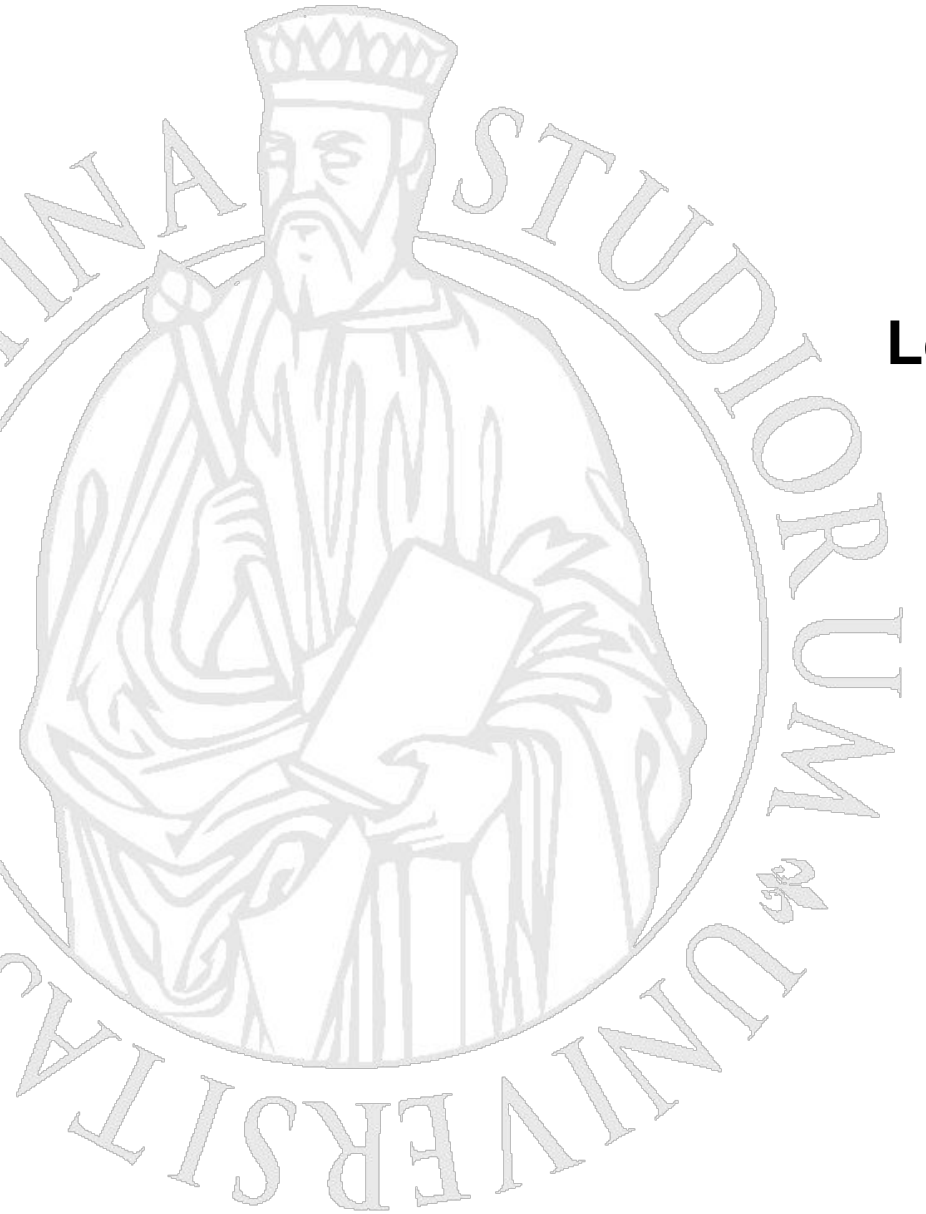


Levenshtein distance

Bellezza Francesco
Bertini Marco

07/09/2023

Il codice sequenziale

```
int m = string1.length()+1;
int n = string2.length()+1;

vector<int> mat(m * n,0);

for (int i = 0; i < m; i++)
{
    mat.at(i * n) = i;
}

for (int j = 0; j < n; j++)
{
    mat.at(j) = j;
}
```

→ Inizializzazione prima riga della matrice.

→ Inizializzazione prima colonna della matrice.

Il codice sequenziale

```
int costoperation = 0;
for (int j = 1; j < n; j++)
{
    for (int i = 1; i < m; i++)
    {
        if (string1[i-1] == string2[j-1])
        {
            costoperation = 0;
        }
        else
        {
            costoperation = 1;
        }
        mat.at(i * n + j) = min(min(mat[(i-1) * n + j] + 1, mat[i * n + j - 1] + 1), mat[(i-1) * n + j - 1] + costoperation);
    }
}
return mat[m * n - 1];
```

- Con il doppio ciclo for itero sulla matrice delle distanze.
- Calcolo il costo dell'operazione di sostituzione (è uguale a 1 se le due stringhe nelle posizioni i-1 e j-1 hanno carattere diverso, 0 altrimenti.

Il codice sequenziale

```
int costoperation = 0;
for (int j = 1; j < n; j++)
{
    for (int i = 1; i < m; i++)
    {
        if (string1[i-1] == string2[j-1])
        {
            costoperation = 0;
        }
        else
        {
            costoperation = 1;
        }
        mat.at(i * n + j) = min(min(mat[(i-1) * n + j] + 1, mat[i * n + j - 1] + 1), mat[(i-1) * n + j - 1] + costoperation);
    }
}
return mat[m * n - 1];
```

- Con il doppio ciclo for itero sulla matrice delle distanze.
- Calcolo il costo dell'operazione di sostituzione (è uguale a 1 se le due stringhe nelle posizioni i-1 e j-1 hanno carattere diverso, 0 altrimenti.
- Calcolo la distanza in posizione (i,j) della matrice in base alla distanza minima tra le due stringhe effettuando eventuali inserimenti, cancellazioni o sostituzioni.

Il codice sequenziale

```
int costoperation = 0;
for (int j = 1; j < n; j++)
{
    for (int i = 1; i < m; i++)
    {
        if (string1[i-1] == string2[j-1])
        {
            costoperation = 0;
        }
        else
        {
            costoperation = 1;
        }
        mat.at(i * n + j) = min(min(mat[(i-1) * n + j] + 1, mat[i * n + j - 1] + 1), mat[(i-1) * n + j - 1] + costoperation);
    }
}
return mat[m * n - 1];
```

Perché anche se abbiamo parlato di matrice, utilizziamo un array?

- Con il doppio ciclo for itero sulla matrice delle distanze.
- Calcolo il costo dell'operazione di sostituzione (è uguale a 1 se le due stringhe nelle posizioni i-1 e j-1 hanno carattere diverso, 0 altrimenti.
- Calcolo la distanza in posizione (i,j) della matrice in base alla distanza minima tra le due stringhe effettuando eventuali inserimenti, cancellazioni o sostituzioni.

Il codice sequenziale

Abbiamo utilizzato un vettore linearizzato per memorizzare una matrice. Per accedere alla posizione (i,j) della matrice è sufficiente accedere alla posizione (i*n)+j , dove n sono il numero di colonne della matrice.

1	2	3	4
5	6	7	8
9	10	11	12

1	2	3	4	5	6	7	8	9	10	11	12

Il codice sequenziale

Abbiamo utilizzato un vettore linearizzato per memorizzare una matrice. Per accedere alla posizione (i,j) della matrice è sufficiente accedere alla posizione (i*n)+j , dove n sono il numero di colonne della matrice.

Matrice di dimensione 3x4

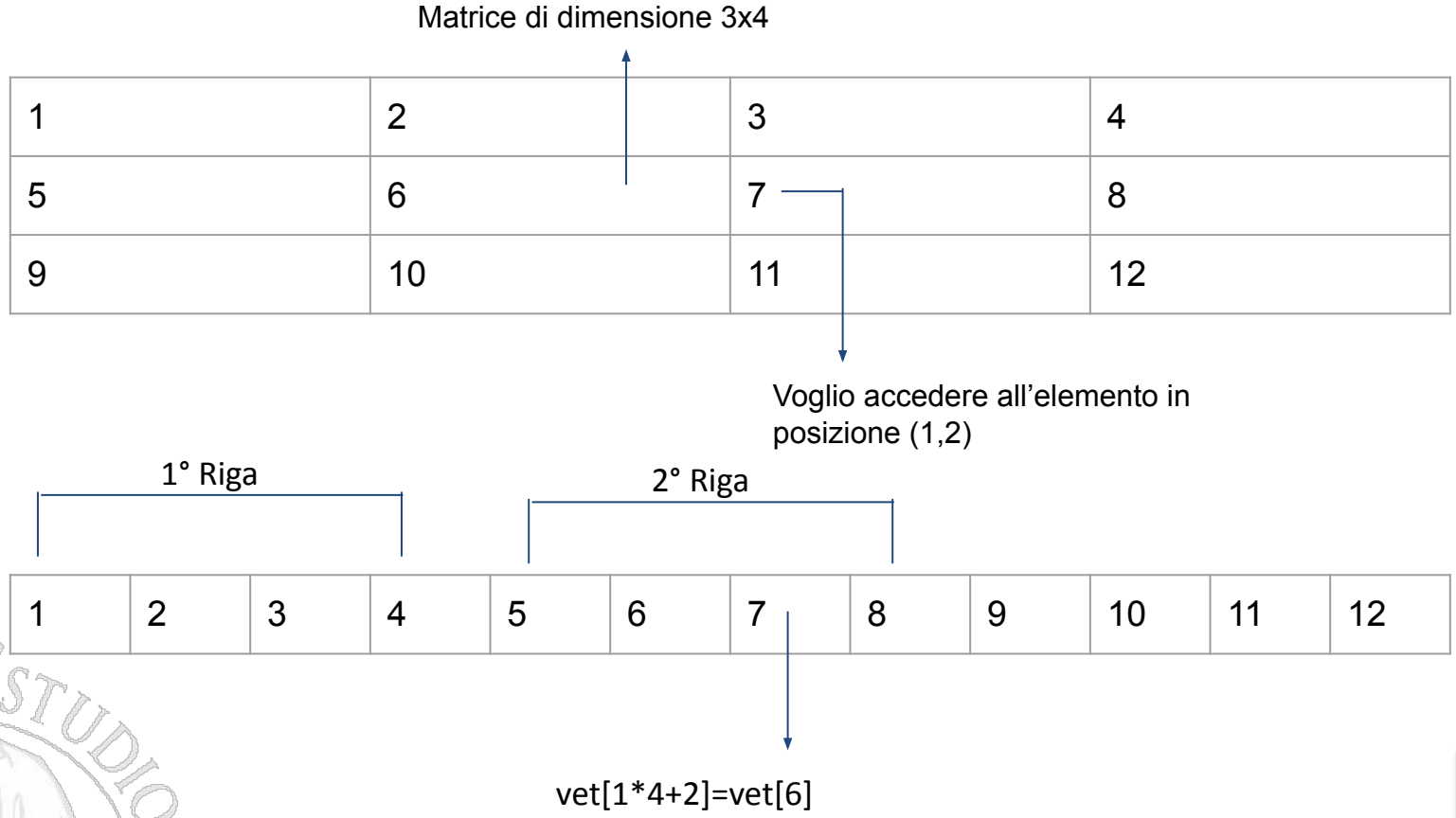
1	2	3	4
5	6	7	8
9	10	11	12

Voglio accedere all'elemento in posizione (1,2)

1	2	3	4	5	6	7	8	9	10	11	12

Il codice sequenziale

Abbiamo utilizzato un vettore linearizzato per memorizzare una matrice. Per accedere alla posizione (i,j) della matrice è sufficiente accedere alla posizione (i*n)+j , dove n sono il numero di colonne della matrice.



L’idea dietro il codice parallelo

- L’algoritmo presenta implicitamente molte dipendenze, quindi non è semplice da parallelizzare.
- L’idea consiste nel parallelizzare il calcolo degli elementi della matrice delle distanze all’interno di una anti-diagonale della matrice.

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

L’idea dietro il codice parallelo

- L’algoritmo presenta implicitamente molte dipendenze, quindi non è semplice da parallelizzare.
- L’idea consiste nel parallelizzare il calcolo degli elementi della matrice delle distanze all’interno di una anti-diagonale della matrice.

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

L’ordine di calcolo delle anti-diagonali deve essere rispettato.
Celle con lo stesso numero possono essere calcolate contemporaneamente.

L’idea dietro il codice parallelo

- L’algoritmo presenta implicitamente molte dipendenze, quindi non è semplice da parallelizzare.
- L’idea consiste nel parallelizzare il calcolo degli elementi della matrice delle distanze all’interno di una anti-diagonale della matrice.

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

L’ordine di calcolo delle anti-diagonali deve essere rispettato.
Celle con lo stesso numero possono essere calcolate contemporaneamente.

Il codice parallelo

```
for(int i=1; i < m; i++){
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (int count = 1; count<=i;count++){
            int j_start = 0;
            int i_start = i+1;
            int costoperation = 0;
            if(j_start + count< n && i_start - count>=1){
                j_start+=count;
                i_start-=count;
                if(string1[i_start-1]==string2[j_start-1]){
                    costoperation=0;
                } else {
                    costoperation=1;
                }
                mat.at(i_start * n + j_start) = min(min(mat[(i_start-1) * n + j_start] + 1, mat[i_start * n + j_start - 1] + 1), mat[(i_start-1) * n + j_start - 1] + costoperation);
            }
        }

    }
}
```

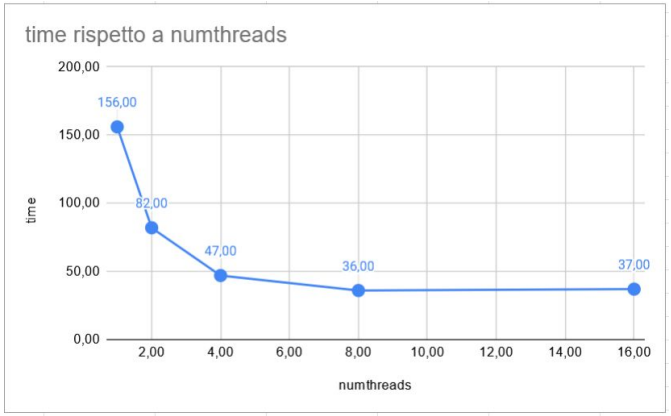
Visita matrice per anti-diagonali fino a completare la metà superiore della matrice delle distanze.

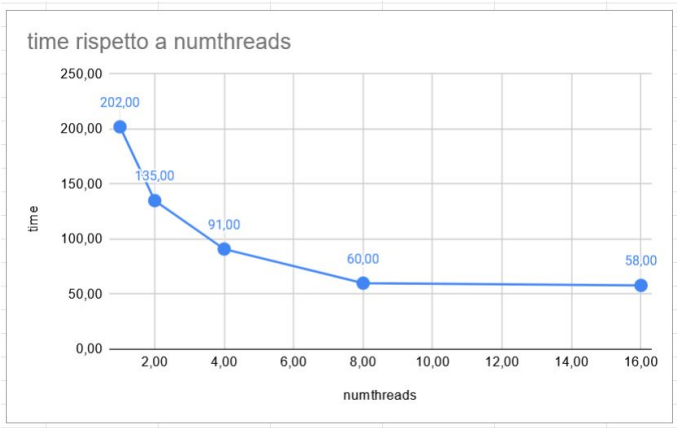
Il codice parallelo

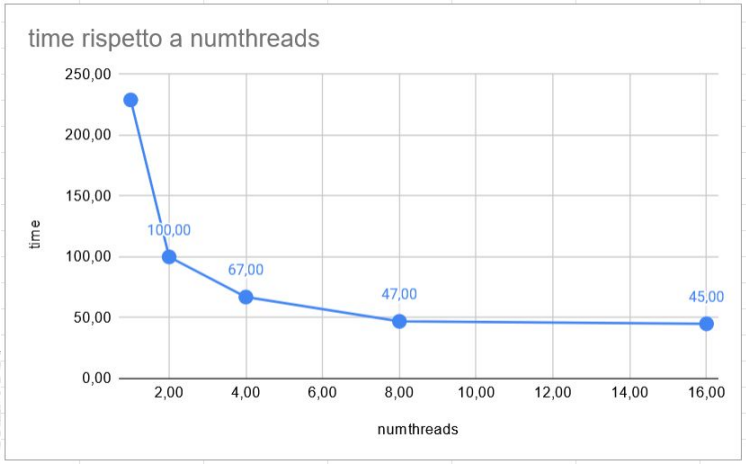
```
for (int j=2; j <= n; j++){
    #pragma omp parallel
    {
        #pragma omp for nowait
        for(int count=1; count < n;count++){
            int j_start = j-1;
            int i_start = m;
            int costoperation = 0;
            if(j_start + count< n && i_start - count>=1){
                j_start+=count;
                i_start-=count;
                if(string1[i_start-1]==string2[j_start-1]){
                    costoperation=0;
                } else {
                    costoperation=1;
                }
                mat.at(i_start * n + j_start) = min(min(mat[(i_start-1) * n + j_start] + 1, mat[i_start * n + j_start - 1] + 1), mat[(i_start-1) * n + j_start - 1] + costoperation);
            }
        }
    }
}
//printmatrix(mat,m,n);
return mat[m * n - 1];
```

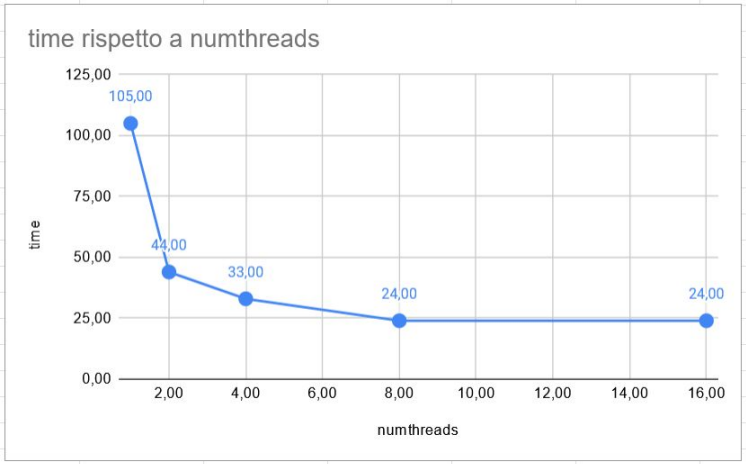
Visita matrice per anti-diagonali fino a completare la metà inferiore della matrice delle distanze.
Infine si ritorna l'ultimo elemento della matrice, che contiene la distanza desiderata.

Test effettuati

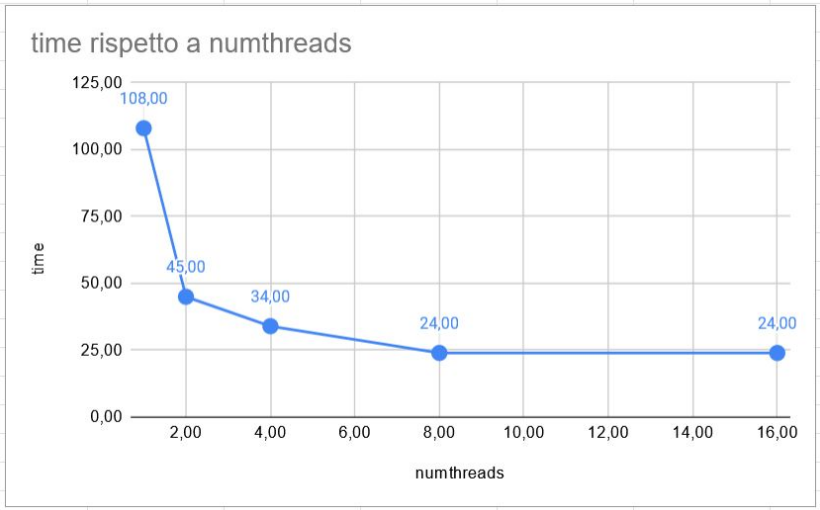


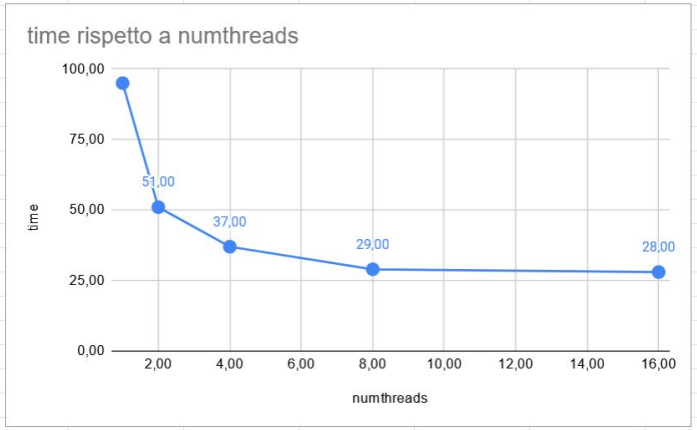






Test effettuati cont.





- I tempi sono stati misurati tramite differenze di omp_get_wtime() prima e dopo l'esecuzione dei vari metodi.

Considerazioni finali

- Il metodo parallelo riesce a raggiungere uno speedup rispetto a quello sequenziale di 4-5 volte con un numero di thread elevato.
- Aggiungere altri thread sopra una certa soglia non migliora ulteriormente le performance.
- Le direttive omp riescono quindi a velocizzare molto il confronto tra due stringhe.
- Gli esempi che sono stati considerati sono sufficientemente grandi per poter garantire una resa ancora più netta del programma parallelo rispetto a quello sequenziale.