Elaborato Parallel Computing: Levenshtein Distance

Lorenzo Bucci
lorenzo.bucci@stud.unifi.it

Abstract

Il progetto ha lo scopo di parallelizzare un algoritmo che permetta di calcolare la distanza di Levenshtein tra due stringhe di testo. La parallelizzazione è stata implementata con due framework diversi: Java e OpenMP. Entrambi i framework sfruttano i diversi core della CPU per la parallelizzazione. Lo scopo principale del progetto è dimostrare la maggiore velocità nell'eseguire l'algoritmo parallelo rispetto a quello sequenziale e calcolarne lo speed-up.

1. Introduzione

Questo elaborato prevede lo sviluppo di un algoritmo parallelo, a partire da un algoritmo sequenziale esistente, che permetta di calcolare la distanza tra due stringhe di testo usando la definizione di distanza introdotta da Vladimir Levenshtein nel 1965.

La distanza è così definita: *date due stringhe A e B, la loro distanza è il numero minimo di modifiche elementari che consentono di trasformare la A nella B. Le modifiche elementari ammesse sono:*

- *cancellazione di un carattere*
- *sostituzione di un carattere*
- *inserimento di un carattere*

Un algoritmo iterativo che permetta di calcolare la distanza tra due stringhe è stato inventato da Robert A. Wagner e Michael J. Fischer nel 1974. Questo algoritmo sarà la base di partenza del progetto.

1.1. Algoritmo di Wagner e Fischer

Date due stringhe *A* e *B* di lunghezza *lenA* e *lenB*, l'algoritmo itera su una matrice *D* di dimensioni (*lenA*+1)x(*lenB*+1) per calcolare la distanza. (Da qui in poi gli indici delle matrici verranno sempre considerati come *zero-based*).

Nella fase di inizializzazione l'algoritmo inserisce nella prima riga di *D* i valori da 0 a *lenB* e nella prima colonna i valori da 0 a *lenA*.

Successivamente l'algoritmo itera sequenzialmente sui restanti elementi della matrice: dall'elemento *D(1,1)* all'elemento *D(lenA,lenB)*. Per ogni iterazione, in *D(i,j)* viene calcolato un valore dipendente dalle due stringhe *A* e *B* ma soprattutto da *D(i-1,j)*, *D(i,j-1)*, *D(i-1,j-1)*.

Terminate le iterazioni, l'elemento *D(lenA,lenB)* conterrà la distanza finale tra la stringa *A* e la stringa *B*.

Di seguito è riportato l'algoritmo in pseudocodice.

```
int LevenshteinDistance (char A[1..lenA],
                         char B[1..lenB])
  declare int D[0.. lenA, 0.. lenB]
  declare int i1, i2, cost

  for i1 from 0 to lenA
     D[i1, 0] := i1
  for i2 from 0 to lenB
     D[0, i2] := i2

  for i1 from 1 to lenA
     for i2 from 1 to lenB
        if A[i1 - 1] = B[i2 - 1] then cost:= 0
        else cost:= 1
        D[i1, i2] := minimum(
                     D[i1 - 1, i2] + 1,
                     D[i1, i2 - 1] + 1,
                     D[i1 - 1, i2 - 1] + cost  )

  return D[lenA, lenB]
```

Algoritmo 1. Calcolo della distanza di Levenshtein mediante l'algoritmo di Wagner e Fischer.

1.2. Parallelizzazione

L'algoritmo è per sua natura sequenziale e per questo non può essere completamente parallelizzato a causa delle dipendenze presenti su valori precedentemente calcolati nella matrice *D*, così come riportato in 1.1.

Da un'attenta analisi delle dipendenze e della struttura dell'algoritmo si può stabilire che l'unica parallelizzazione possibile è il calcolo degli elementi della matrice *D* che si trovano sulle stesse linee diagonali della matrice.

La Tabella 1 mostra un esempio della matrice *D* usata per calcolare la distanza tra due parole d'esempio. In generale, le otto diagonali presenti (rappresentate in verde) devono essere calcolate in modo sequenziale tra loro, ma niente impone che gli elementi al loro interno possano essere calcolati contemporaneamente. Come si evince dalla rappresentazione, la parallelizzazione non può essere uniforme ma sarà massima nella diagonale centrale (più lunga) e minima nelle diagonali iniziali e finali (più corte).

		C	*I*	*E*	*L*	*O*
	0	1	2	3	4	5
C	1	0	1	2	3	4
I	2	1	0	1	2	3
A	3	2	1	1	2	3
O	4	3	2	2	2	2

Tabella 1. Rappresentazione della matrice *D* per il calcolo della distanza tra le stringhe “ciao” e “cielo”. Gli elementi con sfondo grigio sono stati inseriti durante la fase di inizializzazione. Le linee verdi rappresentano le diagonali della matrice i cui elementi possono essere calcolati in parallelo.

$D_{0,0}$ **0**	$D_{0,1}$ **2**	$D_{0,2}$ **5**	$D_{0,3}$ **9**
$D_{1,0}$ **1**	$D_{1,1}$ **4**	$D_{1,2}$ **8**	$D_{1,3}$ **13**
$D_{2,0}$ **3**	$D_{2,1}$ **7**	$D_{2,2}$ **12**	$D_{2,3}$ **17**
$D_{3,0}$ **6**	$D_{3,1}$ **11**	$D_{3,2}$ **16**	$D_{3,3}$ **20**
$D_{4,0}$ **10**	$D_{4,1}$ **15**	$D_{4,2}$ **19**	$D_{4,3}$ **22**
$D_{5,0}$ **14**	$D_{5,1}$ **18**	$D_{5,2}$ **21**	$D_{5,3}$ **23**

Tabella 2. Rappresentazione della matrice *D* divisa in 24 sottomatrici. Il numero in apice identifica la sequenza con cui i task associati vengono lanciati dall’algoritmo parallelo.

Seguendo un approccio naïve si può pensare di generare un numero di task pari al numero di elementi da calcolare e lanciare i task una volta che le dipendenze risultano soddisfatte. Per cui si otterrebbero $lenA * lenB$ task a cui andrebbero associati altrettanti elementi di sincronizzazione (ad es. semafori).

Durante lo sviluppo dell’elaborato si è verificato sperimentalmente che la soluzione naïve non porta alcun vantaggio rispetto alla versione sequenziale: fintanto che *lenA* e *lenB* sono piccoli, si ha un eccessivo overhead; se *lenA* e *lenB* crescono, il numero di task diventa enorme e i troppi context switch sovraccaricano inutilmente la CPU.

Una soluzione che risolve entrambi i problemi è diminuire il numero totale dei task. Un modo per fare ciò è “sequenzializzare” nuovamente il problema raggruppando gli elementi della matrice *D* in sottomatrici. Le sottomatrici verranno calcolate in parallelo tra di loro, i singoli elementi al loro interno in modo sequenziale.

1.3. Algoritmo parallelo con sottomatrici

L’algoritmo parallelo ideato in questo progetto divide la matrice *D* in sottomatrici di uguali dimensioni per evitare di andare incontro alle complicazioni di un’implementazione naïve, descritte in 1.2.

L’utilizzo delle sottomatrici pone però il problema della sincronizzazione: il calcolo di alcuni elementi delle sottomatrici dipende fortemente da altri elementi di altre sottomatrici. Avere un elemento di sincronizzazione per ogni valore della matrice *D* implicherebbe un largo spreco di memoria, specialmente quando *lenA* e *lenB* sono grandi. Si è così pensato di sincronizzare direttamente le sottomatrici: il task associato alla sottomatrice *i,j* può essere lanciato solo quando i task relativi alle sottomatrici *i-1,j; i,j-1* e *i-1,j-1* sono stati completati.

Come riporta la Tabella 2, per diminuire il più possibile il tempo di spin, si è provveduto a sviluppare l’algoritmo parallelo in modo che i task delle sottomatrici fossero lanciati iterando diagonalmente. Questo schema permette di risolvere molto più velocemente le dipendenze rispetto a lanciare i task iterando sulle righe.

Sulle dimensioni da assegnare alle sottomatrici verrà discusso in 2.2.

2. Struttura del progetto

Il progetto è stato sviluppato usando due framework diversi: Java e OpenMP. In entrambe le implementazioni è presente sia l’algoritmo sequenziale che quello parallelo. Ambedue le funzioni che gestiscono gli algoritmi sono chiamate all’interno di una funzione *main* che misura i loro tempi di esecuzione e stampa a video i risultati.

2.1. Implementazione sequenziale

Java, OpenMP. In entrambi i framework l’implementazione sequenziale sviluppata ricalca l’Algoritmo 1, su cui si ha essenzialmente una traduzione dello pseudocodice in linguaggio Java e C++.

2.2. Implementazione parallela

Le implementazioni parallele sviluppate per i due framework seguono l’idea dell’algoritmo parallelo con sottomatrici descritto in 1.3.

Dai test effettuati durante lo sviluppo è risultato palese come la scelta delle dimensioni delle sottomatrici influenzasse la velocità di esecuzione del programma parallelo. Si è quindi deciso di far sì che fosse la funzione chiamante (il *main*) a passare il parametro delle dimensioni delle sottomatrici alla funzione sottostante. In questo modo il *main* può eseguire diverse volte varianti dell’algoritmo parallelo e confrontare i risultati ottenuti.

Analizziamo adesso le due implementazioni che sono state sviluppate nei rispettivi framework.

Java. La versione Java dell’algoritmo fa uso della classe *SubMatrixCalculator* che implementa *Runnable*. Questa classe rappresenta il task associato ad ogni sottomatrice. Al suo interno è presente un solo metodo *run()* che sarà chiamato quando il task verrà lanciato. Il metodo verifica che le dipendenze siano state soddisfatte e calcola in modo sequenziale i valori della matrice *D* nell’intervallo assegnatoli. L’intervallo non è altro che la posizione della sottomatrice all’interno della matrice *D*. Il calcolo viene effettuato attraverso i due loop annidati dell’Algoritmo 1. Una volta che il task ha terminato, risveglia gli altri task che aspettavano la sua terminazione.

La sincronizzazione è stata implementata impiegando i *CountDownLatch* di Java. I latch sono thread-safe e hanno al loro interno un valore intero impostato in fase di inizializzazione. Un task può chiamare su un latch il metodo *countDown()* che decrementa di 1 il valore associato al latch. Altri task possono aver chiamato il metodo *await()* dello stesso latch: fintanto che il valore del latch non è uguale a 0 i task rimarranno in attesa (senza spin lock), quando verranno risvegliati i task potranno proseguire con l'istruzione successiva.

Nella funzione dell'algoritmo parallelo Java è stata creata una matrice di latch idealmente associati ad ogni sottomatrice. Ciascun latch è inizializzato ad 1. Nel momento in cui un task di una sottomatrice viene lanciato, chiama *await()* sui latch associati alle sottomatrici da cui dipende, quando il task termina chiama *countDown()* sul latch associato alla sua sottomatrice.

Il calcolo di una sottomatrice non può partire finché le dipendenze non sono state risolte altrimenti il task andrà in attesa e sarà risvegliato solo quando gli altri task avranno terminato.

La funzione dell'algoritmo parallelo inizializza la matrice *D*, la matrice dei latch e lancia i task delle sottomatrici seguendo lo schema delle diagonali descritto in 1.3. Il lancio viene gestito da un *ExecutorService* su un *FixedThreadPool* con il numero di thread pari al numero di core della CPU.

OpenMP. A differenza dell'implementazione Java, nella versione OpenMP del progetto si è deciso di parallelizzare sia l'inizializzazione che il calcolo effettivo. Questo perché la parallelizzazione dell'inizializzazione è banale in OpenMP.

Lo schema di parallelizzazione del calcolo è il seguente: si divide idealmente la matrice *D* nelle sue sottomatrici come in Tabella 2 e si lanciano in parallelo i task delle sottomatrici di ogni diagonale. Per rispettare le dipendenze si fa uso di una barriera (implicita di OpenMP) tra una diagonale e l'altra, ma le sottomatrici che appartengono alla stessa diagonale sono calcolate in parallelo.

Sebbene più semplice di Java, la sincronizzazione della versione OpenMP è leggermente più lenta visto che si sincronizzano le diagonali e non le singole matrici. Purtroppo OpenMP, essendo un framework di parallelizzazione implicita, non mette a disposizione strumenti di sincronizzazione avanzati come quelli presenti in Java.

3. Risultati

Nei prossimi paragrafi saranno presentati i risultati sperimentali del progetto che dimostreranno come l'implementazione parallela sia migliore rispetto a quella sequenziale in entrambi i framework. Infine, sarà calcolato lo speed-up che concretizzerà numericamente il guadagno ottenuto rispetto alla versione sequenziale.

I risultati mostrati varieranno sulla base della lunghezza delle stringhe e, in particolare per l'algoritmo parallelo, varieranno in base alla dimensione delle sottomatrici così da evidenziarne le differenze. Da alcuni test preliminari è risultato che le dimensioni delle sottomatrici "migliori" (in termini di velocità) fossero nell'intervallo [10,1000]. Per questo, sono stati effettuati 10 test con 10 valori diversi in quell'intervallo.

Sono stati svolti anche alcuni test per sperimentare un diverso numero di thread nei due framework.

Le stringhe su cui è calcolata la distanza sono generate casualmente utilizzando i caratteri nell'intervallo [a-z].

Tutti i risultati sono stati ottenuti su un hardware basato su CPU Intel Core i5-9400 (6 core) con 16 GB di RAM DDR4 a 2666 MHz.

3.1. Risultati temporali Java

Il tempo di esecuzione degli algoritmi è stato misurato in millisecondi impiegando la funzione *nanoTime()* della libreria Java.

In entrambe le versioni degli algoritmi si è misurato il tempo dell'intera funzione che li gestisce: dall'inizializzazione al ritorno del valore della distanza.

Dalla Tabella 3 risulta che si ha sempre un vantaggio nell'eseguire l'algoritmo parallelo tranne quando si costruiscono sottomatrici di dimensioni molto piccole (ad es. 10x10). Per i restanti valori di dimensioni non si evidenzia una significativa varianza: se in specifici problemi fossero presenti dei vincoli sulle dimensioni da assegnare alle sottomatrici l'eventuale perdita (o guadagno) sarebbe pressoché nulla tra un valore e l'altro. In ogni caso, i valori più performanti appartengono all'intervallo [310,510].

A differenza della versione OpenMP, i test si sono interrotti quando la lunghezza delle stringhe ha raggiunto l'ordine dei 30000 caratteri, limite tecnico del tipo *short* in Java (*unsigned short*: non presenti in Java, *int*: occupazione di memoria eccessiva).

Con $lenA = 32041$, $lenB = 30137$ e la dimensione delle sottomatrici fissata a 510, in Tabella 6 sono riportate le velocità di esecuzione dell'algoritmo parallelo con un diverso numero di thread. Seppur di poco rispetto a 12, il numero di thread ideale è pari a 6 (numero di core della CPU). Il tempo di esecuzione risulta quasi doppio con il numero dei thread dimezzato a 3.

3.2. Risultati temporali OpenMP

Il tempo di esecuzione degli algoritmi è stato misurato in millisecondi impiegando funzioni standard della libreria del C++11.

lenA	*lenB*	Implementazione sequenziale	Implementazione parallela (al variare della dimensione delle sottomatrici)									
			10	110	210	310	410	510	610	710	810	910
5123	5047	178	239	49	50	48	53	73	49	52	56	71
10020	10769	683	972	215	199	133	135	156	200	199	146	169
21374	20922	2735	3166	697	712	634	597	677	673	686	658	757
32041	30137	5823	9703	1673	1479	1430	1400	1371	1383	1431	1373	1412

Tabella 3. Tempi di esecuzione in millisecondi delle due implementazioni Java per la misura della distanza di Levenshtein tra due stringhe generate casualmente di dimensioni *lenA* e *lenB*. In verde sono indicati i migliori tempi registrati nell'implementazione parallela al variare della dimensione delle sottomatrici.

lenA	*lenB*	Implementazione sequenziale	Implementazione parallela (al variare della dimensione delle sottomatrici)									
			10	110	210	310	410	510	610	710	810	910
5123	5047	109	62	47	47	31	28	31	31	31	31	31
10020	10769	422	234	125	109	94	94	109	94	109	109	125
21374	20922	1687	734	422	391	391	375	391	391	406	469	469
32041	30137	3625	2000	922	844	797	781	812	781	797	797	828
40245	41673	6312	4234	1562	1437	1391	1375	1422	1391	1391	1422	1406
52389	51024	10062	6781	2531	2265	2250	2203	2156	2140	2156	2178	2281
60157	63704	14484	10640	3875	3359	3344	3187	3172	3125	3125	3233	3172

Tabella 4. Tempi di esecuzione in millisecondi delle due implementazioni OpenMP per la misura della distanza di Levenshtein tra due stringhe generate casualmente di dimensioni *lenA* e *lenB*. In verde sono indicati i migliori tempi registrati nell'implementazione parallela al variare della dimensione delle sottomatrici.

lenA	*lenB*	Speed-up versione Java										Speed-up versione OpenMP									
		10	110	210	310	410	510	610	710	810	910	10	110	210	310	410	510	610	710	810	910
5123	5047	0,7	3,6	3,6	3,7	3,4	2,4	3,6	3,4	3,2	2,5	1,8	2,3	2,3	3,5	3,9	3,5	3,5	3,5	3,5	3,5
10020	10769	0,7	3,2	3,4	5,1	5,1	4,4	3,4	3,4	4,7	4,0	1,8	3,4	3,9	4,5	4,5	3,9	4,5	3,9	3,9	3,4
21374	20922	0,9	3,9	3,8	4,3	4,6	4,0	4,1	4,0	4,2	3,6	2,3	4,0	4,3	4,3	4,5	4,3	4,3	4,2	3,6	3,6
32041	30137	0,6	3,5	3,9	4,1	4,2	4,2	4,2	4,1	4,2	4,1	1,8	3,9	4,3	4,5	4,6	4,5	4,6	4,5	4,5	4,4
40245	41673											1,5	4,0	4,4	4,5	4,6	4,4	4,5	4,5	4,4	4,5
52389	51024											1,5	4,0	4,4	4,5	4,6	4,7	4,7	4,7	4,6	4,4
60157	63704											1,4	3,7	4,3	4,3	4,5	4,6	4,6	4,6	4,5	4,6

Tabella 5. Speed-up del programma nelle due versioni Java e OpenMP al variare della dimensione delle sottomatrici. In verde sono indicati i valori di speed-up relativi alle dimensioni delle sottomatrici più performanti descritte in 3.1 e 3.2.

	Java $lenA = 32041$, $lenB = 30137$, dim. sottomatrici = 510			**OpenMP** $lenA = 32041$, $lenB = 30137$, dim. sottomatrici = 410		
Numero di thread	3	6	12	3	6	12
Tempo di esecuzione	2360	1371	1411	1453	781	797

Tabella 6. Tempi di esecuzione in millisecondi delle due versioni del programma per la misura della distanza di Levenshtein tra due stringhe generate casualmente di dimensioni lenA e lenB al variare del numero dei thread.

In entrambe le versioni degli algoritmi si è misurato il tempo dell'intera funzione che li gestisce: dall'inizializzazione al ritorno del valore della distanza.

Nella misura non è stata considerata né l'allocazione di memoria necessaria alla matrice *D*, né la sua deallocazione.

In Tabella 4 si osserva che l'implementazione parallela è sempre migliore rispetto a quella sequenziale, ma le dimensioni delle sottomatrici che minimizzano più di tutte il tempo di esecuzione sono 410 e 610. Come nella versione Java, la varianza è molto bassa tra una dimensione e l'altra.

In Tabella 6 sono riportate le velocità di esecuzione dell'algoritmo parallelo con un diverso numero di thread. Così come in Java, il numero di thread migliore è pari a 6 leggermente più performante rispetto a 12. Il tempo di esecuzione raddoppia quando il numero dei thread si dimezza a 3.

3.3. Speed-up

In Tabella 5 sono riportati i valori dello speed-up relativi ai risultati ottenuti nelle Tabella 3 e Tabella 4. La versione sequenziale è stata ovviamente eseguita su un solo core, mentre l'implementazione parallela sui 6 core della CPU indicata in 3.

In termini di speed-up, la versione Java è risultata leggermente meno performante rispetto alla versione OpenMP. Tuttavia, scegliendo adeguatamente le dimensioni delle sottomatrici negli intervalli discussi in 3.1 e 3.2 (ed evidenziati in verde nella Tabella 5), le due versioni assumono valori di speed-up molto simili.

Si evidenziano inoltre minori valori di speed-up quando la lunghezza delle stringhe è nell'ordine dei 5000 caratteri. Ciò accade perché, quando si considerano stringhe di piccole dimensioni, l'overhead dell'algoritmo parallelo diventa significativo: se le dimensioni fossero eccessivamente piccole il guadagno sarebbe nullo.

Se la scelta delle dimensioni delle sottomatrici è corretta e non si opera su stringhe di piccole dimensioni, lo speed-up medio su una CPU a 6 core è circa 4,5 per entrambe le versioni del programma.

Lo speed-up dell'algoritmo parallelo presentato in questo progetto segue quindi una tendenza sub-lineare rispettando le premesse descritte in 1.2 sull'impossibilità di parallelizzare interamente l'algoritmo di Wagner e Fischer.