

Bigrams frequency counts using C++ threads

Filippo Mameli
filippo.mameli@stud.unifi.it

Abstract

Un bigramma (o digramma) è un gruppo di due lettere o parole consecutive. L'analisi della frequenza dei bigrammi è comunemente usata nella crittoanalisi o nell'apprendimento automatico. In questo documento si presenta un programma scritto in C++ che conta le occorrenze di bigrammi in un testo. Si mostrano inoltre le differenze di prestazioni tra il programma nella versione parallela e quella sequenziale. I principali strumenti utilizzati sono la classe `std::Thread` e la libreria `Boost`.

Permessi di distribuzione

L'autore di questa relazione permette che questo documento possa essere distribuito a tutti gli studenti UNIFI dei corsi futuri.

1. Introduzione

Un bigramma è una sequenza di due lettere o parole addiacenti. I bigrammi sono utilizzati in vari ambiti. L'analisi statistica del testo con i bigrammi è usata ad esempio in crittoanalisi, in apprendimento automatico o nel riconoscimento vocale. Nella relazione si descrive un programma parallelo scritto in C++ che conta le frequenze dei bigrammi di lettere in un testo. Nella sezione 4 si mostrano inoltre le differenze prestazionali tra il programma parallelo e la sua controparte sequenziale, mostrando lo Speedup e i tempi di esecuzione tra differenti documenti di testo. Nel programma si utilizzano principalmente la classe della libreria standard `Thread` e la libreria `Boost`.

2. Algoritmo di base

Il programma deve parsare ogni singola parola e aggiornare il contatore dei bigrammi trovati. La struttura dati per contare le occorrenze è l'`Hashtable`. Questa è stata scelta perché l'operazione più usata è la ricerca del bigramma trovato e il conseguente incremento del contatore. In un `Hashtable` l'operazione di ricerca ha complessità di tempo pari a $O(1)$ nel caso medio e per questa qualità, la struttura è adatta per l'aggiornamento delle occorrenze. Nell'algoritmo successivo vi è descritto in pseudocodice i passaggi che sono effettuati.

Data: File di testo

Result: Frequenza dei bigrammi contati
lettura del file di testo;

```
while fine del file non raggiunto do
    for per ogni parola in file do
        for per ogni bigramma nella parola do
            Hashtable[bigramma]++;
        end
    end
end
```

Algorithm 1: Algoritmo di base

3. Parallelizzazione

Dall'algoritmo di base si può ricavare facilmente una versione parallelizzata del programma. Infatti è possibile dividere il file in più parti e assegnare ogni porzione ad un thread che potrà eseguire il conteggio dei bigrammi solo sul singolo frammento. Per far questo si deve suddividere il file e poi creare tanti thread quante sono le parti del documento. Il problema principale è la struttura dati condivisa dai thread per aggiornare le occorrenze dei bigrammi. Le strutture dati nella libreria standard di C++ non garantiscono il compor-

tamento delle operazioni di modifica dei dati da parte di più thread. Infatti se due o più thread devono aggiornare il contatore di un singolo bigramma, c'è un problema di dipendenza nell'operazione di modifica. Per questo motivo è stata creata una classe che ingloba `std::unordered_map` e rende thread safe le operazioni necessarie.

3.1. Divisione dell'esecuzione

Nel Codice 1 si può vedere come viene effettuata la creazione dei thread. Ad ogni ciclo si aggiunge un thread al vettore `threads`. Ai thread viene associata una funzione `countFunction` che prende come argomenti `edge` e `bottom` che indicano la posizione del file su cui deve operare il thread. Alla fine del ciclo si richiama la funzione di `join` per ogni thread appartenente al vettore `threads` per terminare la loro esecuzione.

Codice 1. Creazione dei thread

```
44  edge = words.size()*(i+1)/cores;
    threads.push_back(std::thread(
        countFunction, bottom, edge));
46  bottom = edge;
    for (auto& th : threads) th.join();
```

La funzione `countFunction` controlla ogni singola parola contenuta nel vettore `words` cercando le sottostringhe di lunghezza 2 (Codice 2 linea 58). Se il bigramma non è presente nell'hashMap (cioè `count(bigram)==0`), si inserisce. Altrimenti si richiama la funzione `add` che incrementa il contatore di una unità.

Codice 2. Funzione del thread

```
56  string bigram;
    for (size_t i = bottom; words.size() >= (
        edge-1) && i < edge; i++) {
        for (size_t j = 0; j < (words[i].length
            ()-1); j++) {
58      bigram = words[i].substr(j,2);
        if (hashMap.count(bigram)==0)
60      hashMap.insert(bigram);
        else
62      hashMap.add(bigram);
        }
64  }
```

3.2. Thread safe unordered map

`threadsafe_unordered_map` è una classe wrapper di `std::unordered_map`. In questa nuova classe la funzione di `insert` è thread safe e `add` incrementa il valore del contatore dei bigrammi in modo atomico. Come mostrato nel Codice 3, quando viene richiamata `insert` si utilizza `lock_guard` per bloccare la struttura dati e garantire il corretto funzionamento della funzione `countFunction`.

Codice 3. Funzione insert

```
22  void insert(const keyType& key){
    std::lock_guard<std::mutex> guard{mtx};
24  hashtable[key]++;
    }
```

L'atomicità della funzione `add` (Codice 4) è data invece dall'uso di `atomic uint`. Utilizzando questa funzionalità di C++ non ci sono data race e si sincronizza gli accessi all'hashmap tra i singoli thread.

Codice 4. atomic e Funzione add

```
private:
14  unordered_map<keyType, atomic<uint> >
    hashtable;
    mutex mtx;
16
public:
18  void add(const keyType& key){
    hashtable.at(key)++;
20  }
```

3.3. Lettura del file e utilizzo del programma da terminale

L'ultimo aspetto da trattare è la lettura del file e l'utilizzo del programma. `ReadFileUtility` è la classe che legge il contenuto di un file (dato in input alla funzione `readInputFile`) e restituisce un vettore di parole. In `readInputFile` si utilizzano le funzionalità della libreria Boost per la tokenizzazione delle parole nel testo. Nel Codice 5 si può vedere che, passando il testo del file come stringa e `char_separator` a `tokenizer`, si possa creare facilmente il vettore `words` che verrà poi utilizzato per il conteggio dei bigrammi.

Codice 5. Read file

```
boost::container::vector<string>
    readInputFile(string path){
20   ifstream input(path);
    stringstream textStream;
22
    while(input >> textStream.rdbuf());
24   string text = textStream.str();

26   char_separator<char> sep(".,;:)(*',_?!-#$
    %&[]{}\\t\\n\\\\\\r\\'\\\"\\a\\f\\b\\v ");
    tokenizer<char_separator<char>> tokens(
        text, sep);
28   BOOST_FOREACH(string s, tokens){
        words.push_back(s);
30   }
    return words;
32 }
```

3.3.1 Utilizzo del programma

Da terminale si può utilizzare il programma specificando il percorso del file di testo e il numero thread che saranno creati. Si utilizzerà il numero di core della cpu che la macchina ha a disposizione nel caso in cui non si indichi il numero di thread. Nel Codice 6 vi è un esempio.

Codice 6. Read file

```
g++ -std=c++11 ../bigramPar.cpp -o ../bin/
    bigrammiParallelo.out -lboost_system -
    lpthread
2 ../bin/bigrammiParallelo.out ../testFiles/
    file_prova_perm.txt 4
```

4. Analisi delle performance

Per l'analisi delle performance sono stati utilizzati file di varie dimensioni. I tempi di esecuzione del programma parallelo sono stati comparati con quelli della versione sequenziale. Quest'ultima non fa uso dei thread e utilizza `std::unordered_map` nella versione standard.

In Figura 1 vi sono riportati gli Speedup rispetto alla scelta del file. I dati sono relativi al rapporto tra il tempo impiegato dal programma sequenziale e quello del programma parallelo che utilizza 4 thread (uno per ogni core della CPU). Si può vedere che per file molto piccoli (1kb) la ver-

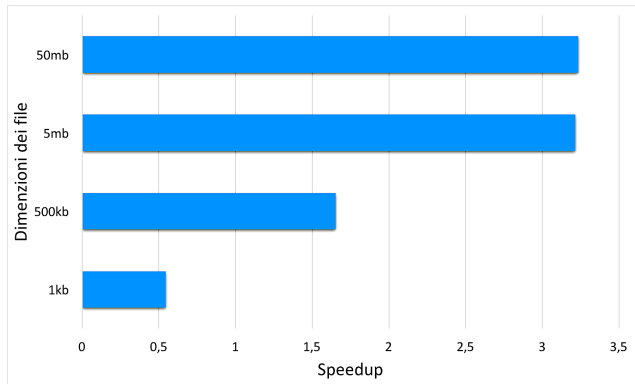


Figura 1. Speedup rispetto alle dimensioni del file

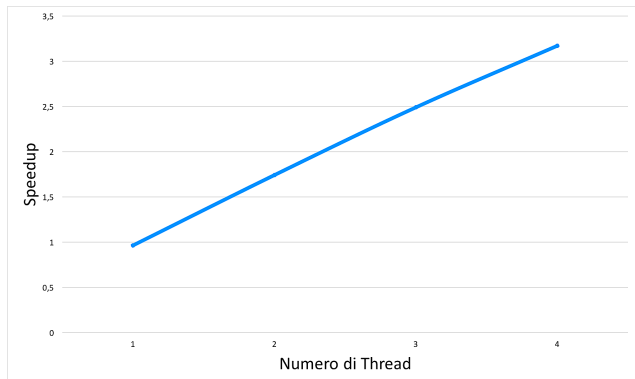


Figura 2. Speedup rispetto al numero di thread

	T. Parallelo	T. Sequenziale
1kb	0,000187	0,000102
500kb	0,002496	0,004127
5mb	0,30084	0,967499
50mb	1	3,47725

Figura 3. Speedup rispetto al numero di thread

sione parallela appesantisce l'esecuzione, ma già per il file di 500kb il programma ha uno Speedup pari a 1.7. Per file intorno ai 5mb o superiori l'overhead causato dalla creazione dei thread diventa irrilevante e lo Speedup si porta intorno a 3.2 e si stabilizza.

La Figura 2 mostra invece lo Speedup rispetto ad un unico file di dimensione 5mb cambiando il numero di thread.

In Figura 3 vi sono riportati i risultati assoluti in secondi.