Di Riccio Tommaso 615257 31/03/2024

Compilazione ed esecuzione

Per compilare è possibile utilizzare il comando:

g++ -std=c++20 -O3 -march=native -I include/ UTWavefront.cpp -o UTW

Per l'esecuzione il comando è:

./UTW [n_threads N min max]

dove:

- *n_threads* è il numero di threads da utilizzare per l'esecuzione (default = 1)
- *N* è la dimensione della matrice quadrata NxN (default = 512)
- *min* è il tempo minimo di attesa (us) inserito all'interno di una cella della matrice (default = 0)
- *max* è il tempo massimo di attesa (us) inserito all'interno di una cella della matrice (default = 1000)

Problemi affrontati e soluzioni

Il primo problema affrontato nello sviluppo della versione parallela di questo codice è l'accesso in concorrenza agli elementi delle diagonali della matrice. Difatti, nonostante sia necessario solo leggere i valori presenti, vogliamo evitare che due thread accedano allo stesso elemento per non effettuare lavoro ridondante riducendo di conseguenza l'efficienza.

Il secondo problema è la sincronizzazione dei thread che devono attendere che tutti gli elementi di una diagonale siano calcolati prima di poter passare agli elementi della diagonale successiva.

Per risolvere il primo problema viene utilizzata una variabile atomica condivisa come indice dell'elemento della diagonale da calcolare. Ogni thread utilizza il comando *fetch_add(1, std::memory_order_seq_cst)* per leggere il valore dell'indice e in modo atomico incrementarlo di uno. Successivamente, viene effettuato un controllo per verificare che l'indice sia minore o uguale al numero di elementi della diagonale corrente. Si noti che questo controllo non è soggetto a problemi di concorrenza, poiché una volta letto il valore dell'indice, il valore di verità della condizione è indipendente dalle azioni degli altri thread.

Il secondo problema è stato risolto utilizzando una barriera (classe *std::barrier* in C++20). Quando un thread legge un indice dalla variabile condivisa maggiore del numero di elementi sulla diagonale, esce dal ciclo *while* ed esegue la chiamata *arrive_and_wait( )* sulla barriera. Una volta che tutti i thread hanno eseguito la chiamata, cioè che tutti gli elementi della diagonale sono stati calcolati, un singolo thread esegue la funzione *on_completion( )* passata come argomento al momento della creazione della barriera. Questa funzione si occupa di incrementare la variabile *diag_k* che indica la diagonale considerata dai thread nelle iterazioni. Dato che la funzione è eseguita prima che i thread siano risvegliati, dunque in assenza di concorrenza, la variabile *diag_k* può essere dichiarata di tipo *int*. Dopo il completamento della funzione i threads possono tornare ad eseguire il ciclo *while* spartendosi gli elementi della nuova diagonale. Questo viene ripetuto per tutte le diagonali della matrice.

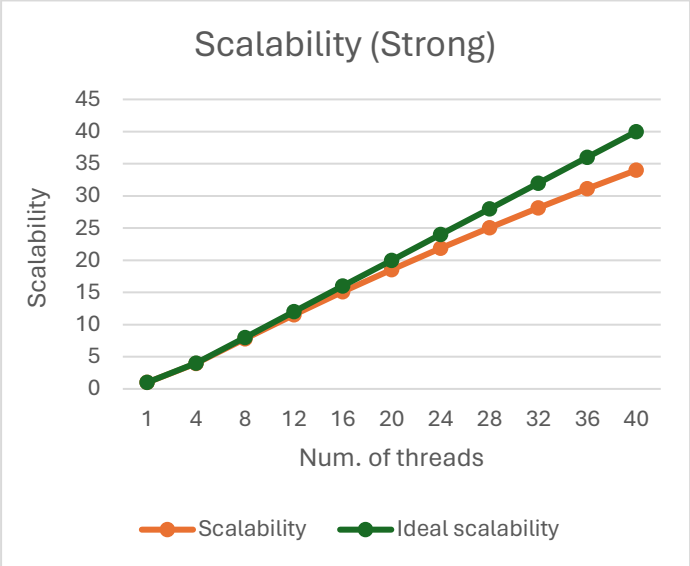
Queste soluzioni offrono il vantaggio di eliminare la necessità di assegnare staticamente gli elementi della diagonale ai singoli thread, evitando così eventuali squilibri nella distribuzione del lavoro. Ciò rende il codice robusto rispetto alla variabilità dei tempi di attesa nelle celle della matrice. Inoltre, evitando l'uso dei mutex, il codice diventa più efficiente quando si utilizza un alto numero di thread.

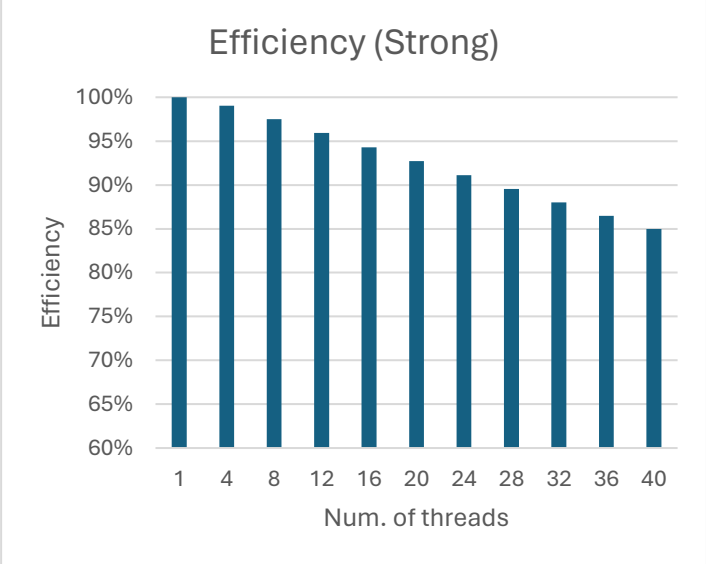
Esperimenti e risultati

Nella valutazione delle performance del codice sono stati effettuati esperimenti sulla weak e strong scalability.

Per analizzare la **strong scalability** la dimensione del problema è stata fissata a 512x512 e il numero di thread è stato fatto variare da 1 a 40.

Threads	1	4	8	12	16	20	24	28	32	36	40
Ideal Time (ms)	65540,0	16385,0	8192,5	5461,7	4096,3	3277,0	2730,8	2340,7	2048,1	1820,6	1638,5
Real Time (ms)	65552,0	16546,0	8402,7	5693,9	4345,1	3534,9	2997,1	2614,5	2327,7	2105,3	1928,1





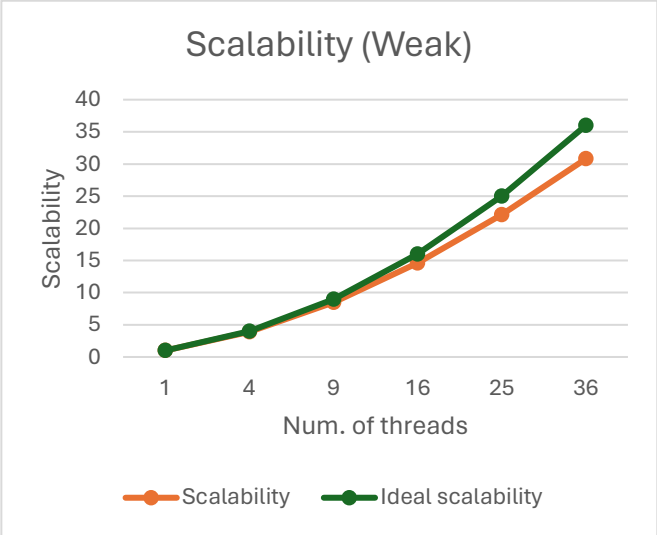
Il tempo ideale è calcolato dividendo la somma dei tempi di attesa di tutta la matrice per il numero di threads. Questa statistica è stata utilizzata solo come riferimento approssimativo ed è possibile utilizzare metodi migliori per calcolare un valore più vicino a quello effettivo prendendo in considerazione i vincoli del problema.

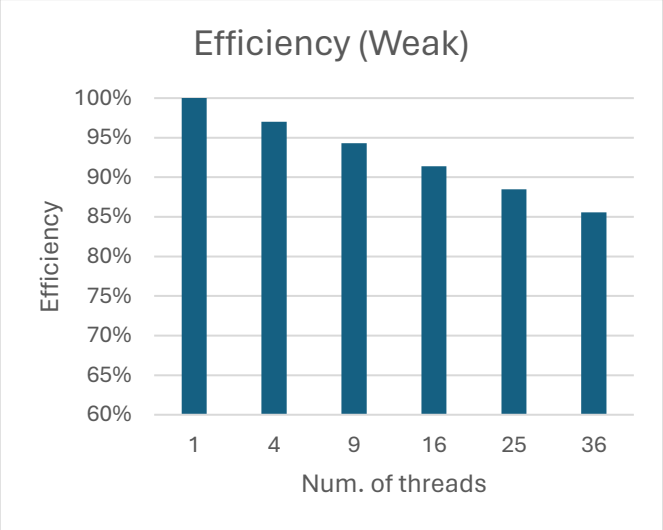
Dai grafici si può notare come l'efficienza decresca in modo lineare all'aumentare del numero di threads fino a raggiungere l'85% con l'utilizzo di 40 threads.

La causa principale di una scalabilità e di una efficienza più bassa di quelle ideali è da ricercare nella struttura del problema affrontato. Man mano che procediamo lungo le diagonali della matrice, il numero di elementi presenti all'interno di esse diminuisce. Poiché non possiamo calcolare gli elementi della diagonale successiva fino a quando non abbiamo completato tutti quelli della diagonale corrente, e il lavoro per il calcolo di un elemento non può essere diviso tra più threads, il massimo parallelismo si ottiene quando gli M elementi della diagonale sono calcolati contemporaneamente da M threads. Tuttavia, se il numero di elementi sulla diagonale corrente è inferiore al numero di thread utilizzati, alcuni di essi rimarranno inattivi, in attesa sulla barriera fin dalla prima iterazione. Ad esempio, utilizzando 40 thread partiremo ad avere un thread inattivo a 39 diagonali dal termine, fino ad avere 39 thread inattivi per l'ultima diagonale, rendendo impossibile una scalabilità pari a 40.

Passando ad analizzare la **weak scalability**, dato che la dimensione del problema è pari a NxN per mantenere il carico di lavoro di ogni thread costante si è scelto di utilizzare N = 80 * i con i ∈ [1,2,3,4,5,6] e un numero di thread pari a i^2 (i.e. # threads ∈ [1, 4, 9, 16, 25, 36]). Per ogni dimensione del problema considerata si è calcolato anche il tempo di esecuzione utilizzando un singolo thread.

Threads	1	4	9	16	25	36
Problem size N	80	160	240	320	400	480
Sequential Time (ms)	1631,2	6467,1	14510,9	25709	40074,7	57609,5
Ideal Parallel Time (ms)	1629,9	1616,4	1612,0	1606,5	1602,7	1599,9
Parallel Time (ms)	1630,6	1666,7	1710,0	1758,1	1811,7	1870,4





I risultati sono molto simili a quelli ottenuti nell'analisi della strong scalability e i commenti riportati per essa sono validi anche per questi esperimenti. Infatti, se aumentando sia la dimensione del problema che il numero di thread utilizzati rende la prima parte del lavoro realizzabile in ugual tempo, il maggiore numero di thread porta ad incrementare il numero di diagonali per le quali non è possibile sfruttare a pieno le risorse a disposizione.

Infine, per verificare che grazie all'assegnazione dinamica degli elementi ai thread i parametri *min* e *max* abbiano una bassa influenza sulle performance del programma, sono stati effettuati altri tre esperimenti. Il numero di thread è stato fissato a 20, il parametro min a 0, mentre a max sono stati assegnati i valori 1000, 5000 e 1000.

Threads	20	20	20
Min - Max	0 - 1000	0 - 5000	0 - 10000
Seq Time	65540	327700,7	655401,8
Parallel Time	3535,2	17641,8	35274,9
Efficiency	93%	93%	93%

Come si può vedere l’efficienza rimane costante al 93%, confermando quanto riportato.

Commenti aggiuntivi

I valori riportati sono il risultato di una media effettuata su 5 iterazioni. È stato scelto di non riportare statistiche come varianza, massimo, minimo, ecc. in quanto le variazioni sono trascurabili.