Relazione Progetto Parallel Programming
N-Grammi

Ruci Sindi
7090797 sindi.ruci@stud.unifi.it

September 13, 2022

1 Introduzione

Un ***bi-gramma*** è l'insieme di due lettere consecutive presenti all'interno di un testo. Ad esempio se prendiamo la frase: "La casa di Maria si trova in montagna", i bi-grammi di questa frase sono: la - ac - ca - as e così via, fino ad arrivare alla fine della frase.
Se invece di voler trovare gruppi di due lettere, si desiderasse indagare quali sono i gruppi di tre lettere allora si andrebbe a calcolare i ***tri-grammi*** di un testo. Ovviamente si può calcolare gruppi di lettere di cardinalità n generica, in questa situazione si parla di ***n-grammi***.
La consegna di questo progetto chiedeva di calcolare i bi-grammi e i tri-grammi di un testo e di procedere al calcolo seguendo sia una implementazione sequenziale che una parallela.

1.1 Struttura del codice

Il codice presenta tre file con estensione .cpp e due header files. I tre file sono il main e le due implementazioni richieste. I file header permettono l'esecuzione delle due implementazioni da parte del main. Le due risoluzioni non presentano funzioni in comune perché si sono voluti mantenere i due procedimenti totalmente separati, anche se c'era la possibilità di poter avere porzioni di codice condivise.
Il testo da analizzare è stato ricavato dalla pagina Wikipedia che parla della storia dell'informatica. Il testo è stato ripulito da tutti i caratteri speciali come punteggiatura o lettere accentate. Essendo che ci interessa sapere la ricorrenza di un bi-gramma (tri-gramma), dopo la pulizia il testo è stato riportato interamente in lettere minuscole. La pulizia del testo è avvenuta tramite un breve script in Python in quanto a noi interessava la differenza di performance tra l'analisi sequenziale e quella parallelizzata del testo, non tanto le tempistiche di pulizia del codice. Infatti il testo ripulito era di dimensioni contenute, circa 50KB, in seguito ho copiato e incollato il testo varie volte in vari file, in modo da poter testare il codice C++ con file di grandezza differente.

Listing 1: Pulizia del codice con Python

```
def text_in_right_form(file):
file = open(file, 'r', encoding='utf-8')
text = transform_text(file)
file2 = open('testo.txt','w',encoding='utf-8')
file2.write(text)
file2.close()
file.close()


def transform_text(file):
text = file.read()
new_text = re.sub(r'[^a-zA-Z]','',text);
text = new_text.lower()
return text
```

2 Implementazione Sequenziale

L'implementazione sequenziale della consegna si divide in due parti principali:

- lettura del testo dal file
- analisi del testo letto

La lettura del file è stata semplicemente eseguita tramite una funzione .open() da cui è poi stata letto l'intero testo carattere per carattere e inserito all'inerno di una stringa lunga tanto quanto il testo
L'analisi del testo invece è stata fatta considerando il testo come un'unica stringa e utilizzando un finestra, grande due caratteri nel caso del bi-gramma e tre caratteri nel caso del tri-gramma, che scorreva a partire dall'inizio del file, un carattere alla volta, leggeva l'n-gramma risultante e aggiornava una mappa. La mappa è costituita da una strina e un intero, nel caso in cui l'n-gramma fosse già presente nella mappa, allora si aggiornava il conteggio sommando uno all'intero associato alla chiave; se invece non era presente si inseriva una nuova voce nella mappa con l'n-gramma e l'inero inizializzato a 1.

```
map<string, int> calculate_n_gram(int n_gram_dim, string text){
    map<string, int> dictionary;


    for (int i = 0; i < text.size() - n_gram_dim +1; i++)
    {
        string n_gram;

        for (int j = 0;  j < n_gram_dim; j++)
        {
            n_gram.push_back( c: text[ i + j]);
        }

        if (dictionary.count( x: n_gram))
        {
            dictionary[n_gram] = dictionary[n_gram] + 1;
        } else
        {
            dictionary.insert( x: pair<string, int> ( &: n_gram,  y: 1));

        }

    }

    return dictionary;
}
```

Figure 1: Codice sequenziale

3 Implementazione Parallela

L'implemantazione parallela segue la falsa riga della sequenziale. L'idea è dividere il file in un numero di parti equivalente alla quantità di thread disponibili nel PC, nel mio caso si arriva fino a un massimo di 8 thread in quanto la CPU presente nel computer è composta da 4 core, ognuna con due thread. Una volta suddiviso il file, si assegna a ogni thread il compito di analizzare una parte del file in maniera del tutto parallela rispetto al resto dell'esecizione. Ogni thread ha una propria mappa all'inerno della quale conserva i risultati dell'analisi della porzione di testo assegnata al thread. Per quanto riguarda la lettura del testo, questa avviene esattamente nello stesso modo sia nella versione sequenziale che in quella parallela.

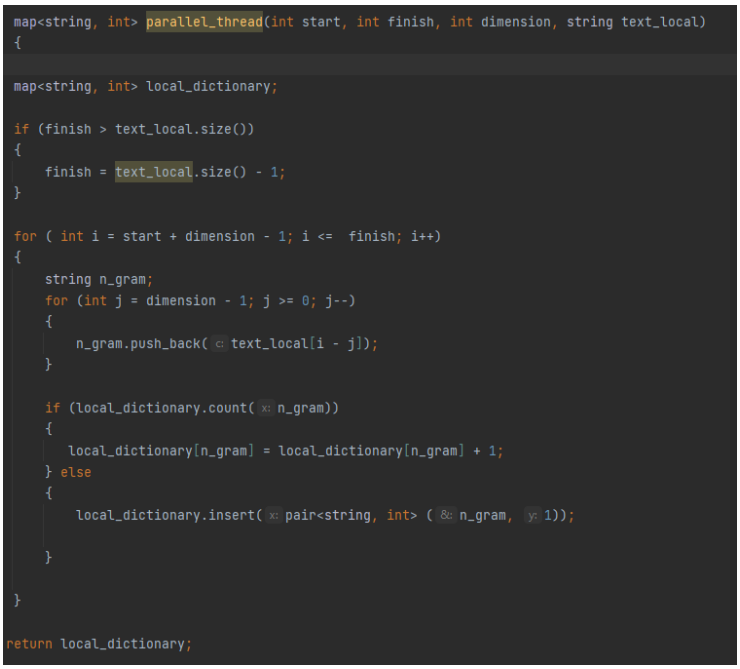

Figure 2: Codice thread

4 Confronto delle prestazioni

Dopo questa breve introduzione delle implementazioni passiamo ora al confronto prestazionale. Per prima possiamo vedere come le prestazioni cambiano mantenendo al massimo i thread nella versione parallela e cambiando i testi analizzati, mettendoli in ordine crescente di grandezza.
Le statistiche presentate sotto, riportano i dati per i tri-grammi.

Grandezza del file	Tempo esecuzione sequenziale	Tempo esecuzione parallela
50KB	163 (millisecondi)	36(millisecondi)
100KB	319(millisecondi)	78(millisecondi)
10MB	27(secondi)	5(secondi)
50MB	136(secondi)	24(secondi)
100MB	272(secondi)	48(secondi)

Possiamo notare come le prestazioni dell'implementazione parallela sono in ogni caso migliori rispetto a quelle dell'implementazione sequenziale, per i file più piccoli è stato necessario calcolare i tempi di esecuzione in millisecondi in quanto i secondi risultavano essere 0 per entrambe le implementazioni. Questo mostra come per file piccoli non sia necessario parallelizzare in quanto i tempi di esecuzione sono praticamente istantanei. Notevole è la differenza di tempistiche per quanto riguarda i file più grandi poiché notiamo come la versione parallela

riesca sempre a rimanere sotto il minuto, mentre la sequenziale è ben al di sopra.

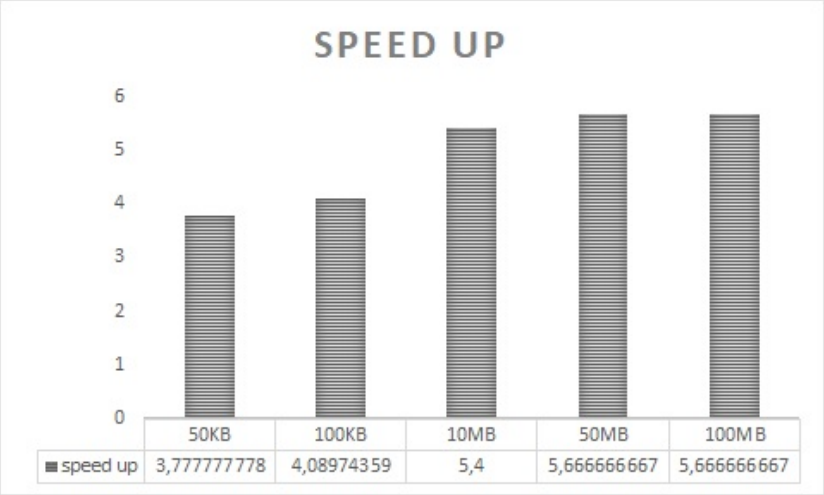

Figure 3: SpeedUp delle prestazioni

Il grafico mostra ancora meglio come i benefici della parallelizzazione siano importanti man mano che il file aumenta, fino ad arrivare a una tempistica di esecuzione fino a 6 volte maggiore. Se manteniamo il file di grandezza stabile e variamo i thread, otteniamo i seguenti risultati.

	Tempo di esecuzione
Sequenziale	21 secondi
1 Thread	21 secondi
2 Thread	10 secondi
3 Thread	7 secondi
4 Thread	5 secondi
5 Thread	5 secondi
6 Thread	3 secondi
7 Thread	2 secondi
8 Thread	2 secondi
10 Thread	3 secondi

Notiamo come aumentando il numero di thread migliorano i tempi di esecuzione, ma nel momento in cui si eccede i thread a noi disponibili, allora i tempi di esecuzione cominciano a peggiorare. Possiamo fare lo stesso confronto per quanto riguarda i bi-grammi.

Grandezza del file	Tempo esecuzione sequenziale	Tempo esecuzione parallela
50KB	58 (millisecondi)	21(millisecondi)
100KB	118(millisecondi)	34(millisecondi)
10MB	9(secondi)	2(secondi)
50MB	45(secondi)	9(secondi)
100MB	91(secondi)	19(secondi)

Anche in questo caso notiamo come i tempi di esecuzione della versione parallela siano decisamente migliori rispetto a quelli della versione sequenziale. Come per

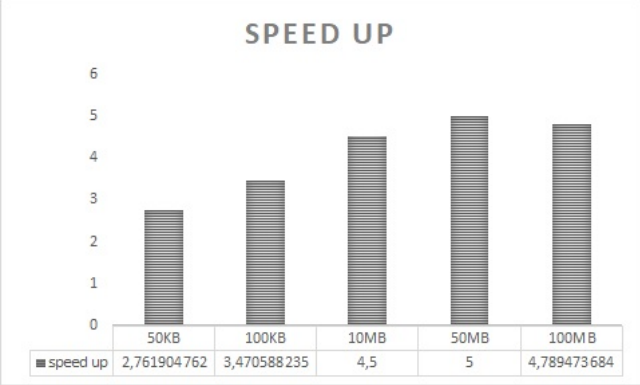

Figure 4: SpeedUp delle prestazioni dei bigrammi

i tri-grammi è giusto vedere le prestazioni nel caso dei bi-grammi variando il numero di thread messi a disposizione per il programma. Anche in questo caso si è utilizzato un file di dimensioni fissate, 10MB.

	Tempo di esecuzione
Sequenziale	9 secondi
1 Thread	9 secondi
2 Thread	5 secondi
3 Thread	3 secondi
4 Thread	3 secondi
5 Thread	2 secondi
6 Thread	2 secondi
7 Thread	2 secondi
8 Thread	1 secondi
10 Thread	2 secondi

Anche qui notiamo il progressivo miglioramento con l'aumentare dei thread messi a disposizione, ma appena si supera il numero di thread presenti, le prestazioni tendono a peggiorare.