Di Riccio Tommaso 615257 23/05/2024

Compilazione ed esecuzione

Per compilare è possibile utilizzare il comando:

g++ -std=c++17 -I[FF-path] -O3 -o Word-Count-FF-par Word-Count-FF-par.cpp (Versione 1)

g++ -std=c++17 -I[FF-path] -O3 -o Word-Count-FF-par Word-Count-FF-par2.cpp (Versione 2)

Per l'esecuzione il comando è:

./Word-Count-FF-par /opt/SPMcode/A2/filelist.txt [extraworkXline] [topk] [showresults] [nthreads] [chunk-size]

dove

- *extraworkXline* è il lavoro extra svolto per ogni riga (default = 0)
- *topK* è il numero di parole da mostrare in ordine di frequenza (default = 10)
- *showresults* indica se i risultati devono essere mostrati o no (default = 0)
- *nthreads* è il numero di thread da utilizzare per l'esecuzione (default = 1)
- *chunk_size* è il numero di linee elaborate da ogni worker (default = 10000)

Problemi affrontati e soluzioni

La prima versione implementata consiste nell'utilizzare una ff_Farm dove:

- l'Emitter legge i file, inserendo le linee in un vettore *chunk* fino a raggiungere la dimensione *chunk_size*. Al raggiungimento di questa condizione il *chunk* viene inviato a un worker della farm. Per inserire *chunk_size* linee nel *chunk* possono essere utilizzati più file ed esso viene inviato con una dimensione minore solo quando sono terminati i file da elaborare.
- il Worker crea una unordered_map che utilizza per contare le occorrenze delle parole all'intero del *chunk* ricevuto dall'Emitter. Questa umap viene poi inviata al Collector.
- Il Collector riceve le umap dagli workers e aggiorna la propria unordered_map. Quest'ultima al termine dell'esecuzione conterrà il risultato finale.

In questa versione il numero di thread minimo è tre (un Emitter, un Worker, un Collector).

Nella seconda versione implementata la ff_Farm utilizza un singolo nodo come Emitter e Collector. La logica del programma rimane equivalente alla prima versione ma con un singolo nodo che si occupa di entrambe le funzioni assegnate ad Emittor e Collector. In questo modo si cerca di comprendere se a parità di numero di thread, utilizzare più risorse il processamento di un *chunk* porti a un vantaggio prestazionale. In questa versione il numero di thread minimo è due (un Emitter-Collector e un Worker).

Per entrambe le versioni, come nel precedente assignment, la variabile *total_words* è stata trasformata in *atomic_int*. Questo ha permesso agli workers di eseguire in parallelo il metodo *tokenize_line(...)*, su diverse linee di testo, mantenendo la correttezza del conteggio delle parole totali.

Esperimenti e risultati *(test eseguiti su spmcluster.unipi.it)*

Sia per la versione sequenziale che per quella parallela sono stati considerati i tempi di esecuzione escludendo il sorting, dato che quest'ultimo era equivalente per entrambe.

Nella seguente tabella sono riportati i tempi per la **versione sequenziale** utilizzando 0, 1000 e 10000 come lavoro extra

Sequenziale			
Extra	0	1000	10000
Tempo (sec)	5,6	19,91	148,62

Per le versioni parallele sono riportati i tempi di esecuzione utilizzando 3, 5, 10 e 20 threads, con i relativi *extraworkXline* e *chunk*_size utilizzati. La *chunk*_size è stata scelta sperimentando diversi valori per le varie combinazioni.

Questi sono i risultati della **prima versione** (ff_Farm con Emitter, Workers e Collector)

3 Threads (1 Worker)			
Extra	0	1000	10000
Chunk size	100000	100000	100000
Tempo (sec)	5,89	20,11	147,48

5 Threads (3 Workers)			
Extra	0	1000	10000
Chunk size	25000	25000	25000
Tempo (sec)	2,29	7,54	51,24

10 Threads (8 Workers)			
Extra	0	1000	10000
Chunk size	50000	25000	10000
Tempo (sec)	1,99	4,51	20,64

20 Threads (18 Workers)			
Extra	0	1000	10000
Chunk size	500000	15000	10000
Tempo (sec)	2,88	7,72	15,17

Speedup		Extra		
		0	1000	10000
Threads	3	0,95	0,99	1,01
	5	2,45	2,64	2,90
	10	2,81	4,41	7,20
	20	1,94	2,58	9,80

Mentre i risultati della **seconda versione** (ff_Farm con Emitter-Collector e Workers) sono

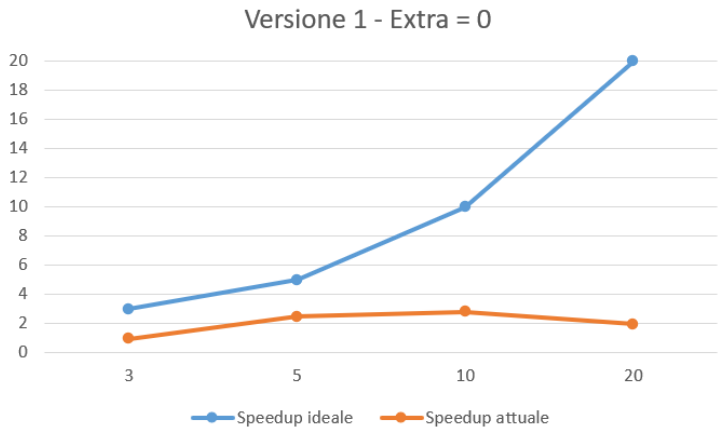
3 Threads (2 Worker)			
Extra	0	1000	10000
Chunk size	100000	100000	100000
Tempo (sec)	4,36	11,91	76,58

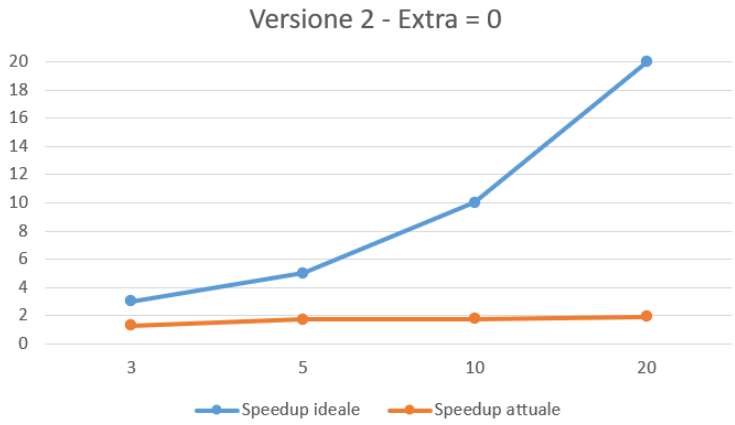
5 Threads (4 Workers)			
Extra	0	1000	10000
Chunk size	25000	25000	25000
Tempo (sec)	3,23	7,43	39,61

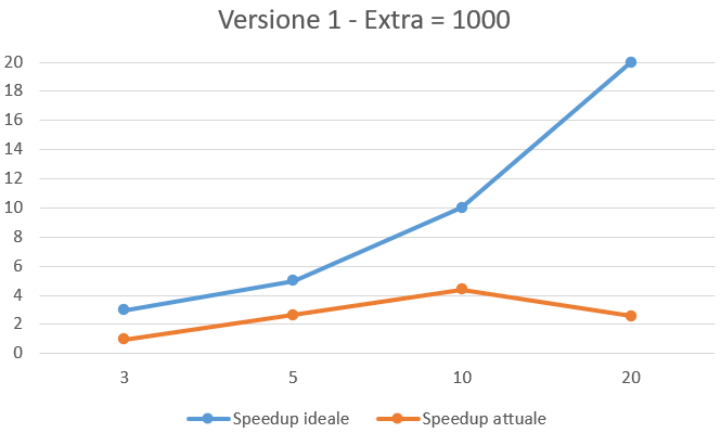
10 Threads (9 Workers)			
Extra	0	1000	10000
Chunk size	50000	25000	10000
Tempo (sec)	3,17	5,73	20,36

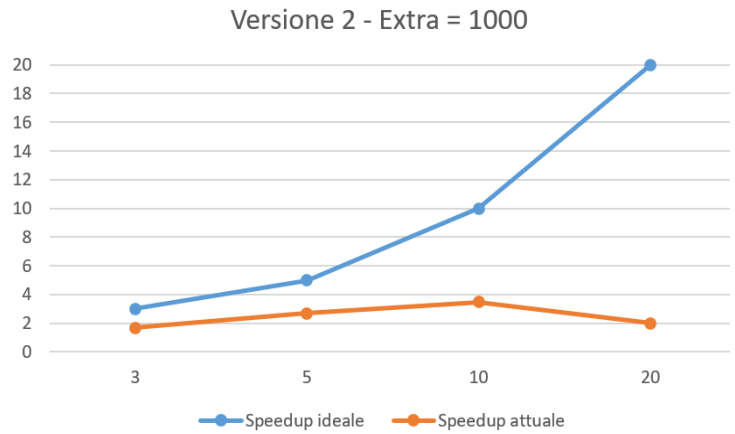
20 Threads (19 Workers)			
Extra	0	1000	10000
Chunk size	500000	15000	10000
Tempo (sec)	2,92	9,93	16,9

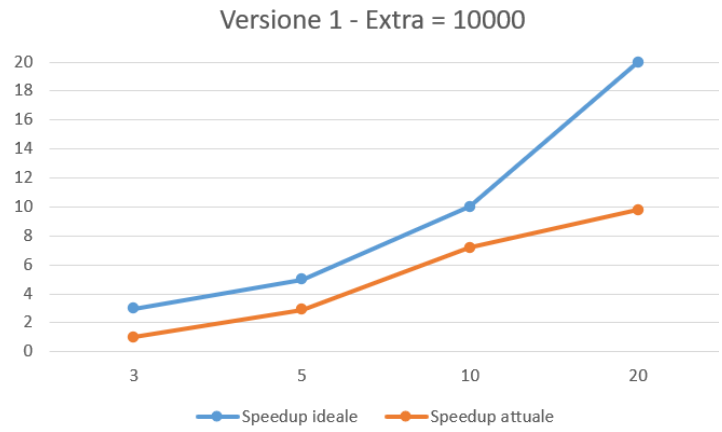
Speedup		Extra		
		0	1000	10000
Threads	3	1,28	1,67	1,94
	5	1,73	2,68	3,75
	10	1,77	3,47	7,30
	20	1,92	2,01	8,79

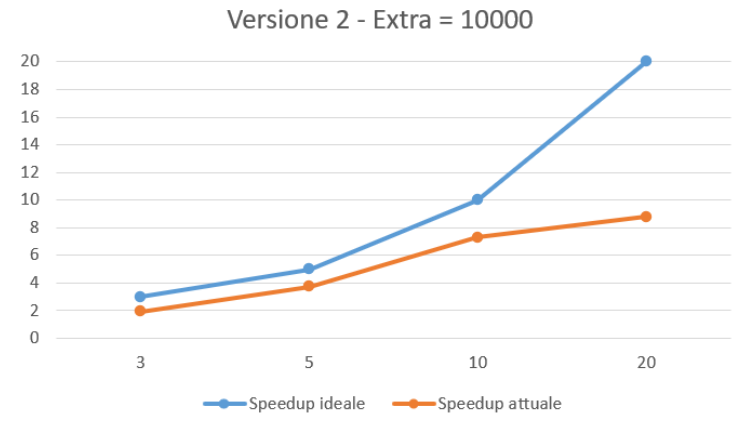












La seconda versione risulta generalemente più veloce con 3 e 5 threads, a riprova che quando le risorse sono scarse è conveniente spostarne l'utilizzo nel processamento dei chunk di testo. Con 10 e 20 threads invece la prima versione risulta più prestazionale mostrando come, quando si hanno abbastanza risorse, utilizzare un thread dedicato a leggere i file e creare i chunk ed un thread dedicato a fare la reduce delle unordered_map locali ricevute, porti ad un vantaggio prestazionale.

Una eccezione a queste osservazioni si ha con 5 threads e *extraworkXline* pari a 0, dove la prima versione risulta migliore della seconda (speedup 2,45 vs 1,73). Questo conferma quanto scritto nel precedente assignment: quando *extraworkXline* ha valore pari a 0 avere un singolo thread incaricato sia creare i *chunk* da inviare agli worker sia di aggiornare la unordered_map con i risultati parziali, crea un collo di bottiglia già con 3 workers nella farm.