PC-2016/17 Progetto finale
Implementazione concorrente di un algoritmo per il calcolo dei bigrammi

Tommaso Ceccarini
6242250
tommaso.ceccarini1@stud.unifi.it

Federico Schipani
6185896
federico.schipani@stud.unifi.it

Abstract

I bigrammi sono una coppia di parole o lettere adiacenti all'interno di un testo. Questa relazione descrive l'algoritmo e le relative scelte implementative di un programma scritto in C++ che calcola la frequenza dei bigrammi di parole che occorrono all'interno di un insieme di file di testo utilizzando un approccio multithreading.

1. Introduzione

I bigrammi sono una coppia di parole o lettere adiacenti all'interno di un testo. Questa relazione descrive l'algoritmo e le relative scelte implementative di un programma scritto in C++ che calcola la frequenza dei bigrammi di parole che occorrono all'interno di un insieme di file di testo utilizzando un approccio multithreading. L'algoritmo segue uno schema di tipo produttore consumatore in cui:

1. I produttori si occupano di leggere i file di testo da una cartella

2. I consumatori hanno il compito di calcolare i bigrammi dei file di testo processati dai produttori

Nello specifico, ciascun produttore, dato il percorso del file da elaborare, legge il contenuto del file e produce un token per ogni parola nel testo. I bigrammi saranno dati dalle coppie di token che vengono determinati in questa fase. Il dato prodotto quindi consiste in un insieme di token che viene posto in una coda. I consumatori, d'altra parte, restano in attesa che ci siano dati da consumare e quando ciò accade, ciascuno di essi è incaricato di contare il numero di occorrenze di ogni bigramma presente nel testo che ha acquisito. Dopo questa fase, ogni consumatore aggiorna il contenuto della tabella che memorizza il numero di occorrenze globali dei bigrammi.

2. Implementazione

In questa sezione vengono spiegate le scelte che hanno portato all'implementazione del programma sviluppato in C++ per calcolare il numero di occorrenze dei bigrammi in più file di testo. Nel corso delle diverse sottosezioni vengono anche descritte le diverse strutture dati che sono state utilizzate.

2.1. Scegliere i file da leggere

Questa operazione è stata implementata utilizzando una coda concorrente i cui elementi sono i percorsi dei file da elaborare. A questo scopo è stata utilizzata la libreria `boost::filesystem::path` per rappresentare i percorsi dei file e la classe `ConcurrentQueue`, reperibile dalla repository GitHub concurrentqueue, per implementare la coda concorrente `fileQueue` in cui i percorsi sono memorizzati. I path sono inseriti nella coda iterando sull'oggetto `targetDir`, di tipo `boost::filesystem::path`, che rappresenta il percorso della cartella da cui i file vengono prelevati, quest'ultimi sono inseriti all'interno di `fileQueue` tramite il metodo `enqueue`.

Codice 1: Lettura della cartella e selezione dei file di testo

```
1    fs::path targetDir("/home/cecca/ClionProjects/CPP-Bigrams
         ↪ /File/Italiano");
2    fs::directory_iterator it(targetDir), eod;
3    std::vector<std::thread> threadVector;
4    BOOST_FOREACH(fs::path const &p, std::make_pair(it, eod))
         ↪ {
5                    if (fs::is_regular_file(p)) {
6                        fileQueue.enqueue(p);
7                    }
8                }
```

2.2. Schedulare i task: Utilizzo di due threadpool

Come è stato accennato nell'introduzione, il programma sviluppato realizza uno schema produttore consumatore per contare le occorrenze dei bigrammi. I thread che rappresentano i produttori e i consumatori vengono gestiti mediante

due distinte thread pool realizzate mediante la libreria CTPL, reperibile sulla repository CTPL. Una delle threadpool si occupa di schedulare i task da assegnare ai produttori mentre l'altra fa altrettanto con i consumatori. Le due thread pool sono oggetti contenuti nei metodo che avviano rispettivamente la produzione e la consumazione. Inoltre, ai metodo `startProducer()` e al metodo `startConsumer()` viene passato il numero di thread con cui inizializzare le due piscine.

Codice 2: Operazione di avvio della consumazione

```
void Consumer::consume(int threadNumber, int maxSize) {
    ctpl::thread_pool thread_pool(threadNumber);
    bool resized = false;
    for (int i = 0; i < expectedFiles; i++) {
        thread_pool.push([this, maxSize, &thread_pool, &
            ↪ resized](int id) {
            this->calcBigrams(id, maxSize, &thread_pool, &
                ↪ resized);
        });
    }
    thread_pool.stop(true);
    bigrams.writeHtmlFile("/home/cecca/ClionProjects/CPP-
        ↪ Bigrams/File/bigrammi.html", 1000);
}
```

Codice 3: Operazione di avvio della produzione

```
void Producer::produce(int threadNumber) {
    ctpl::thread_pool thread_pool(threadNumber);
    unsigned long queueSize = fileQueue.size_approx();
    for (int i = 0; i < queueSize; i++) {
        thread_pool.push(
                [this](int id, moodycamel::ConcurrentQueue<
                    ↪ boost::filesystem::path> *&fileQueue
                    ↪ ,
                       moodycamel::ConcurrentQueue<std::
                           ↪ vector<std::string>> *&q) {
                    this->elaborateText(id, fileQueue, q);
                }, &fileQueue, &q);
    }


    thread_pool.stop(true);
    *notified = true;
    *done = true;
    cv->notify_all();
}
```

2.3. Lettura dei file

Come si può notare dal Codice 4 per realizzare questa fase viene istanziato un oggetto `producer` di tipo `Producer` a cui vengono passate la coda concorrente `fileQueue` e una coda concorrente `q` sempre realizzata mediante la libreria moodycamel utilizzando la classe concurrentqueue i cui elementi sono vettori di stringhe. Successivamente viene chiamata la funzione membro `producer.startProducer()` a cui viene passato come parametro il numero di thread di tipo produttore. Questa funzione di occupa innanzitutto di inizializzare la threadpool dei produttori utilizzando il parametro passato in input per poi effettuare tante chiamate alla funzione membro `thread_pool.push()` quanti sono i file da elaborare. Tale funzione membro riceve in input:

1. Una lambda expression che a sua volta riceve un parametro numerico e le due code concorrenti `fileQueue` e `q` e si occupa di richiamare il metodo `elaborateText()` che si occupa di fare la lettura vera e propria del file
2. La coda `fileQueue` da cui vengono estratti i percorsi
3. La coda `q` in cui sono prodotti i token

Come appena accennato, la funzione membro `elaborateText()` si occupa di realizzare la lettura vera e propria di un file: viene estratto il percorso del file dalla coda `fileQueue` e utilizzando un oggetto di tipo `mapped_file` della libreria `boost::iostreams` ne viene rappresentato il contenuto. i token sono realizzati definendo il tipo `tokenizer` i cui elementi sono dei caratteri separatori. I caratteri separatori sono rappresentati mediante l'oggetto `sep` della classe `boost::char_separator<char>` mentre `tokenizer` è definito a partire dalla classe `boost::tokenizer<boost::char_separator<char>`. I token veri e proprio sono realizzati istanziando l'oggetto `tok` di tipo `tokenizer` con i parametri `readFile, sep`, in cui `readFile` è una stringa che contiene il testo del file che è stato letto. Per realizzare quindi il dato `producerUnit` che consiste in un vettore di stringhe, è sufficiente iterare su `tok` per memorizzare le `string` che rappresentano i token. Infine, il dato prodotto `producerUnit` viene posto nella coda `q`. In Codice 5 è riportata l'implementazione del metodo `elaborateText()`.

Codice 4: creazione oggetti producer e consumer

```
    Producer producer(q, fileQueue, &m, &cv, &done, &notified
        ↪ );
    Consumer consumer(q, fileQueue, (int) fileQueue.
        ↪ size_approx(), &m, &cv, &done, &notified);
    std::thread threadProducer = producer.startProducer(48);
    std::thread threadConsumer = consumer.startConsumer(1,36)
        ↪ ;
    threadProducer.join();
    threadConsumer.join();
    return 0;
```

Codice 5: Elaborazione del testo della produzione

```
void Producer::elaborateText(int id, moodycamel::
    ↪ ConcurrentQueue<boost::filesystem::path> *&fileQueue
    ↪ ,
                             moodycamel::ConcurrentQueue<std
                                 ↪ ::vector<std::string>>
                                 ↪ *&q) {
```

```
    boost::filesystem::path path;

    fileQueue->try_dequeue(path);
    int k = 0;
    boost::iostreams::mapped_file file(path);
    string readFile = file.data();
    std::string word;
    std::vector<std::string> producerUnit;
    producerUnit.reserve(readFile.size());
    typedef boost::tokenizer<boost::char_separator<char>>
        ↪ tokenizer;
    boost::char_separator<char> sep{" ,.;-<>_()[]{}"};
    tokenizer tok{readFile, sep};
    for (const auto &t : tok) {
        producerUnit.push_back(t);
    }
    q->enqueue(producerUnit);
    cv->notify_one();
    *notified = true;

}
```

2.4. Calcolo dei bigrammi

Per realizzare questo compito viene istanziato un oggetto di tipo `Consumer` che riceve `q`, `fileQueue` e il numero di file su cui calcolare i bigrammi. La gestione dei thread e dei task da eseguire è analoga all'operazione di lettura dei file. Con la differenza che, in questo caso, i thread sono dei consumatori e il task da compiere è rappresentato dal metodo `calcBigrams()`. In questo caso, quindi, l'operazione di `consume()` consiste in un ciclo `for` che esegue tante iterazioni quanti sono i file nella cartella sorgente in cui viene richiamata la funzione membro `thread_pool.push()` che riceve una lambda expression che si occupa di chiamare la funzione membro `calcBigrams()` la quale rappresenta il task da eseguire. Come prima operazione, `calcBigrams()` definisce una `unordered_map m` che ha come parametri la struct `Key`, il tipo primitivo `int` e un `KeyHasher`. Key viene definita all'interno di Consumer.h e serve a rappresentare i bigrammi all'interno dell'`unoderded_map` mentre `KeyHasher` rappresenta la funzione hash che viene applicata su `Key`. Dopo questa fase, il consumatore resta in attesa che ci siano dati da consumare e quando questa condizione si verifica, il dato prelevato `text`, che consiste in un vettore di stringhe, viene scandito per istanziare l'oggetto `bigram` di tipo `Key` con le due stringhe `text[i]` e `text[i+1]`. Successivamente viene controllato se il bigramma appena creato è già presente in `m` e, in tal caso, il valore che rappresenta il numero di occorrenze di tale bigramma nel file che viene letto viene incrementato, altrimenti, inizializzato a 1. Dopo che `text` è stato elaborato, `m` conterrà le occorrenze dei bigrammi del file che è stato preso in considerazione. A questo punto è necessario aggiornare il conteggio globale delle occorrenze per i bigrammi nella hashmap concorrente. A tale fine è stata definita la libreria `ConcurrentUnorderedIntMap.hpp` che, utilizzando un `mutex`, garantisce l'accesso in mutua esclusione al valore intero su cui effettuare l'incremento. In particolare, viene definito un lock sull'oggetto `mutex` prima di effettuare le operazioni di incremento e inizializzazione. Questa modalità di accesso è definita nelle funzioni membro `insertAndIncrement()` e `insertAndSet()`. Quindi `calcBigrams()` aggiorna la hashmap globale iterando su `m`, ossia, per ogni bigramma considerato viene controllato se è già presente nell'hashmap globale e, in tal caso, viene incrementato il valore della relativa occorrenza tramite una `insertAndIncrement()`. Altrimenti, il valore delle occorrenze globali viene inizializzato a 1 mediante una `insertAndSet()`. Nel Codice 6 è riportata l'implementazione del metodo `calcBigrams()`.

Codice 6: Calcolo dei bigrammi

```
void Consumer::calcBigrams(int id, int maxSize, ctpl::
    ↪ thread_pool *threadPool, bool *resized) {
    std::unique_lock<std::mutex> lock(*m);
    std::vector<std::string> text;
    unordered_map<Key, int, KeyHasher> m;
    if (!(*resized) && *done) {
        threadPool->resize(maxSize);
        *resized = 1;
    }
    while (!(*done || *notified)) {
        cv->wait(lock);

    }
    q.try_dequeue(text);
    *notified = false;

    for(int i = 0;i < text.size()-1;i++){
        Key bigram(text[i], text[i + 1]);
        if(m.find(bigram) != m.end()){
            m[bigram]++;
        } else{
            m[bigram] = 1;
        }
    }

    for(auto elem : m){
        int counter = elem.second;
        if (bigrams.find(elem.first) != bigrams.end()) {
            bigrams.insertAndIncrement(elem.first, counter);
        }else{
            bigrams.insertAndSet(elem.first, 1);
        }

    }

}
```

3. Analisi delle performance

Per effettuare l'analisi delle performance sono stati usati dei file prelevati da http://www.gutenberg.org/. Tutti i test sono stati effettuati sulla macchina del MICC messa a disposizione, che ha due processori Intel Xeon E5-2620 V3. I risultati delle varie esecuzioni sono riportati in Tabella 1, 2

e 3. In Tabella 1 e in Tabella 2 sono stati presi in considerazione diversi casi ottenuti dalla combinazione del numero di thread consumatore e thread produttore, nel caso in cui essi valgono 1, 12, 24, 36 e 48. Nelle righe sono riportati i valori crescenti del numero di thread consumatore, nelle colonne invece sono riportati il numero dei thread produttore. Nelle celle sono riportati i tempi di esecuzione riscontrati nei casi che sono stati presi in considerazione. Questi risultati sono ottenuti facendo una media di 10 esecuzioni effettuate per quei particolari valori. Nella Tabella 1 sono riportati i tempi dell'esecuzione con 78 file, mentre nella Tabella 3 sono stati presi in considerazione 234 file.

Tabella 1: Test con 78 elementi

#C \ #P	1	12	24	36	48
1	16,6 s	17,4s	17,4s	17,1s	16,9s
12	17,8s	19,1s	18,5s	18,9s	18,5s
24	17,6s	19,7s	19,9s	20,2s	19,6s
36	18,7s	20,3s	20,5s	20,4s	19,7s
48	19,7s	21,3s	21,4s	20,1s	20,9s

Tabella 2: Test con 234 elementi

#C \ #P	1	12	24	36	48
1	45,1s	48,8s	51,1s	50,0s	51,1s
12	47,4s	51,8s	51,4s	53,1s	55,5s
24	50,2s	54,7s	55,9s	55,6s	55,4s
36	52,7s	55,7s	56,0s	56,0s	56,9s
48	54,7s	54,9s	58,5s	55,5s	56,7s

Nel corso dei test effettuati è stato riscontrato che il processo di produzione veniva compiuto più velocemente rispetto al processo di consumazione. A questo scopo è stata implementata la possibilità di specificare due parametri diversi per il consumatore: uno che determina il numero iniziale di thread consumatore, l'altro che specifica il numero massimo di thread che viene raggiunto quando il processo di produzione è giunto al termine. In Tabella 3 sono riportati i casi in cui il numero di produttori sia 36 o 48 e i consumatori partono da 1 e arrivano ad un numero massimo di 12, 24, 36 o 48, mentre il numero di file è sempre 234. Come si può notare confrontando la Tabella 3 con la Tabella 2 la possibilità di aumentare il numero di thread che effettuano il processo di consumazione migliora i tempi di esecuzione.

Tabella 3: Test con 234 elementi, consumatori variabili

#C \ Max #P	12	24	36	48
48	50,8s	49,9s	49,6s	50,4s
36	49,8s	50,5s	50,1s	49,7s