Progetto di High Performance Computing 2023/2024

Nicolò Monaldini, matr. 0001031164
25/01/2024

Introduzione

La presente relazione descriverà in che modo si è parallelizzato il programma preso in esame. La maggior parte del tempo di computazione è impiegato all'interno di tre cicli, che saranno oggetto di parallelizzazione. Due di questi, contenuti rispettivamente nei metodi reset_displacements() e move_circles(), rispettano il pattern embarassingly parallel ed hanno entrambi costo asintotico $O(n)$. L'ultimo è quello presente all'interno del metodo compute_forces(): esso rappresenta la parte più onerosa di computazione in quanto contiene un ciclo annidato ed ha costo $O(n^2)$. Il ciclo annidato rispetta il pattern embarassingly parallel, in quanto accede in lettura e scrittura agli stessi dati per ogni iterazione del ciclo esterno.

Versione OpenMP

Per quanto riguarda il metodo compute_forces(), data la dipendenza tra iterazioni successive del ciclo esterno, si potrebbe parallelizzare solamente il ciclo interno con un #pragma omp parallel for. Tuttavia, se ogni thread avesse una copia privata delle variabili dx e dy per ogni struct circle, si potrebbe parallelizzare il ciclo esterno, ed effettuare una riduzione sui valori di queste variabili alla fine della computazione presente nel metodo. Per questo motivo, al posto di mantenere le variabili dx e dy all'interno della struct circle si è preferito spostare tali variabili all'interno di due array, in modo da applicare a questi ultimi il pattern di riduzione attraverso la clausola reduction di OpenMP, potendo parallelizzare in questo modo il ciclo esterno. Un'ulteriore operazione di riduzione è stata utilizzata per trovare il numero totale delle intersezioni tra i cerchi.

Si noti come in questo modo le sia le iterazioni del ciclo interno sia quelle del ciclo innestato diventano indipendenti, rendendo possibile il collassamento dei cicli tramite la clausola collapse(2). Tuttavia, trattandosi di un ciclo con spazio di iterazioni non rettangolare (poiché il ciclo interno ha un numero di iterazioni che dipende dall'indice del ciclo esterno), tale clausola può essere utilizzata solamente a partire dalla versione 5.0 di OpenMP, e in ogni caso non sarebbe possibile utilizzare la clausola schedule [1]. Perciò si è scelto di parallelizzare il ciclo esterno con un partizionamento a grana fine, applicando il paradigma master-worker al fine di ottenere un bilanciamento del carico attraverso un'allocazione dinamica dei carichi di lavoro.

Nella versione seriale fornita, ad ogni iterazione del ciclo presente nel main() viene eseguito prima il metodo reset_displacements(), poi il metodo compute_forces() ed infine il metodo move_circles(). Dato che i valori delle variabili dx e dy sono inizializzati a zero, è possibile spostare l'esecuzione della funzione reset_displacements() dopo la funzione move_circles(). In questo modo è possibile aggiornare la posizione dei cerchi ed azzerare le variabili dx e dy in un unico ciclo Per questo motivo sono state sostituite le funzioni compute_forces() e reset_displacements(), che eseguono queste operazioni in due cicli in successione, con una funzione update_circles() che in unico ciclo svolge entrambe le operazioni. Questo porta ad un marginale miglioramento dovuto al fatto che la gestione di un ciclo causa un minimo di overhead.

A parità di input il risultato della versione parallelizzata con OpenMP è diverso da quello della versione seriale in quanto cambia l'ordine in cui vengono effettuate le somme, e ciò porta a risultati leggermente diversi a causa della non associatività della somma in virgola mobile.

Si analizza di seguito la scalabilità e l'efficienza della soluzione proposta. Tutte le valutazioni sono state effettuate sul server utilizzato per le lezioni di laboratorio. Per valutare la scalabilità è stato considerato lo speedup, calcolato come $S(p) = \frac{T_{parallel}(1)}{T_{parallel}(p)}$ $con\ p = 1, \dots, 12$. I tempi utilizzati nella

formula sono ricavati facendo la media dei tempi di più esecuzioni, per evitare la presenza di valori spuri.

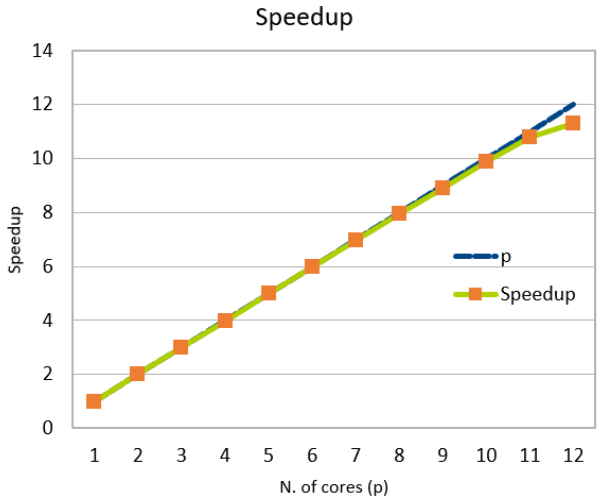

Figura 1: Speedup versione OpenMP

Dal grafico in figura 1 si può notare come, con l'aumentare del numero di processori, lo speedup effettivo si distanzi dallo speedup massimo teorico, ovvero $S(p) = p$. Avendo a disposizione un numero maggiore di processori ci si aspetta che lo speedup tenda asintoticamente a un valore che dipende dalla porzione di programma che viene parallelizzata, secondo la legge di Amdahl.

Per quanto riguarda l'efficienza sono state calcolate la strong scaling efficiency, calcolata come $E(p) = \frac{S(p)}{p}$ $con\ p = 1, \dots, 12$, e la weak scaling efficiency, calcolata come $W(p) = \frac{T_1}{T_p}$ $con\ p = 1, \dots, 12$, dove T_1 rappresenta il tempo impiegato da 1 processore per svolgere 1 unità di lavoro e T_p il tempo impiegato da p processori per svolgere p unità di lavoro. Per il calcolo dello speedup e della strong scaling efficiency il programma è stato eseguito con 20000 cerchi e 20 iterazioni.

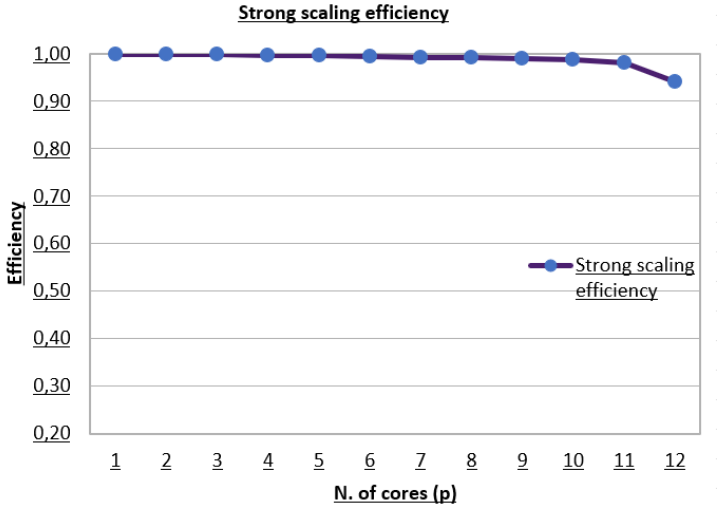

Figura 2: Strong scaling efficiency versione OpenMP

Dal grafico in figura 2 si può notare come, all'aumentare dei processori, la strong scaling efficiency diminuisca allontanandosi dal valore massimo, cioè 1. Dato che questa metrica rappresenta la frazione del tempo di esecuzione spesa in fasi computazionali, ci aspetta che all'aumentare dei processori la strong scaling efficiency tenda asintoticamente a 0, poiché col crescere del numero di processori diminuisce il rapporto tra il tempo impiegato per fasi computazionali e il tempo impiegato in fasi di sincronizzazione e comunicazione tra thread.

Per il calcolare la weak scaling efficiency è necessario che ogni processore svolga una quantità di lavoro costante, ovvero che $f(n_p, p) = const$, dove f rappresenta la funzione che calcola il lavoro

svolto da ogni processore. Per calcolare i vari T_p è necessario eseguire il programma con n_p diverso a seconda del numero di processori coinvolti nella computazione. Dato che la quantità di lavoro "seriale" svolto è $O(it * {n_p}^2)$, è necessario che $\frac{it*{n_p}^2}{p} = const \rightarrow n_p = \sqrt[2]{\frac{const*p}{it}} \rightarrow n_p = \sqrt[2]{p} * \frac{const\prime}{\sqrt[2]{it}}$. Si è scelto di mantenere il numero di iterazioni costante a 20, in questo modo si ricava n_p in funzione del numero di processori utilizzati per la computazione secondo la formula $n_p = \sqrt[2]{p} * \frac{const'}{\sqrt[2]{20}} = \sqrt[2]{p} * const''$. In questo caso si è posto $const'' = 10000$.

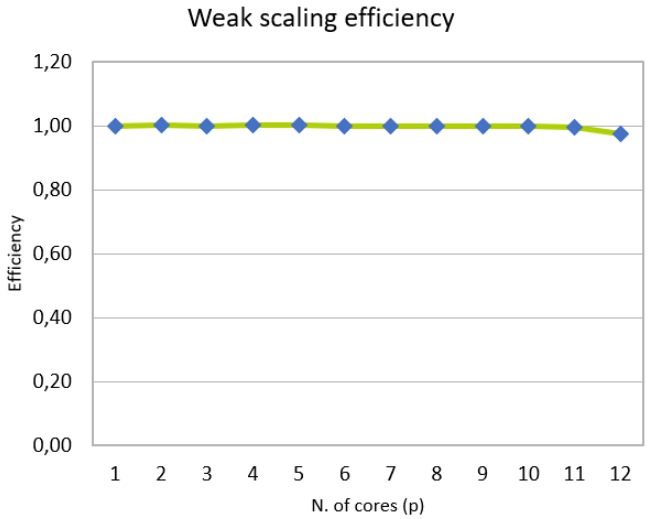

Figura 3: Weak scaling efficiency versione OpenMP

Dal grafico in figura 3 possibile notare un andamento simile rispetto al grafico in figura 2, in quanto anche in questo caso con l'aumentare del numero di processori si tende asintoticamente a 0, in quanto aumenta il tempo dedicato alle fasi di comunicazione tra i vari thread, a discapito delle fasi di computazione.

p	Average wct	Speedup	Strong scaling efficiency
1	24,27	1,00	1,00
2	12,14	2,00	1,00
3	8,10	3,00	1,00
4	6,08	3,99	1,00
5	4,87	4,98	1,00
6	4,07	5,97	1,00
7	3,49	6,95	0,99
8	3,06	7,93	0,99
9	2,73	8,91	0,99
10	2,46	9,87	0,99
11	2,25	10,80	0,98
12	2,15	11,29	0,94

Figura 4: Dati per il calcolo di speedup e strong scaling efficiency

p	Average wct	Weak scaling efficiency
1	24,28	1,00
2	24,26	1,00
3	24,27	1,00
4	24,25	1,00
5	24,25	1,00
6	24,26	1,00
7	24,27	1,00
8	24,28	1,00
9	24,29	1,00
10	24,32	1,00
11	24,40	0,99
12	24,90	0,97

Figura 5: Dati per il calcolo di weak scaling efficiency

Sono riportati in figura 4 e 5 i dati utilizzati per il calcolo delle metriche trattate e i corrispondenti valori ottenuti.

Versione CUDA

Per prima cosa, dato che nella versione seriale i dati relativi ai cerchi sono mantenuti all'interno di un array di strutture, si è deciso di cambiare tale struttura dati utilizzando una struttura di array, poiché in questo modo ci saranno più accessi in memoria di dati contigui da parte di thread dello stesso warp, che è un caso in cui gli accessi in memoria possono essere aggregati per aumentare il tasso di utilizzo dei bus.

Nella parallelizzazione del metodo compute_forces() non è possibile assegnare semplicemente ad ogni thread l'esecuzione di un'iterazione del ciclo esterno, per poi effettuare una riduzione sugli array che contengono le variabili dx e dy, dato che la memoria a disposizione di ogni thread non è sufficiente a mantenere array di grandi dimensioni. Per questo motivo si è scelto di assegnare ad ogni thread il calcolo delle variabili dx e dy relative ad un unico cerchio. Facendo in questo modo per ogni cerchio di indice *i* non è sufficiente calcolare le intersezioni con i cerchi di indice *j=i+1, i+2, ..., n-1*, ma è necessario calcolarle con i cerchi di indice *j=0, ..., n-1, j≠i*.

Nella funzione kernel relativa a questa computazione ogni thread deve poi avere cura di aggiornare solo i valori delle variabili dx e dy relative al cerchio di cui si occupa, e di contare come intersezioni solo quelle con elementi di indice superiore al proprio. Per evitare gli accessi in memoria globale, ogni thread mantiene i valori delle rispettive variabili dx e dy all'interno di variabili locali, e scrive il risultato in memoria globale alla fine della computazione.

Per il calcolo del numero totale di intersezioni è stato applicato il pattern riduzione, in particolare utilizzando una versione analoga a quella vista a lezione tramite l'uso delle funzioni atomiche CUDA, con qualche modifica. In primo luogo, il thread di indice globale 0 (ovvero il thread di indice 0 del blocco 0) si occupa di assegnare 0 alla variabile che conterrà il risultato della riduzione, in questo modo non è necessario che quest'ultima sia già inizializzata correttamente all'inizio della computazione. Inoltre, ad ogni addizione che viene effettuata durante la riduzione, si controlla che l'elemento che viene sommato sia stato effettivamente calcolato da un thread durante la computazione, in questo modo non è necessario che il numero di elementi sia multiplo della dimensione del blocco.

Per le funzioni reset_displacements() e move_circles() è stata applicata una strategia analoga alla versione OpenMP e sono state sostituite con un kernel update_circles(). Ogni thread si occupa di aggiornare la posizione di un cerchio e azzerare le rispettive variabili dx e dy.

Curiosamente, anche il risultato della versione CUDA è diverso rispetto alla versione seriale, nonostante l'ordine in cui si effettuano le somme sia lo stesso. Questo è dovuto al fatto che le GPU utilizzano delle tecniche diverse rispetto alle CPU per gli arrotondamenti delle operazioni in virgola mobile, come ad esempio la Fused Multiply-Add [2]. Nell'articolo presente nella bibliografia sono presenti dei flag di compilazione che, se configurati opportunamente, disattivano alcuni di questi meccanismi e in questo modo si ottiene un risultato più simile a quello del programma seriale.

Si analizzano di seguito le prestazioni del programma CUDA. Dato che utilizzando il programma non ha controllo sul numero di CUDA core utilizzati nella computazione, non è possibile usare le stesse metriche utilizzate per la versione per CPU. Si utilizza piuttosto il throughput e lo speedup rispetto alla versione parallelizzata con OpenMP, nel caso in cui usi il numero massimo di processori a disposizione. Dato che il costo del programma è $O(it * n^2)$, dove n rappresenta il numero di cerchi ed it il numero di iterazioni, il throughput (come milioni di operazioni al secondo) è calcolato con la formula $Throughput = \frac{it*n^2}{t*10^6}$. In questo caso si è scelto di porre $it = 20$ per ogni esecuzione. Per lo speedup rispetto alla versione parallelizzata per CPU si usa invece la formula $S = \frac{T_{CPU}}{T_{GPU}}$.

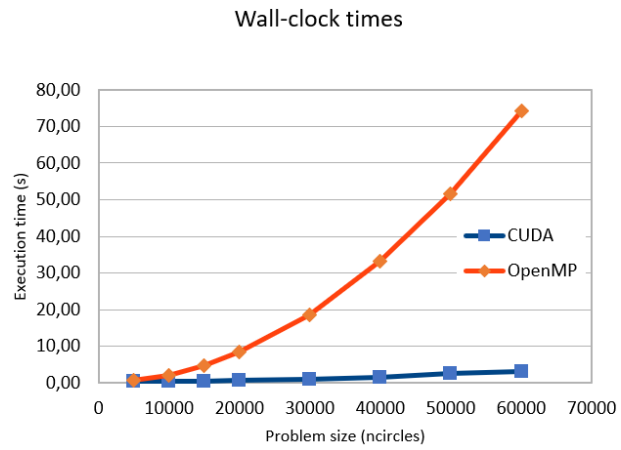

Figura 6: Grafico del wall-clock time

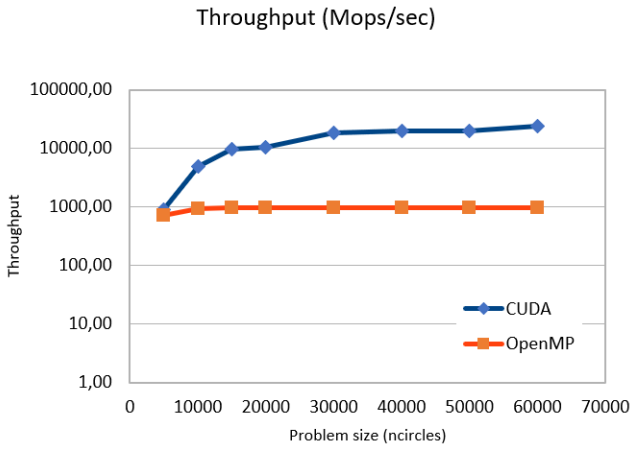

Figura 7: Grafico del throughput

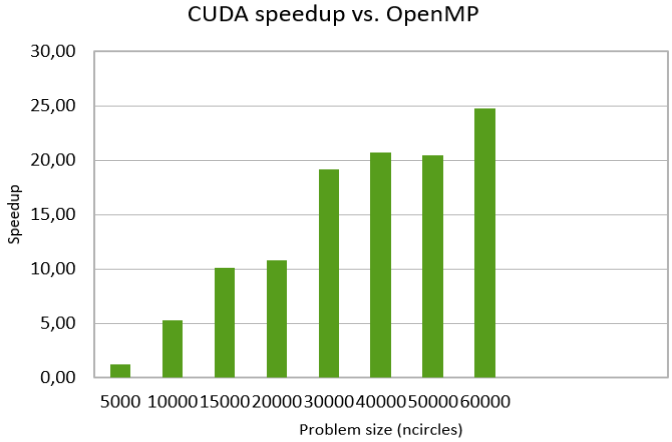

Figura 8: Speedup della versione CUDA rispetto alla versione OpenMP

Nel grafico in figura 6 si può notare come la curva relativa al programma CUDA aumenti di valore più lentamente, nonostante abbia anch'essa un andamento quadratico, poiché il parallelismo riduce il tempo di esecuzione di un fattore costante che dipende dal numero di unità di esecuzione che sono impiegate.

Il throughput di entrambe le versioni si stabilizza attorno a un valore, che è più alto nel caso della versione CUDA in quanto i tempi di esecuzione di quest'ultima sono più bassi rispetto alla versione OpenMP. Il vantaggio in termini di prestazioni della versione CUDA rispetto alla versione OpenMP è minore per input piccoli, probabilmente a causa della riduzione effettuata su GPU, che a causa delle varie operazioni di sincronizzazione tra thread risulta svantaggiosa per input piccoli rispetto ad una riduzione effettuata serialmente dalla CPU.

Problem size (ncircles)	Wall-clock time		Throughput (Mops/s)		CUDA speedup vs OpenMP
	CUDA	OpenMP	CUDA	OpenMP	
5000	0,56	0,70	894,72	711,28	1,26
10000	0,41	2,15	4933,20	929,00	5,31
15000	0,47	4,74	9647,21	949,44	10,16
20000	0,77	8,35	10395,28	958,34	10,85
30000	0,97	18,65	18516,92	964,90	19,19
40000	1,60	33,08	20006,75	967,28	20,68
50000	2,52	51,64	19829,33	968,20	20,48
60000	3,00	74,27	23980,40	969,49	24,74

Figura 9: Dati per il calcolo delle prestazioni della versione CUDA

Sono riportati in figura 9 i dati utilizzati per il calcolo di queste metriche e i corrispondenti valori ottenuti.

Conclusioni

In conclusione, è possibile notare il notevole miglioramento in termini di tempo di esecuzione che si ha con la versione CUDA. Utilizzando le più recenti versioni di OpenMP sarebbe possibile utilizzare il collassamento di cicli non rettangolari per provare ad ottenere una versione più ottimizzata del programma. Nella la versione CUDA si potrebbe gestire in maniera diversa la riduzione per input di piccole dimensioni.

Riferimenti bibliografici

[1] New features in OpenMP 5.0 and 5.1

[2] Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs