Progetto di High Performance Computing 2022/2023

Marco Sangiorgi, matr. 0000971272

21 aprile 2023

1 Introduzione

In questa relazione sono valutate le scelte fatte nella realizzazione delle versioni parallele dell'algoritmo SPH [3] nella sua versione in linguaggio C [1], e le conseguenti misure di prestazione.

Il codice è stato parallelizzato sfruttando il parallelismo a memoria condivisa fornito da OpenMP e il parallelismo massivo delle GPU NVidia fornito da CUDA.

2 Hardware utilizzato

Sono state utilizzate 3 diverse macchine di test, con diverse architetture CPU (RISC, CISC), in modo da avere uno strumento di confronto dei codici sviluppati in diversi casi d'uso.

Su alcune macchine, sono stati utilizzati ambienti virtuali (container, ambienti di build) per compilare i codici ed eseguirne i programmi.

Le macchine utilizzate sono le seguenti:

	Server	**Win10**	**Mac**
Descrizione	Server istituzionale	Windows Laptop	MacBook
CPU	Intel Xeon 1.70GHz	Intel i7 (...)	Apple M1 3.20GHz
N. di Core	12 core	6 core + HyperT	4 core
RAM	64GB	16GB	16GB
S.O.	Ubuntu 9.4.0	Windows 10	MacOS 13.1
Compilatori	gcc 9.4.0 clang 10.0.0	MinGW gcc 11.3.0 MSVC clang 19.28.29913	gcc 9.4.0
NVCC	v11.5.119	v11.3.155	/
Virtualizzazione	/	MinGW 64bit	Docker Ubuntu 9.4.0

Tabella 1: Caratteristiche delle macchine utilizzate

3 Struttura dati utilizzata

La struttura dati del codice base è stata modificata da AoS (Array of Strucure) a SoA (Structure of Array) per i seguenti motivi:

1. L'analisi delle prestazioni favorisce strutture di tipo SoA, anche sul semplice codice seriale.
2. L'utilizzo di strutture di tipo SoA permette di sfruttare a pieno la programmazione CUDA (almeno per questo codice), riducendo gli accessi alla memoria globale della GPU.

4 Modellazione

Ponendo N il numero di particelle.

È stato utile modellare semanticamente l'insieme delle computazioni svolte come l'insieme di operazioni necessarie per visitare una matrice NxN, come descritto dall'annidamento di 2 cicli nelle funzioni principali dell'algoritmo (`compute_density_pressure` e `compute_forces`).
Pertanto, l'algoritmo è $\mathbf{O}(N^2)$.

5 Versione OMP

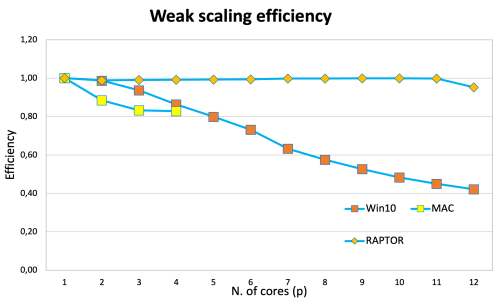
5.1 Versione 1 - OMPv1

OMPv1 sfrutta il pattern ***Embarassing Parallel*** e ***Reduction***.

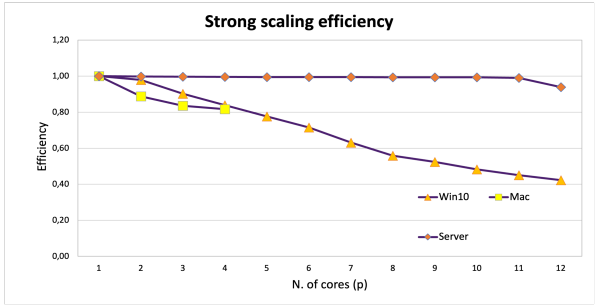
Nello specifico, ogni thread del pool esegue una riga della matrice NxN, nella quale vengono svolte operazioni di scrittura/lettura indipendenti dalle altre righe della matrice.

Analizzando la funzione più onerosa del codice (`compute_forces`), una suddivisione troppo grossolana del dominio (matrice NxN), porta - per via di un ramo condizionale corposo con numerose letture e calcoli - ad avere una distribuzione del carico di lavoro non bilanciata tra i vari thread. Pertanto, nel parallelizzare, è stato scelto di partizionare il dominio a grana finissima schedulandone l'assegnazione ai vari thread in fase di esecuzione (direttiva OpenMP `"schedule(dynamic,1)"`).
Dai grafici 1a, 1b e 1c si nota che:

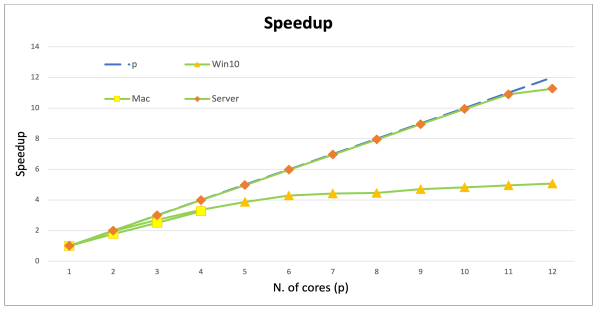
1. OMPv1 su Server si comporta come da aspettative (vedi 1a, 1b, 1c).
2. Gli ambienti di virtualizzazione utilizzati dalle macchine Win10 e MAC incidono negativamente sulle prestazioni di OMPv1 (vedi 1a, 1b, 1c).
3. OMPv1 su Win10 si comporta con una certa linearità fino all'utilizzo di tutti i 6 core fisici (vedi 1c).
4. OMPv1 su Server ha un lieve calo delle prestazioni utilizzando tutti i 12 core a disposizione (vedi 1a, 1b, 1c). Tuttavia, risulta una problematica non legata al programma OMPv1 in sé, ma da ricercare nell'architettura hardware e software del sistema Server.



(a) Grafico sulla Weak Scaling Efficiency di OMPv1.



(b) Grafico sulla Strong Scaling Efficiency di OMPv1.



(c) Grafico dello Speedup di OMPv1.

Figura 1: Grafici sulle prestazioni di OMPv1

5.2 Versione 2 - OMPv2

OMPv2 ***elimina i cicli ridondanti*** dell'algoritmo SPH base e sfrutta il pattern ***Reduction***.

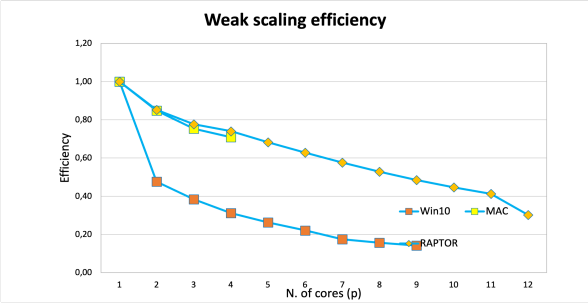
Nello specifico, sfrutta il fatto che ogni coppia di particelle viene visitata due volte nei due cicli annidati delle funzioni principali. In altre parole, si nota che la matrice NxN delle operazioni da svolgere è simmetrica rispetto alla diagonale principale, a meno di un cambio di segno.

Pertanto, il nuovo algoritmo (***SPH-new***) è stato ottimizzato per visitare solo il triangolo superiore della matrice NxN, riunendo in sé anche le operazioni del triangolo inferiore.
Analizzando *SPH-new*, si nota che:

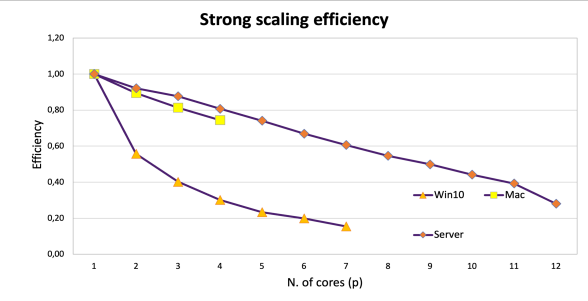
- *SPH-new* nella sua versione seriale è x2 volte più veloce di SPH base (vedi 2d).
- *SPH-new* è $O(\binom{N}{2})$[1].
- *SPH-new* ha introdotto delle Loop-carried dependencies tra le righe della matrice NxN.

Analizzando OMPv2 e i grafici 2a,2b e 2c, si nota che:

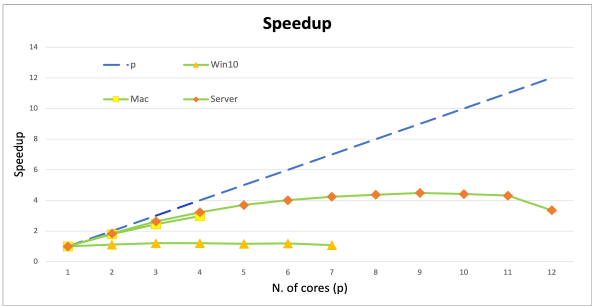
1. OMPv2 è soggetto a $O(N)$ overhead di creazione/distruzione dei pool di thread.
2. OMPv2 su Win10 ha un comportamento anomalo, ricollegabile ad un probabile fenomeno di False Sharing [2] riguardante l'array implicitamente creato da OpenMP per eseguire le riduzioni (vedi speedup costante in 2c).
3. OMPv2 non è efficiente come OMPv1 in termini di scalabilità (vedi 2a e 2b), anche per via di $O(N)$ costrutti di sincronizzazione tra i vari thread.
4. OMPv2 condivide i punti 2,3 e 4 di OMPv1.



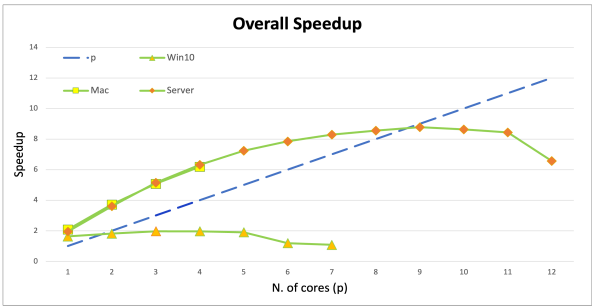
(a) Grafico sulla Weak Scaling Efficiency di OMPv2.



(b) Grafico sulla Strong Scaling Efficiency di OMPv2.



(c) Grafico dello Speedup di OMPv2.



(d) Grafico dello Speedup su SPH base di OMPv2

Figura 2: Grafici sulle prestazioni di OMPv2

5.3 Versione 3 - OMPv3

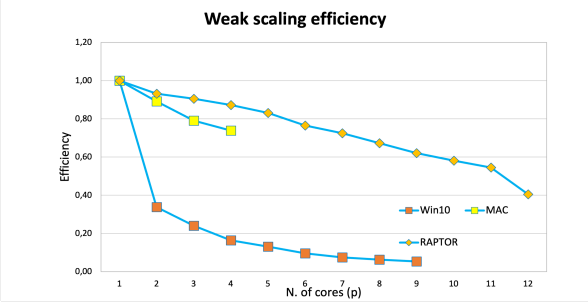
OMPv3 si basa su OMPv2 ma ***elimina overhead di creazione/distruzione dei pool di thread***.

Nello specifico, si fa carico di gestire il pattern *Reduction* manualmente riducendo a $O(1)$ il numero di creazioni/distruzioni dei pool di thread nell'algoritmo, a costo di una maggiore quantità di codice.
Analizzando OMPv3 e i grafici 3a,3b e 3c, si nota che:

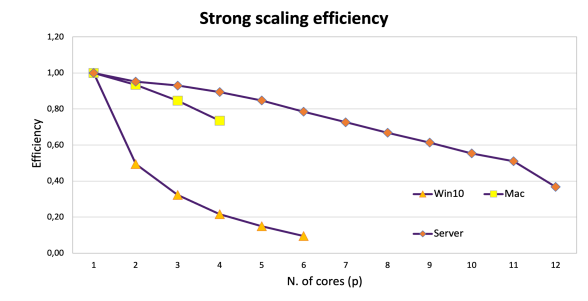
1. OMPv3 condivide tutti punti, tranne il punto 1, di OMPv2.

[1] Nel grafico 2a, *SPH-new* è stato comunque considerato come $O(N^2)$

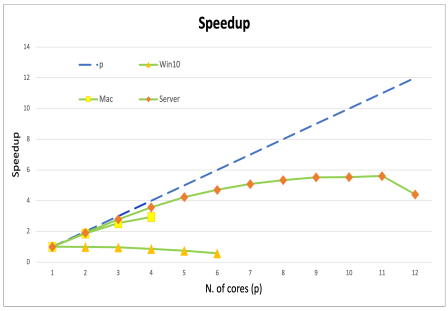
2. OMPv3 su Win10 è ancora meno efficiente di OMPv2 in termini di scalabilità (vedi 3a,3b), sebbene abbia prestazioni migliori in termini di speedup (vedi 3c).
   La motivazione è probabilmente da ricercarsi in una migliore gestione delle riduzioni da parte di OpenMP.



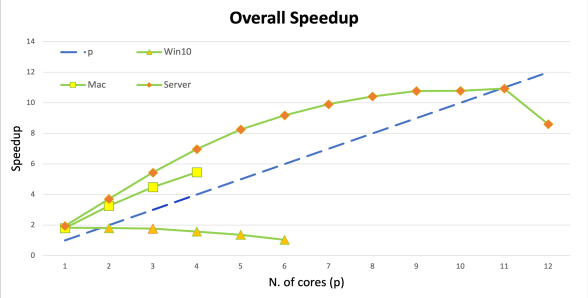
(a) Grafico sulla Weak Scaling Efficiency di OMPv3.



(b) Grafico sulla Strong Scaling Efficiency di OMPv3.



(c) Grafico dello Speedup di OMPv3.



(d) Grafico dello Speedup di OMPv3 rispetto a SPH base.

Figura 3: Grafici sulle prestazioni di OMPv3

5.4 Versione 4 - OMPv4

OMPv4 si basa su OMPv3 ma sfruttando a pieno le ***direttive atomiche di OpenMP***, eliminando il fenomeno del False Sharing [2] e le sincronizzazioni legate alle Loop-carried Dependencies introdotte da *SPH-new*.

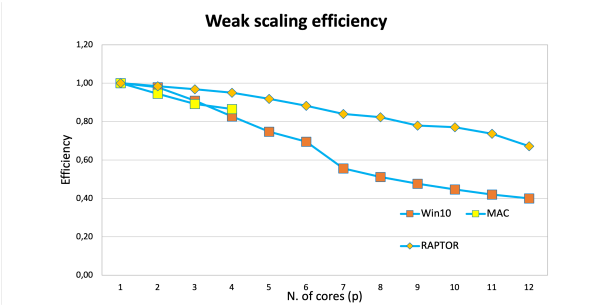
Nello specifico, è stato sfruttato la piena concorrenza tra i thread facendo affindamento ai meccanismi di mutua esclusione di OpenMP adottando somme atomiche direttamente nelle aree di memoria condivisa.

Tuttavia, è stato necessario aggiungere O(N) operazioni per inizializzare le zone di memoria condivisa prima della libera concorrenza tra i thread.
Analizzando OMPv4 e i grafici 4a,4b e 4c, si nota che:

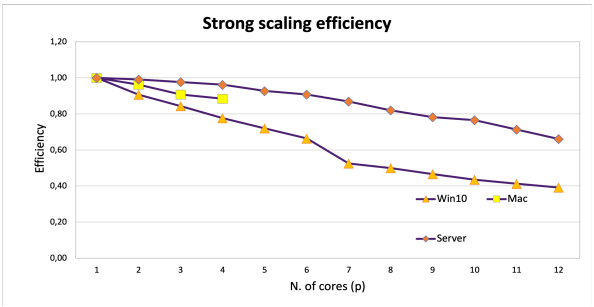
1. OMPv4 è molto più efficiente in termini di scalabilità delle precedenti due versioni (vedi 4a e 4b), sebbene non possa considerarsi tanto efficiente quanto OMPv1.

2. OMPv4 elimina il possibile fenomeno dovuto al False Sharing [2](vedi 4c), raggiungendo speedup maggiori delle precedenti due versioni.

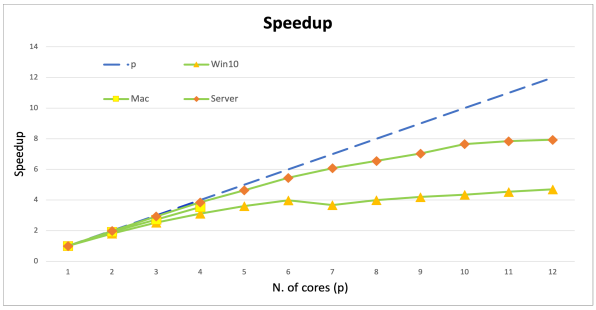
3. OMPv4 condivide i punti 2,3 e 4 di OMPv1.



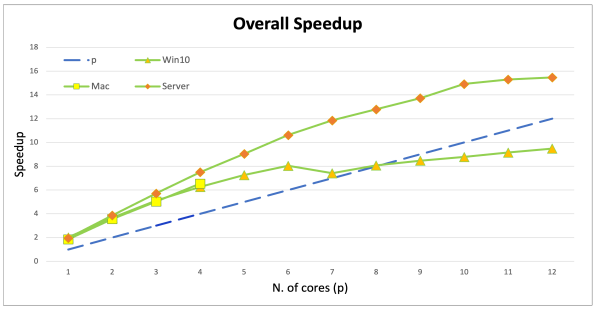
(a) Grafico sulla Weak Scaling Efficiency di OMPv4.



(b) Grafico sulla Strong Scaling Efficiency di OMPv4.



(c) Grafico dello Speedup di OMPv4.



(d) Grafico dello Speedup di OMPv4 rispetto a SPH base

Figura 4: Grafici sulle prestazioni di OMPv4

6 Versione CUDA

Le versioni CUDA di SPH sviluppate sfruttano il pattern ***Embarassing Parallel*** e ***Reduction***.
Nello specifico:

- È stata utilizzata una struttura dati di tipo SoA per garantire accessi contigui in memoria globale minimizzando il più possibile il numero di Cache Miss.
- È stato usato il meccanismo di ***Selection and Masking*** per evitare che lo scheduler di un warp serializzi le istruzioni a causa di rami condizionali, interrompendone la parallelizzazione.
- Il pattern *Reduction*, come visto a lezione, è stato implementato facendo uso di Shared Memory.
- Le costanti sono lette dalla constant memory.

6.1 Versione 1 - CUDAv1

CUDAv1 utilizza N threads per eseguire le operazioni delle N righe indipendenti della matrice NxN, come per OMPv1.

6.2 Versione 2 - CUDAv2

CUDAv2 sfrutta le ***direttive atomiche di CUDA***, ed utilizza $\binom{N}{2}$ threads, ciascuno per una ogni coppia di particelle. In altre parole, esegue le operazioni del triangolo superiore della matrice NxN riunendo in sé anche quelle del triangolo inferiore, come in *SPH-new*.

Questo versione richiede un elevato numero di operazioni atomiche che fungono da collo di bottiglia. Inoltre, richiede un numero di thread che cresce col quadrato della dimensione dell'input rendendolo non favorevole ad essere scalabile.

Tuttavia, è da considerarsi interessante per eventuali sviluppi futuri.

6.3 Grafici sulle prestazioni CUDA

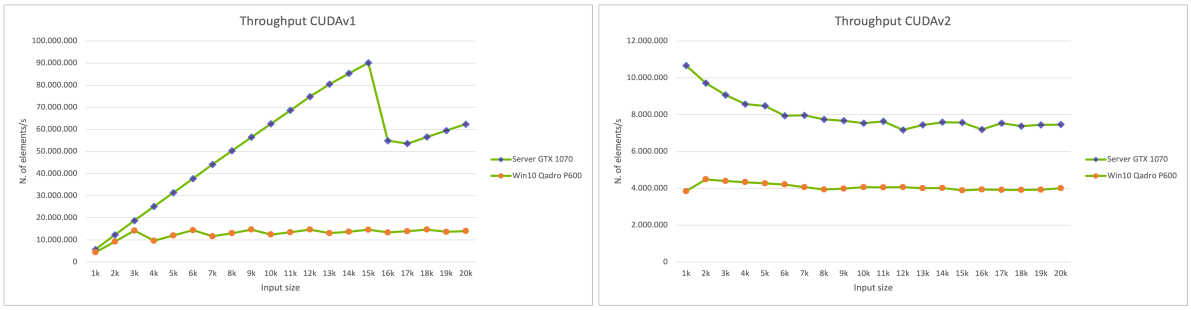
Dai grafici 5a,5b, si nota che:

- Si verifica una caduta di Throughput al raggiungimento della capacità massima (BLKDIM) di thread schedulabili concorrentemente di ogni Stream Multiprocessor(SM) presente sulla GPU. (vedi 5a - CUDAv1 - dove il numero di thread equivale alla grandezza dell'input).

  Pertanto, le cadute di throughput si hanno ogni: $SM * BLKDIM$ threads

  Per Server (GPU GeForce GTX 1070) questa caduta si verifica ogni 15360 threads (15 SM * 1024 $threads$).

  Per Win10 (GPU Quadro p600) ogni 3072 threads (3 SM * 1024 $threads$).



(a) Grafico dello Throghuput di CUDAv1

(b) Grafico dello Throghuput di CUDAv2

Figura 5: Grafici sulle prestazioni di delle versioni CUDA di SPH

7 Conclusioni

Si può concludere di aver affrontato il problema seguendo diverse strade e sfruttando il potenziale dei linguaggi e dell'hardware sottostante.

Soprattutto nelle versioni CUDA si nota la possibilità di migliorare il codice in eventuali sviluppi futuri sfruttando maggiormente la Shared Memory.

Riferimenti bibliografici

[1] A concise 2D implementation of Müller's interactive smoothed particle hydrodynamics (SPH) paper in C++. https://github.com/cerrno/mueller-sph.

[2] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *USENIX Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, Sept. 1993. USENIX Association.

[3] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 154–159, Goslar, DEU, 2003. Eurographics Association.