PC-2022/23 Levenshtein distance

Francesco Bellezza
E-mail address
francesco.bellezza@stud.unifi.it

Abstract

All'interno di questo progetto abbiamo implementato un algoritmo in c++ che calcola la levenshtein distance tra due stringhe. E' stata anche effettuata una versione parallela di tale algoritmo, sfruttando openmp. I test sono stati effettuati su un pc contenente un Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. Dalle varie prove che sono state effettuate, abbiamo notato uno speedup di circa 3 del programma parallelo rispetto al programma sequenziale. Inoltre, questo vale solo per stringhe abbastanza lunghe, infatti, per le stringhe di piccole dimensioni, non c'è una differenza sostanziale di tempi di esecuzione tra l'algoritmo sequenziale e quello parallelo.

1. L'algoritmo

L'algoritmo consiste nel creare inizialmente una matrice di dimensione m*n, dove m e n sono le dimensioni della prima e della seconda stringa da confrontare aumentate di uno. Si definisce la prima riga e la prima colonna di elementi, dove al suo interno sono salvati semplicemente l'indice di riga e di colonna corrispondenti. Successivamente, con un doppio ciclo for annidato, si scorre la matrice per colonne, incrementando volta volta l'indice di riga. (L'indice di riga e di colonna sono rappresentati dalla lettera i e j rispettivamente). Si confronta i caratteri relativi agli indici di e j diminuiti di 1 e, in base all'esito del risultato, si aggiorna la matrice. Infine, il risultato desiderato, si trova nell'ultima casella in basso a destra della matrice.

1.1. Il codice sequenziale

Il codice sequenziale consiste nel ricreare l'algoritmo così come è stato posto. Abbiamo deciso di non utilizzare una vera e propria matrice, ma di sfruttare un vettore linearizzato. Questo perché ci consente di risparmiare spazio, e l'algoritmo di per sé risulterà più efficiente. L'unica differenza è che se vogliamo accedere all'elemento di riga i e colonna j dovremmo scrivere: mat[i*n+j], dove n è il numero di colonne della matrice, i rappresenta l'indice di riga e j l'indice di colonna. Dopo avere inizializzato le variabili nel modo in cui è descritto all'interno dell'algoritmo, si procede con i confronti ad ogni iterazione del ciclo for più interno (che itera sulle righe): se le due stringhe in posizione i-1 e j-1 rispettivamente sono uguali, allora significa che il costo della operazione è nullo, altrimenti il costo è uguale ad uno. Infine si assegna in posizione i e j della matrice il minimo tra l'elemento che sta alla sinistra di (i,j) aumentato di 1, l'elemento che sta sopra (i,j) aumentato di 1, e l'elemento che sta nella diagonale sinistra di (i,j) aumentato del costo dell'operazione precedentemente calcolato.

1.2. Il codice parallelo

Il codice parallelo invece richiede di iterare sulla matrice in maniera differente. Dobbiamo visitarla in modo tale da poter parallelizzare le attività. Il metodo adottato consiste nel parallelizzare il calcolo degli elementi della matrice che si trovano sulle anti-diagonali. Questo è possibile purché vengono calcolati gli elementi sulle antidiagonali in ordine da sinistra verso destra. Tale parallelizzazione può risultare efficiente solo per matrici di dimensioni abbastanza grandi. Abbiamo sfruttato la parallelizzazione anche per inizializzare la prima riga e la prima colonna della matrice, grazie all'istruzione #pragma omp parallel for. Per calcolare le varie celle che si trovano sull'antidiagonale abbiamo suddiviso la matrice in due sezioni: la prima sezione riguarda la metà sinistra della matrice, la seconda invece riguarda la metà destra di quest'ultima. In entrambi i casi, per quanto detto precedentemente sul fatto che possiamo parallelizzare solo gli elementi che si trovano sulla stessa antidiagonale, abbiamo parallelizzato solo il ciclo for più interno. Per rendere ogni iterazione del ciclo for indipendente dalle altre,

abbiamo utilizzato le variabili j_start e i_start: inizialmente assumono per tutti i thread lo stesso valore, successivamente però verranno impostate al valore che dipende dal valore di count, variabile che viene utilizzata come contatore del ciclo. Nel ciclo relativo alla metà sinistra della matrice la posizione di partenza (x,y) è uguale a (0,i+1), mentre nella seconda metà della matrice la posizione di partenza è (j-1,m). Per il resto, la logica dell'algoritmo è la stessa, cambia solo il modo in cui accediamo alla matrice.

2. Analisi della correttezza

Per verificare o meno la correttezza di tale codice abbiamo inizialmente confrontato i risultati ottenuti con quelli ottenuti dalla pagina wikipedia fornita dal professore. Inoltre, per testi di grandi dimensioni, abbiamo verificato che l'algoritmo sequenziale e parallelo avessero i medesimi risultati.

3. Analisi delle prestazioni

In questo paragrafo verranno specificate le specifiche hardware utilizzate per testare il progetto. Inoltre, verranno forniti alcuni test per verificare che l'approccio parallelo sia effettivamente più veloce di quello sequenziale.

3.1. Hardware

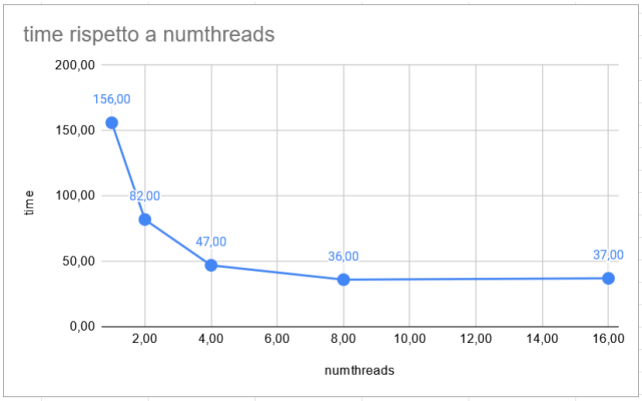
Abbiamo testato questo programma C++ su un computer con CPU Intel Core i5-8250U 1.60 GHz, 1801 Mhz, 4 core e 8 processori logici. Abbiamo preso diverse sezioni dell'Amleto di Shakespeare e due file di testo generati casualmente.

3.2. Settings dei test

Per creare i vari test, abbiamo scelto di utilizzare una variabile numthreads che definiamo all'inizio del file distance.cpp, in questo modo, possiamo scegliere prima di ogni esecuzione quanti thread utilizzare per il processo. I file sono di dimensioni nell'ordine di 30-60 Kbyte.

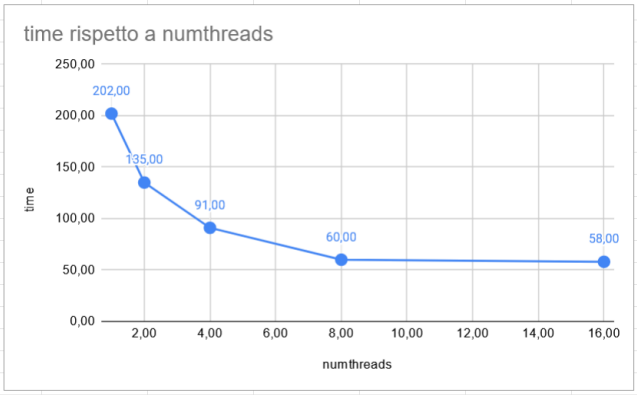
3.3. Test confronto tra i file hamlet_1.txt e hamlet_1.txt

Da questo test dovremmo ottenere il risultato 0, in quanto stiamo confrontando lo stesso testo.



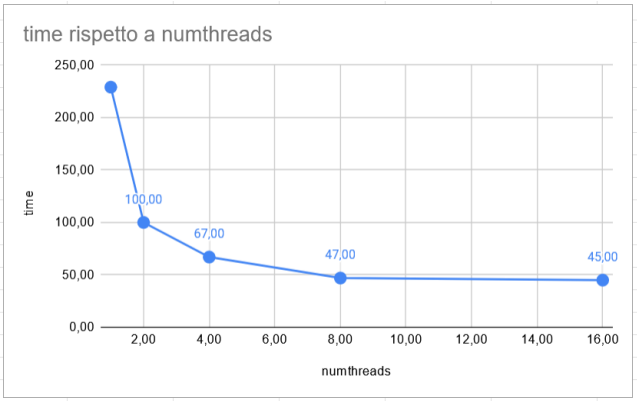
In questo caso abbiamo misurato uno speedup di 3,09.

3.4. Test confronto tra i file hamlet_1.txt e hamlet_2.txt



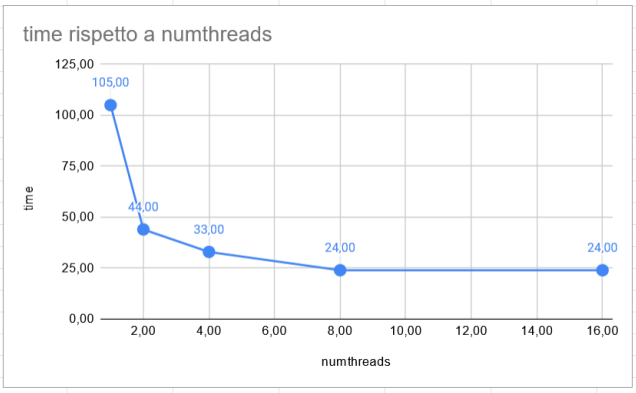
Lo speedup in questo esempio è intorno a 2,35.

3.5. Test confronto tra i file hamlet_2.txt e hamlet_3.txt



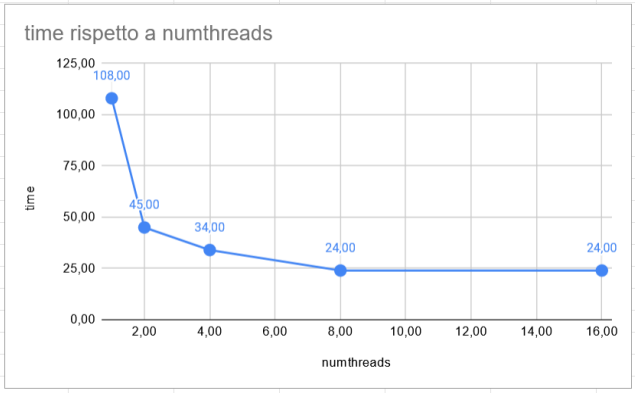
In questo caso invece abbiamo uno speedup di 3,54.

3.6. Test confronto tra i file hamlet_3.txt e hamlet_4.txt



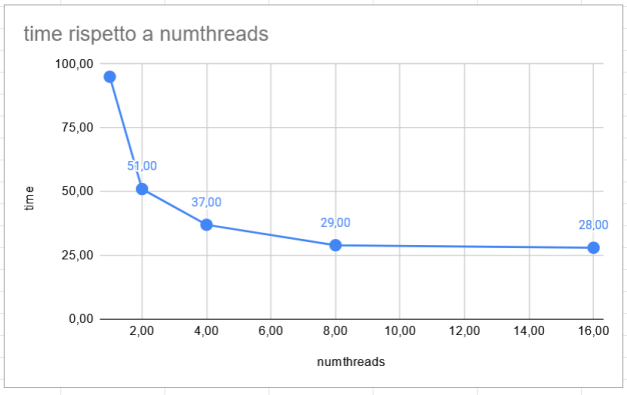
Lo speedup anche in questo caso si aggira intorno a 3, per la precisione ha un valore di 3,36.

3.7. Test confronto tra i file hamlet_3.txt e text1.txt



Questa volta lo speedup medio è di 3,40.

3.8. Test confronto tra i file text1.txt e text2.txt



Infine, l'ultimo speedup risulta quello meno sostanzioso, ed è di 2,62.

3.9. Esempi di piccole dimensioni

Per quanto riguarda gli esempi di piccole dimensioni, sono state effettuate delle prove con stringhe di 20-30 caratteri e, con testi di queste grandezze, non è riscontrabile un miglioramento e o peggioramento delle performance dell'algoritmo parallelo rispetto a quello sequenziale.

4. Considerazioni finali

Il programma parallelo performa considerevolmente meglio rispetto al programma sequenziale. Questo avviene solo per file di dimensioni sufficientemente grande affinché si possa sfruttare a pieno la parallelizzazione dei processi. Lo speedup medio dei tre esempi è di circa 3.