhead* per le operazioni parallele. In particolare l'inizializzazione degli array di appoggio e l'aggiornamento dei valori calcolati per ogni particella provocano un peggioramento delle prestazioni, poiché ogni ciclo parallelizzato mediante una clausola `pragma omp for` prevede comunicazioni di sincronizzazione tra tutti i processi che possono incidere negativamente sui tempi d'esecuzione finale. Questo può essere facilmente osservabile dalla Fig. 2a dove la curva dello Speedup tende a crescere sempre meno linearmente all'aumento del numero di processi. Un altro fattore che potrebbe incidere nel peggioramento dello Speedup è la creazione e distruzione del pool di thread ad ogni funzione parallelizzata, provocando un aumento non indifferente dell'*overhead* complessivo.

Piattaforma	$T_{OMP,1}$	$T_{OMP,2}$	$T_{OMP,3}$	$T_{OMP,4}$	$T_{OMP,5}$	$T_{OMP,6}$	$T_{OMP,7}$	$T_{OMP,8}$	**Speedup**
Server (Intel Xeon)	78.68	39.59	26.38	19.83	15.91	13.26	11.40	10.00	7.86
Apple M1 Pro	37.00	18.84	12.45	9.30	7.47	6.26	5.38	4.76	7.76
Linux VM (M1)	24.87	13.12	8.67	6.67	5.25	4.66	3.96	3.49	7.12

Table 1: Tempi di esecuzione su differenti piattaforme della versione implementata attraverso **OpenMP**, per ogni core da 1 a 8. Lo speedup è calcolato come $S = \frac{T_{OMP,1}}{T_{OMP,8}}$

Per quanto riguarda la capacità di **scalabilità** del programma parallelo, oltre all'analisi dello Speedup, è necessario analizzare il grado di efficienza. Come mostrato dalla Fig. 2b, l'aumento del numero di processi mantenendo costante la dimensione del problema risulta in una ***Strong Scaling Efficiency*** media superiore allo 0.9. Questa misura dimostra come la parallelizzazione effettuata migliori l'esecuzione complessiva del programma anche con un alto numero di core.

Un ulteriore indicatore è dato dalla ***Weak Scaling Efficiency*** (mostrato in figura Fig. 3), la quale determina l'efficienza del programma aumentando il numero di processi operanti, ma allo stesso tempo aumentando in modo costante le unità di lavoro per ogni processo. È facilmente intuibile come la complessità dell'algoritmo sia asintoticamente pari a $\Theta(N^2)$, permettendo così il calcolo della funzione $f(n_p, p) = const = n_p^2/p$ ottenendo in questo modo un input pari a $n_p = \sqrt{p} \cdot N_0$, con N_0 scelto pari a 1.500. In questo modo ogni processo effettua la stessa quantità di lavoro all'aumentare del numero complessivo processi.



(a) (b)

Figure 2: **Speedup**(a) e **Strong Scaling Efficiency**(b) della versione parallela utilizzando OpenMP

Piattaforma	$T_{OMP,1}$	$T_{OMP,2}$	$T_{OMP,3}$	$T_{OMP,4}$	$T_{OMP,5}$	$T_{OMP,6}$	$T_{OMP,7}$	$T_{OMP,8}$
Server (Intel Xeon)	3.584	3.592	3.586	3.588	3.588	3.597	3.611	3.614
Apple M1 Pro	1.269	1.301	1.339	1.347	1.355	1.368	1.390	1.411
Linux VM (M1) 0.858	0.883	0.906	0.914	0.939	0.976	1.026	1.110	

Table 2: Tempi di esecuzione su differenti piattaforme della versione implementata attraverso **OpenMP**, per ogni core da 1 a 8, per i test sulla **Weak Scaling Efficiency**, aumentando il carico di lavoro secondo la formula $n_p = \sqrt{p} \cdot N_0$ con $N_0 = 1.500$.

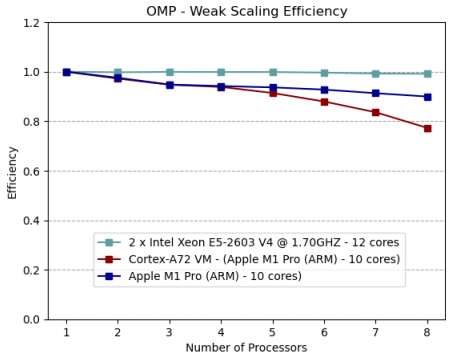


Figure 3: **Weak Scaling Efficiency** della versione parallela utilizzando OpenMP.

Versione MPI

Utilizzando il paradigma di parallelizzazione a memoria condivisa mediante l'uso di **MPI**, risulta di fondamentale importanza la comunicazione dell' aggiornamento dei valori delle particelle durante ogni passaggio della funzione `update`. La strategia utilizzata quindi, è stata quella di assegnare ad ogni processore una sotto-porzione del vettore delle particelle su cui lavorare (`w_particles`), ma allo stesso tempo mantenere l'intero array per poter leggere i valori di tutte le particelle. In questo modo viene favorito l'uso delle funzioni `MPI_Scatter` e `MPI_Gather`. Risulta però evidente come non sia possibile condividere il vettore in modo banale, poiché composto da strutture del tipo `particle_t`. È necessario quindi definire un nuovo tipo di dato

MPI, nello specifico un *derived datatype* di tipo ***Struct*** che permetta la condivisione delle particelle (sarebbe stato sufficiente un *datatype* di tipo *contiguous* siccome il tipo di dati dei campi delle particelle era il medesimo).

Come detto in precedenza, ad ogni funzione chiamata all'interno della procedura `update`, è necessaria una comunicazione a tutti i processi dei valori aggiornati delle particelle su cui si ha lavorato: ciò viene ottenuto mediante l'uso della funzione `MPI_Bcast` subito dopo aver riunito tutte le sotto porzioni del vettore di particelle attraverso `MPI_Gather`. Questa configurazione di operazioni viene ulteriormente semplificata dall'uso della funzione `MPI_Allgather` la quale effettua le due operazioni sopra descritte sperabilmente in modo più efficiente.

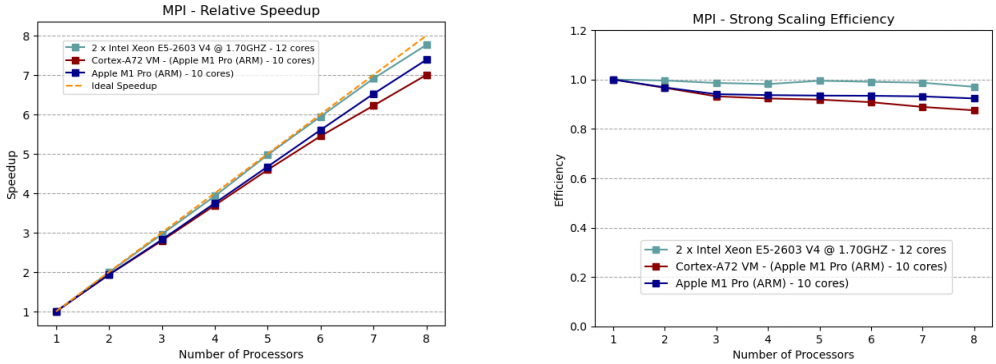
Per poter rendere il codice valido anche per dimensioni dell'array di particelle non divisibile per il numero di processi, sono state utilizzate le varianti `MPI_Scatterv` e `MPI_Allgatherv`.

Analisi versione Parallela

I risultati dei test sulla versione parallelizzata attraverso MPI sono riportati nella Tabella 3. Essi innanzitutto mostrano una graduale riduzione dei tempi d'esecuzione, e in particlare da un buon **Speedup** complessivo (Fig. 4a). È facilmente intuibile, in base alla logica del programma parallelo, che esiste un *overhead* non indifferente per la comunicaizone dell'array aggiornato delle particelle attraverso un operazione di *broadcast* a tutti i processi. Aumentando il numero di processi quindi, è inevitabile un peggioramento complessivo dello Speedup. Nonostante ciò, nel programma si è massimizzato l'uso delle comunicazioni collettive MPI, in modo tale da evitare situazioni di *deadlock* o *busy waiting*, sfruttando anche una possibile ottimizzazione delle operazioni di comunicazione da parte del compilatore. La *Strong Scaling efficiency* suggerisce inoltre che nonostante questo *overhead*, all'aumentare dei processi ma mantenedo invariata la dimensione del carico di lavoro, l'efficienza complessiva media (delle 3 piattaforme analizzate), rimane al di sopra di 0.95, mostrando quindi un buon grado di scalabilità. È interessante notare però come la curva mostrata nella Fig. 4b mostri un picco positivo quando sono impiegati 5 processi MPI. Considerando la dimensione dell'input pari a 5.000, potrebbe essere che i processi abbiano beneficiato di una distribuzione del carico perfettamente bilanciata, sfruttando anche una possibile ottimizzazione da parte della comunicazione collettiva.

Piattaforma	$T_{MPI,1}$	$T_{MPI,2}$	$T_{MPI,3}$	$T_{MPI,4}$	$T_{MPI,5}$	$T_{MPI,6}$	$T_{MPI,7}$	$T_{MPI,8}$	**Speedup**
Server (Intel Xeon)	74.99	37.65	25.35	19.10	15.07	12.61	10.85	9.66	7.76
Apple M1 Pro	16.20	8.37	5.75	4.32	3.46	2.89	2.49	2.18	7.40
Linux VM (M1)	18.67	9.64	6.63	5.01	4.05	3.41	2.98	2.66	7.01

Table 3: Tempi di esecuzione su differenti piattaforme della versione implementata attraverso **MPI**, per ogni core da 1 a 8. Lo speedup è calcolato come $S = \frac{T_{MPI,1}}{T_{MPI,8}}$



(a) (b)

Figure 4: **Speedup**(a) e **Strong Scaling Efficiency**(b) della versione parallela utilizzando MPI

Mediante l'analisi della **Weak Scaling Efficiency** attraverso la Tabella 4 è possibile ampliare l'analisi della scalabilità, mostrando come mantenendo costante le unità di lavoro come per la versione OpenMP ($n_p = \sqrt{p} \cdot N_0$) al variare del numero di processi p, risulti una buona efficienza complessiva (Fig 5).

Gli studi sull'efficienza confermano come all'aumentare del numero di processi il *setup* e le comunicazioni MPI diventino via via più onerose, sia da un punto di vista di lavoro individuale per ogni processo (*weak scaling*), sia da un punto di vista di diminuzione della granularità del problema e il relativo sbilanciamento tra computazione e comunicazioni nel caso di studio dello *Speedup* e *Strong Scaling Efficiency*.

Piattaforma	$T_{MPI,1}$	$T_{MPI,2}$	$T_{MPI,3}$	$T_{MPI,4}$	$T_{MPI,5}$	$T_{MPI,6}$	$T_{MPI,7}$	$T_{MPI,8}$
Server (Intel Xeon)	3.410	3.417	3.423	3.411	3.415	3.451	3.420	3.415
Apple M1 Pro	0.731	0.747	0.766	0.767	0.767	0.770	0.779	0.783
Linux VM (M1)	0.838	0.861	0.893	0.899	0.907	0.916	0.937	0.953

Table 4: Tempi di esecuzione su differenti piattaforme della versione implementata attraverso **MPI**, per ogni core da 1 a 8, per i test sulla **Weak Scaling Efficiency**, aumentando il carico di lavoro secondo la formula $n_p = \sqrt{p} \cdot N_0$ con $N_0 = 1.500$.

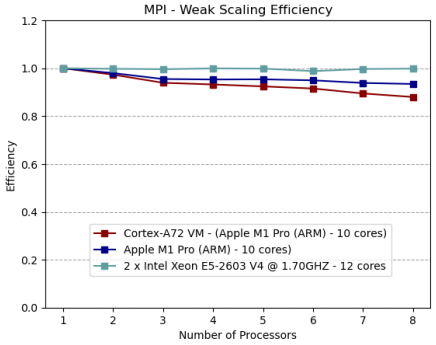


Figure 5: **Weak Scaling Efficiency** della versione parallela utilizzando MPI.

Conclusioni

Le analisi effettuate mettono in luce come entrambe le versioni presentino buoni risultati in termini di performance e scalabilità. Per quanto riguarda la versione parallelizzata mediante OpenMP, la direttiva `collapse` sicuramente complica la gestione della parallelizzazione e in generale rende il codice meno comprensibile. Le opportunità di parallelizzazione erano molteplici (e.g. parallelizzare uno dei due cicli e in caso di corsa critica effettuare *reduction*), ma si è optato per la versione che utilizza più direttive OpenMP massimizzando la cooperazione fra processi e l'equa divisione del carico di lavoro.

D'altra parte, la versione che utilizza MPI è stata implementata in modo tale da sfruttare al massimo l'utilizzo di comunicazioni collettive tra tutti i processi, anche se questo approccio (come visto precedentemente) può provocare un aumento dell'*overhead* complessivo all'aumentare del numero di processi MPI.

Le analisi effettuate mirano ad illustrare, oltre al grado di ottimizzazione mediante le relative versioni parallele del codice seriale, come l'esecuzione su differenti piattaforme possa avere caratteristiche differenti. In particolare, le prestazioni dei processori Intel Xeon presentano una buona scalabilità ma peggiori tempi d'esecuzione rispetto ad un processore non server-oriented come la CPU Apple M1, la quale invece presenta peggiori capacità di scalabilità. Riguardo quest'ultima, era prevedibile che la *Strong* e *Weak Scaling Efficiency* all'interno della macchina virtuale presentino risultati peggiori rispetto all'esecuzione nativa, ma è interessante notare come i tempi d'esecuzione per la versione OpenMP diano risultati migliori rispetto alla versione nativa, suggerendo magari un'implementazione e ottimizzazione differente della libreria OpenMP e del compilatore GCC su MacOS e Linux.

In futuro, potrebbe essere possibile un'ulteriore ottimizzazione del codice parallelo integrando i *vector datatype* **SIMD** per il calcolo delle proprietà delle particelle. In particolare mediante un *vector datatype* di quattro *float* è possibile accorpare quattro iterazioni di un ciclo for in una sola, trasformando i costrutti `if` nelle relative maschere di bit. Questa operazione però risulta tutt'altro che banale, ma sarebbe interessante vedere se effettivamente questa integrazione può portare a benefici in termini di performance.