Parallel Edit Distance
Parallel Programming for Machine Learning

Niccolò Arati
niccolo.arati@edu.unifi.it

Abstract

*Questo progetto vuole analizzare i vantaggi di un'implementazione parallela dell'algoritmo di Edit Distance. In particolare si vuole andare ad osservare lo **speedup**, calcolato come rapporto tra il tempo di esecuzione sequenziale e tempo di esecuzione parallelo, al variare di diverse configurazioni dei parametri principali dell'algoritmo, oltre che al variare di diverse implementazioni dell'algoritmo sequenziale. Inoltre, verranno analizzati i risultati anche cambiando il numero di threads relativi alla configurazione parallelizzata.*

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduzione

Edit Distance, chiamata anche Distanza di Levenshtein, è una misura di similarità tra due stringhe A e B. In particolare indica il numero minimo di operazioni base da effettuare per trasformare la stringa A nella stringa B. Le operazioni base considerate in questo lavoro sono:

- **Inserimento** di un carattere;
- **Sostituzione** di un carattere;
- **Rimozione** di un carattere.

L'algoritmo può essere sintetizzato con la seguente formula ricorsiva:

$$lev_{A,B}(i,j) = \begin{cases} max(i,j), \text{if } min(i,j) = 0 \\ min \begin{cases} lev_{A,B}(i-1,j) + 1 \\ lev_{A,B}(i,j-1) + 1 \\ lev_{A,B}(i-1,j-1) + id \end{cases} \end{cases} \quad (1)$$

Nella Eq. (1), $id = \mathbf{1}(A_i \neq B_i)$, dove $\mathbf{1}$ corrisponde alla funzione indicatrice.
Osservando la formula, per ogni elemento (i, j) della matrice, corrispondente al confronto tra l'i-esimo carattere di A e il j-esimo carattere di B, viene calcolato il valore nel seguente modo:

- Se i caratteri corrispondenti agli indici i e j sono uguali, il valore di Edit Distance rimane quello assegnato ad $(i-1, j-1)$, in quanto non sono necessarie operazioni di base.
- Altrimenti, si considerano le operazioni di rimozione (prima riga), inserimento (seconda riga) e sostituzione (terza riga) e si prende quella che minimizza l'incremento del valore di Edit Distance.

Si è scelto di valutare ogni operazione di base con lo stesso impatto, infatti ciascuna aumenta di 1 il valore della Edit Distance.
Lo scopo del progetto è quello di misurare lo speedup ottenuto parallelizzando l'algoritmo rispetto alla sua versione sequenziale. Per parallelizzare è stato utilizzato OpenMP, il linguaggio di programmazione utilizzato è quindi il C++.
Tutti gli esperimenti sono stati svolti sul server ssh "papavero.dinfo.unifi.it".

1.1. Versioni sequenziali dell'algoritmo

Sono state implementate 3 versioni sequenziali dell' algoritmo:

- **Full Matrix:** sfrutta una matrice 2D per memorizzare i risultati intermedi del calcolo della distanza. Le dimensioni della matrice corrispondono alla lunghezza delle due stringhe confrontate. L'algoritmo inizia con una stringa vuota e iterativamente calcola i vari elementi della matrice riga per riga;
- **Matrix Row:** memorizza due righe della matrice per volta, riducendo quindi la complessità in termini di memoria occupata. L'algoritmo itera tra le stringhe riga per riga, memorizzando i calcoli correnti e della precedente iterazione in due vettori corrispondenti alle righe i-esima e (i - 1)esima della matrice completa;
- **Skew Diagonal:** costruisce la matrice scorrendola per diagonali oblique da destra verso sinistra. In Fig. 1 viene mostrata intuitivamente l'idea generica dell'algoritmo.

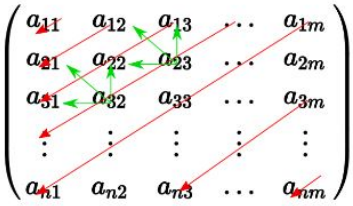

Figure 1. Le freccie rosse rappresentano le diagonali principali della matrice. Ogni elemento delle diagonali per essere calcolato necessita di 3 elementi, indicati dalle freccie verdi, che appartengono alla diagonale precedente e, di conseguenza, sono già conosciuti.

1.2. Versione parallela dell'algoritmo

L'approccio più comune per implementare Edit Distance è quello di Full Matrix, ma non è parallelizzabile perchè presenta una **race condition**, in quanto per scorrere la matrice si utilizzano due cicli for annidati, e il ciclo interno ha una dipendenza dall'indice del ciclo esterno.
Per questo si parallelizza la versione **Skew Diagonal**, in quanto ogni elemento della diagonale considerata dipende da 3 elementi appartenenti alla precedente diagonale, che è già stata calcolata all'iterazione precedente. Con questo approccio si evita quindi la race condition, consentendo la parallelizzazione.

2. Organizzazione del codice

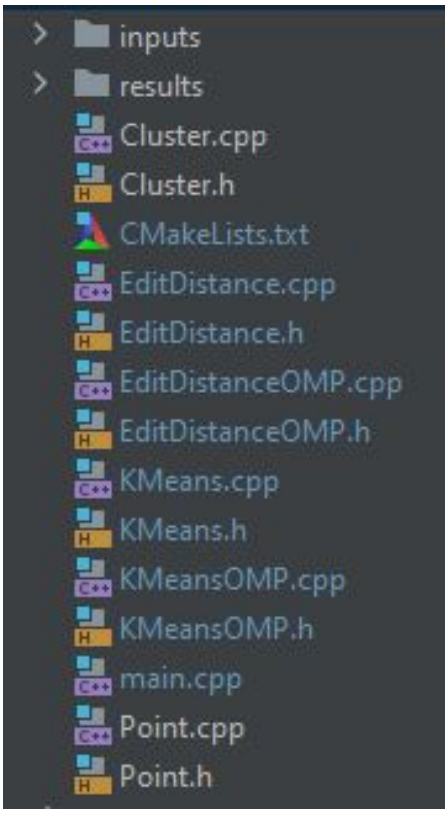

Figure 2. Organizzazione del codice.

Si è utilizzato CLion come IDE e MinGW come compilatore.
Il codice, i cui file sono mostrati in Fig. 2, presenta dei file appartenenti ad un altro progetto con directory comune, i file che interessano questo progetto sono:

- **EditDistance.h/cpp:** contengono le implementazioni sequenziali degli algoritmi;
- **EditDistanceOMP.h/cpp:** contengono l'implementazione parallela dell'algoritmo;
- **main.cpp:** contiene le funzioni relative ai test svolti ed altre funzioni ausiliarie.

In ognuna delle sequenti implementazioni, la matrice contenente i risultati di Edit Distance (o i vettori corrispondenti alle righe della matrice per Matrix Row) è stata memorizzata attraverso un **std::vector**, in modo da gestire in automatico eventuali memory leaks.

2.1. Implementazioni sequenziali

Nei file indicati in precedenza sono presenti dichiarazioni e definizioni di 3 funzioni, una per ogni versione sequenziale di Edit Distance che si è deciso di implementare.
Le dichiarazioni sono mostrate in Fig. 3.
levenshteinDistFM implementa l'approccio Full Matrix di Edit Distance, richiede in input le due stringhe da confrontare e le loro lunghezze. In output restituisce un intero, corrispondente al valore di Edit Distance tra le due stringhe considerate.
levenshteinDistSD implementa l'approccio Skew Diagonal, come in precedenza richiede in input le due stringhe da confrontare con le loro lunghezze, e restituisce un intero col valore di Edit Distance.
levenshteinDistMR implementa l'approccio Matrix Row, la funzione richiede gli stessi input delle due funzioni precedenti, e anche l'output corrisponde.

```
int levenshteinDistFM(const std::string& word1, const std::string& word2, int length1, int length2);

int levenshteinDistSD(const std::string& word1, const std::string& word2, int length1, int length2);

int levenshteinDistMR(const std::string& word1, const std::string& word2, int length1, int length2);
```

Figure 3. Dichiarazioni delle 3 versioni sequenziali implementate di Edit Distance

2.2. Implementazione parallela

Nel file indicato in precedenza sono presenti dichiarazioni e definizioni di 2 funzioni, che implementano l'approccio Skew Diagonal parallelizzando in due modi diversi. Le dichiarazioni richiedono gli stessi input delle funzioni sequenziali, con l'aggiunta di un intero che indichi il numero di thread con cui procedere alla parallelizzazione.
La differenza tra le due funzioni verrà analizzata meglio in Sec. 3.

2.3. main.cpp

Nel file sono presenti i test relativi alla parallelizzazione dell'algoritmo di Edit Distance (presentati in Sec. 4), insieme alla funzione di **main** che esegue i test, e alla funzione **random_string**. Quest'ultima è la funzione che si occupa di generare randomicamente le stringhe da confrontare con Edit Distance. Richiede in input la lunghezza desiderata per la stringa e il seed per eventualmente replicare la generazione della stringa. Per la generazione della stringa sono stati considerati tutti i caratteri alfanumerici, comprendenti quindi le lettere, sia maiuscole che minuscole, e le cifre.

3. Parallelizzazione con OpenMP

Il codice contenente l'implementazione dell'approccio Skew Diagonal è sato parallelizzato nella parte relativa al calcolo della matrice di Edit Distance. Sono presenti due cicli for, il primo scorre le diagonali, mentre il secondo calcola il valore degli elementi che appartengono alla diagonale considerata.
Sono stati considerati due possibili approcci per la parallelizzazione.

3.1. Singola parallel overhead

```
#pragma omp parallel default(none) firstprivate(dMIN, dMAX, M, N) \
shared(distMatrix, word1, word2) num_threads(threads)
{
    for (int d = dMIN; d <= dMAX; d++) {
        int iMIN = std::max(d, 1);
        int iMAX = std::min(M + d, N - 1);
        #pragma omp for
        for (int i = iMIN; i <= iMAX; i++) {
            int k = d < 0 ? 1 : -1;
            int j = M + d - i + k;
            if (word1[i - 1] != word2[j - 1]) {
                distMatrix[i * N + j] = std::min(std::min(distMatrix[(i - 1) * M + j],
                                                          distMatrix[i * M + j - 1]),
                                                 distMatrix[(i - 1) * M + j - 1]) + 1;
            } else {
                distMatrix[i * N + j] = distMatrix[(i - 1) * N + j - 1];
            }
        }
        if (d == -1) {
            d += 2;
        }
    }
}
```

Figure 4. Parallelizzazione con blocco parallel al ciclo for esterno.

Come osservabile in Fig. 4, per il primo approccio viene creata una sezione **omp parallel** in

corrispondenza del ciclo for più esterno, mentre per il ciclo for più interno viene specificata una sezione **omp for**.
I threads vengono quindi spawnati al primo ciclo for, mentre vengono poi distribuiti tra le varie iterazioni del secondo ciclo for. Così facendo, i threads vengono generati solo una volta per l'intera esecuzione dell'algoritmo, e questo teoricamente dovrebbe migliorare il tempo di esecuzione. La questione verrà esaminata in Sec. 4.1
Per quanto riguarda le clausole della sezione parallel, vengono specificati come firstprivate gli indici relativi alle diagonali della matrice e le lunghezze delle stringhe, in modo che ogni thread possa calcolare in una variabile locale gli indici per il ciclo for interno. A causa di questa dipendenza non è stato possibile specificare una clausola di collapse.
La matrice contenente i valori di Edit Distance e le due stringhe sono invece state specificate come shared.

3.2. Multiple parallel overhead

```
for (int d = dMIN; d <= dMAX; d++) {
    int iMIN = std::max(d, 1);
    int iMAX = std::min(M + d, N - 1);
    #pragma omp parallel for default(none) firstprivate(iMIN, iMAX, d, M, N) \
    shared(distMatrix, word1, word2) num_threads(threads)
    for (int i = iMIN; i <= iMAX; i++) {
        int k = d < 0 ? 1 : -1;
        int j = M + d - i + k;
        if (word1[i - 1] != word2[j - 1]) {
            distMatrix[i * N + j] = std::min(std::min(distMatrix[(i - 1) * M + j],
                                                      distMatrix[i * M + j - 1]),
                                             distMatrix[(i - 1) * M + j - 1]) + 1;
        } else {
            distMatrix[i * N + j] = distMatrix[(i - 1) * N + j - 1];
        }
    }
    if (d == -1) {
        d += 2;
    }
}
```

Figure 5. Parallelizzazione con blocco parallel al ciclo for interno.

In Fig. 5 si osserva invece come la sezione parallel e la sezione for vengano specificate contemporaneamente in corrispondenza del ciclo for più interno. Così facendo, i threads verrano generati e assegnati ad ogni iterazione del ciclo for esterno, e questo dovrebbe risultare in tempi computazionali più elevati rispetto al precedente approccio.
Osservando le clausole della sezione parallel for, rispetto a prima sono indicati come firstprivate anche gli indici del ciclo for interno, mentre le variabili shared sono identiche.

4. Esperimenti svolti

Tornando al file main.cpp, le funzioni ausiliarie scritte per i test sono le seguenti:

- testStringSearchFMTime: richiede in input le due stringhe da confrontare, e misura il tempo impiegato dall'algoritmo Full Matrix per eseguire il calcolo di Edit Distance.
  In input si può specificare anche un booleano, repeat, che se impostato a true ripete il test 10 volte e fa la media dei risultati temporali ottenuti;

- testStringSearchSDTime: misura il tempo computazionale dell'algoritmo Skew Diagonal, gli input corrispondono alla funzione precedente;

- testStringSearchMRTime: misura il tempo computazionale dell'approccio Matrix Row, nuovamente gli input sono identici;

- testStringSearchSD_OMPTime: misura il tempo impiegato dall'algoritmo Skew Diagonal parallelizzato col primo approccio, rispetto alle funzioni precedenti in input richiede anche un intero, che indica il numero di threads usati per la parallelizzazione;

- testStringSearchSD_OMP2Time: come la funzione precedente, ma utilizzando il secondo approccio per la parallelizzazione di Skew Diagonal;

- **compareTimeStringSearch**: esegue le 5 funzioni precedenti, e confronta i risultati temporali degli algoritmi sequenziali con quelli paralleli, salvando i valori corrispondenti di speedup in un vettore.
  Il vettore viene poi restituito in output;

- **compareTimeStringSearchSD**: confronta i tempi computazionali di Full Matrix e Skew Diagonal parallelizzato col primo approccio, salvando gli speedup in un vettore che verrà

reso in output.
I motivi per cui vengono confrontati specificamente questi due algoritmi sono indicati in Sec. 4.1;

- **testEditDistance**: funzione in cui sono esplicati i test veri e propri, viene eseguita dal main.

In generale i parametri tenuti in considerazione per gli esperimenti sono il numero di threads usati per la parallelizzazione e la lunghezza delle stringhe confrontate, oltre ai diversi approcci presentati per Edit Distance.
I risultati vengono riprodotti per 10 volte e poi ne viene fatta la media, quindi vengono salvati in appositi file csv contenuti nella cartella results. Eventuali grafici vengono generati tramite del codice python (non reso disponibile).
I risultati dei test vengono salvati in appositi file csv contenuti nella cartella **results**.
Di seguito vengono presentati i test svolti sugli algoritmi di Edit Distance.

4.1. Test1: variare length su vari approcci

Con questo primo test si vuole andare a confrontare i 3 approcci sequenziali con le due versioni di parallelizzazione. Per il confronto vengono considerate delle lunghezze di stringa variabili (come indicato in Tab. 1), mentre il numero di threads viene lasciato costante a 16.
Non ci sono particolari aspettative per questo test, l'unica cosa che potremmo ipotizzare è che l'approccio sequenziale Matrix Row sia il più complicato da migliorare (come tempo computazionale), in quanto è quello che utilizza meno memoria.
Per quanto rigurarda le due versioni parallele, ci aspetteremmo di osservare quanto supposto in Sec. 3 al momento di presentare le due parallelizzazioni.

Partendo dal confronto tra i 3 algoritmi sequenziali, si nota che per ogni lunghezza considerata lo speedup è presente, segno dell'effettiva utilità della parallelizzazione con lunghezze comparabili con quelle considerate nella tabella. L'approccio sequenziale Skew Diagonal risulta essere la tecnica che impiega più tempo di esecuzione, mentre come preventivato Matrix Row risulta essere la più veloce delle tre, anche se comunque non ha prestazioni paragonabili alla parallelizzazione.
Osserviamo anche come aumentando la lunghezza delle stringhe lo speedup cresca di conseguenza, indicando che la parallelizzazione riesce attraverso l'assegnazione del lavoro ai threads a gestire meglio le strutture dati contenenti i calcoli intermedi per Edit Distance (in Sec. 4.2 questo aspetto verrà meglio analizzato).
Infine, osservando il confronto tra i due approcci paralleli, si nota che effettivamente l'approccio con una singola parallel overhead risulta più veloce, anche se i miglioramenti sono piuttosto contenuti, soprattutto all'aumentare della lunghezza delle stringhe confrontate (con una lunghezza di 28000 vi è una differenza di circa 0.1 tra i due speedup).
Questo porta a pensare che con un numero inferiore di threads potenzialmente l'approccio con più parallel overhead possa avere prestazioni migliori, perchè avendo più overhead verrà speso più tempo per la gestione dei threads (generazione e assegnazione del lavoro), ma diminuendo il numero di threads andiamo a limitare questo impatto.
Perciò è stato eseguito nuovamente il codice con un numero di thread pari a 4.

	FM	SD	MR
P1, len=5k	3.2071	3.70324	2.31112
P2, len=5k	2.85164	3.29279	2.05497
P1, len=15k	3.3498	5.15474	2.43479
P2, len=15k	3.18759	4.90512	2.31688
P1, len=28k	3.63548	6.48565	2.58662
P2, len=28k	3.52203	6.28326	2.5059

Table 1. Tabella con i valori di speedup al variare di algoritmo sequenziale (colonne, FM indica Full Matrix, SD Skew Diagonal e MR Matrix Row), algoritmo parallelo(P1 indica il primo approccio, P2 il secondo) e lunghezza delle parole(indicata con len).

Osservando i risultati presenti in Tab. 2 si nota che stavolta lo speedup maggiore si ottiene, seppur lievemente, col secondo approccio parallelo.
Tra i due approcci paralleli non sembra esserci

	FM	SD	MR
P1, len=5k	1.94831	2.22286	1.3185
P2, len=5k	2.02005	2.3047	1.36705
P1, len=15k	1.85594	2.8919	1.34012
P2, len=15k	1.8461	2.87656	1.33301
P1, len=28k	1.84884	3.40718	1.33429
P2, len=28k	1.86523	3.43739	1.34611

Table 2. Tabella con i valori di speedup al variare di algoritmo sequenziale, algoritmo parallelo e lunghezza delle parole (come Tab. 1) usando 4 threads per la parallelizzazione.

quindi una differenza significativa, nel proseguo dei test si è scelto di utilizzare l'implementazione con singola parallel overhead, in quanto ha prestazioni leggermente maggiori usando più threads. Tra gli obiettivi dei test c'è anche quello di osservare le prestazioni sfruttando il più possibile l'hardware, e scegliendo questo approccio ci garantiamo del tempo computazionale minore.
Per quanto riguarda la scelta degli algoritmi sequenziali, si è deciso di utilizzare l'approccio Full Matrix, non solo perchè è il più diffuso a livello pratico, ma anche perchè dai risultati è quello che offre prestazioni nella media rispetto alle altre due versioni sequenziali.

4.2. Test2: variare length

Per il secondo test si va a variare la lunghezza delle stringhe, tenendo costante il numero di threads ad 8 per concentrarsi il più possibile sull'effettivo impatto del parametro di lunghezza delle stirnghe.
Ciò che ci aspettiamo è che all'aumentare della lunghezza delle stringhe, aumenti anche il valore di speedup, in quanto andiamo ad influire sulla complessità dell'algoritmo sequenziale di Edit Distance, che generalmente è:

$$\mathcal{O}(n * m) \tag{2}$$

Nella Eq. (2) n e m sono le lunghezze delle due stringhe da comprarare.

Osservando i risultati in Fig. 6, quello che ci aspettavamo si è in parte verificato, in quanto fino ad una lunghezza di 12000 la crescita dello speedup è costante, mentre da 20000 in poi si stabilizza su un valore di 2.6 circa (con un lieve calo a 2.4 con una lunghezza di 32000).

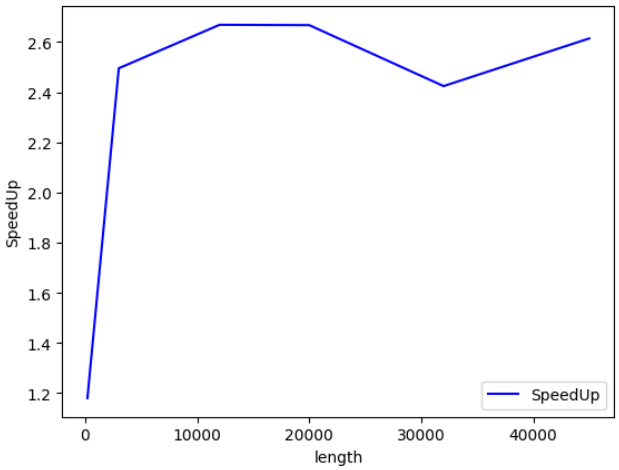

Figure 6. Valori di speedup all'aumentare della lunghezza delle stringhe confrontate. Le lunghezze considerate sono 200, 3000, 12000, 20000, 32000 e 45000.

Possiamo ipotizzare che questo avviene per la saturazione dello heap, che ad esempio gestisce male l'allocazione di una matrice 32000x32000. Nemmeno con la divisione del lavoro tra i vari threads si riesce a migliorare la situazione, anche se da Tab. 1 possiamo osservare che con lunghezza di 28000 e 16 threads abbiamo uno speedup di 3.5 circa.
Una possibile soluzione è quindi quella di aumentare il numero di threads che lavorano per il calcolo della matrice di Edit Distance, almeno fino ad arrivare ad un certo valore oltre al quale avremo saturazione anche della CPU (rendendo quindi inutile anche aumentare il numero di threads).

4.3. Test3: variare nThreads

Infine, per l'ultimo test si va a valutare l'impatto del parametro relativo al numero di threads, tenendo costante a 20000 la lunghezza delle stringhe. Il numero di threads viene fatto variare tra 2 e 16 considerando i multipli di 2.
Ciò che ci aspettiamo è che all'aumentare del numero di threads aumentino anche le prestazioni dell'algoritmo parallelizzato, portando quindi a dei valori di speedup crescenti.

Come si nota dai risultati in Fig. 7, effettivamente all'aumentare del numero di threads au-

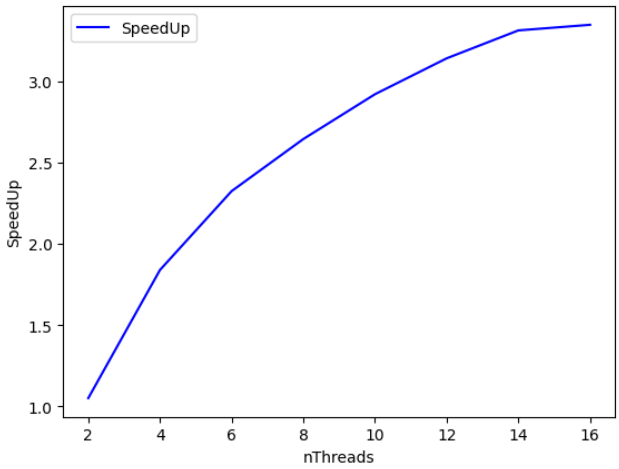

Figure 7. Valori di speedup ottenuti variando il numero di threads utilizzati per la versione parallelizzata di Edit Distance.

menta anche lo speedup tra l'algoritmo sequenziale e quello parallelo. Ovviamente l'aumento di speedup non è infinito, si arriverà ad un valore di soglia per il numero di threads oltre al quale la CPU risulterà saturata.