Parallel Bigrams/Trigrams

Francesco Bizzarri

Gennaio 2023

Relazione dell'elaborato mid-term per il corso di Parallel Programming for Machine Learning

1 Introduzione

In questo progetto vengono implementate due versioni di codice per calcolare l'istogramma degli n-grams di parole e di caratteri presenti in un testo. Vedremo le differenze di performance in termini di speed-up tra la versione parallela e quella sequenziale.

2 Il corpus

Come testo su cui calcolare gli istogrammi viene utilizzato più volte il romanzo *La fattoria degli Animali* di George Orwell. In particolare analizzeremo i risultati su un testo composto dallo stesso romanzo replicato 100 e 500 volte. Creando un corpus rispettivamente di 2.362.000 e 11.810.000 parole.

3 N-grams di parole

3.1 Versione sequenziale

Il loop principale del programma sequenziale per estrarre n-grams di parole è il seguente:

```
unordered_map<string, int> histogram;
string word;
vector<string> previous_words(n - 1);
while (file >> word) {
        if (num_words >= n - 1) {
            string ngram = "";
            for (int i = 0; i < n - 1; i++) {
                ngram += previous_words[i] + " ";
            }
```

```
            ngram += word;
            histogram[ngram]++;
        }
        for (int i = 0; i < n - 2; i++) {
            previous_words[i] = previous_words[i + 1];
        }
        previous_words[n - 2] = word;
        num_words++;
    }
    print_histogram(histogram);
```

Notiamo che il programma compone gli n-grams tenendo traccia delle parole precedenti a quella della corrente iterazione attraverso il vettore *previous_words*.

3.2 Versione parallela

A differenza della versione precedente le parole del file testuale prima di essere processate vengono inserite in una struttura dati (in questo caso un vettore). Così che un team di thread possa dividersi equamente il lavoro da fare su ogni porzione del vettore.

```
    vector<string> words;
    string word;
    while (file >> word) {
        words.push_back(word);
    }
    file.close();
```

Dopodiché inizia il vero e proprio loop che crea l'istogramma utilizzando delle direttive di OpenMP:

```
    int size = words.size();

#pragma omp parallel num_threads(NUM_THREADS)
    {

        unordered_map<string, int> thread_histogram;

#pragma omp for nowait
        for (int i = n - 1; i < size; i++) {
            string ngram = "";
            for (int j = i - n + 1; j < i + 1; j++) {
                ngram += words[j] + " ";
            }
            ngram.pop_back(); // to remove the extra blank space
            thread_histogram[ngram]++;
```

```
        }

#pragma omp critical
        for (auto [ngram, count]: thread_histogram) {
            global_histogram[ngram] += count;
        }

    }   // end of parallel region
    print_histogram(global_histogram);
```

Per parallelizzare il codice è utilizzato il pattern della riduzione. Ogni thread ha il proprio istogramma privato su cui calcolare gli n-grams. Per creare l'istogramma globale, alla fine, viene utilizzata una sezione critica in modo tale da evitare race conditions. Ho sperimentato anche usando una direttiva atomic all'interno del ciclo for, invece della sezione parallela. Ma i risultati non sono migliori.

4 N-grams di caratteri

Entrambe le versioni del codice utilizzano questa funzione per estrarre n-grams di caratteri da una singola parola:

```
vector<string> extract_ngrams_from_word(const string& word) {
    int n = this->getNgramLength();
    vector<string> ngrams;
    if (word.size() >= n) {
        for (int i = 0; i < word.size() - (n - 1); i++) {
            ngrams.push_back(word.substr(i, n));
        }
    }
    return ngrams;
}
```

4.1 Versione sequenziale

```
    string word;
    vector<string> ngrams;
    while (file >> word) {
        ngrams = extract_ngrams_from_word(word);
        for (auto ngram : ngrams) {
            histogram[ngram]++;
        }
    }
    print_character_histogram(histogram);
```

4.2 Versione parallela

```
    vector<string> words;
    string word;

    while (file >> word) {
        words.push_back(word);
    }
    file.close();

    int size = words.size();

#pragma omp parallel num_threads(NUM_THREADS)
    {

        unordered_map<string, int> thread_histogram;
        vector<string> current_ngrams;

#pragma omp for nowait
        for (auto word: words) {
            current_ngrams = extract_ngrams_from_word(word);
            for (auto ngram : current_ngrams) {
                thread_histogram[ngram]++;
            }
        }

#pragma omp critical
        for (auto [ngram, count]: thread_histogram) {
            global_histogram[ngram] += count;
        }

    }   // end of parallel region
    print_character_histogram(global_histogram);
```

Anche in questo caso il ragionamento è lo stesso. Le parole estratte dal file testuale vengono prima salvate in un vettore. Così che poi il team di thread, creato con la sezione parallela, possa dividersi il lavoro che verrà ricombinato insieme solo alla fine.

5 Analisi dei risultati

L'analisi dei risultati si basa sul tempo di esecuzione (wall clock) delle intere funzioni. Che comprendono la parte di lettura del file testuale di input e la stampa in console di una porzione di istogramma. Comparando solo i cicli di cruda computazione ovviamente i valori di speed-up sarebbero superiori.

5.1 Trigrams

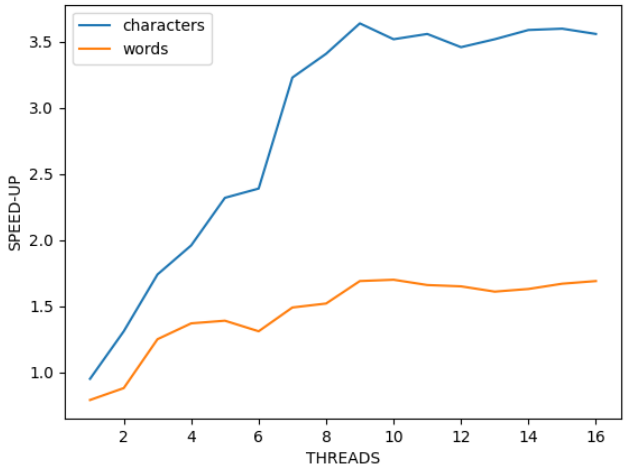

Figure 1: Corpus di 2 mln di parole

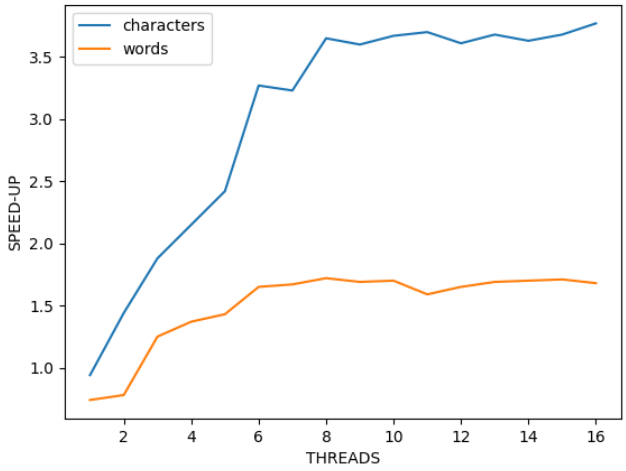

Figure 2: Corpus di 11 mln di parole

5.2 Bigrams

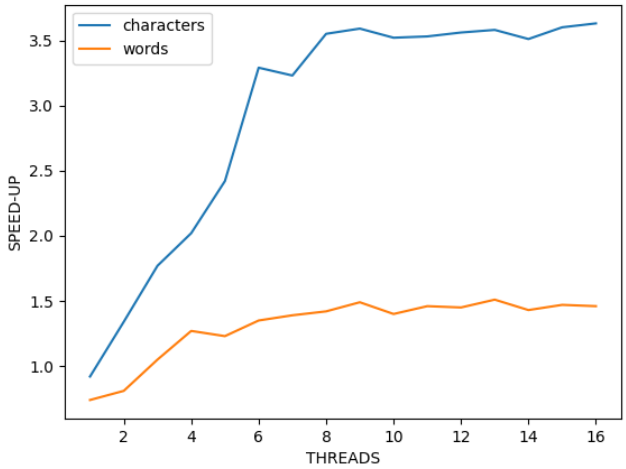

Figure 3: Corpus di 2 mln di parole

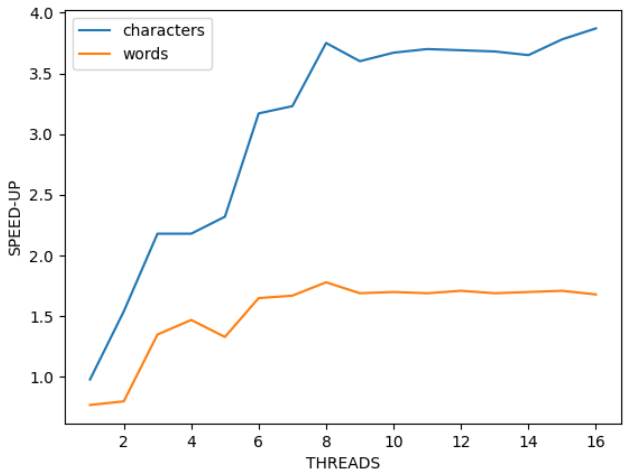

Figure 4: Corpus di 11 mln di parole

6 Conclusioni

Notiamo come la parallelizzazione del codice abbassi i tempi di esecuzione fino a 4 volte nel caso di ngram di caratteri. È evidente come al crescere del corpus non ci siano significativi miglioramenti in termini di speed-up. Estrarre ngram di caratteri è computazionalmente più costoso, per questo i vantaggi della parallelizzazione sono più evidenti. Inoltre è bene ricordare come questi risultati sono ottenuti dall'esecuzione dell'intera funzione di estrazione degli ngram. Che comprende la lettura del file testuale dal disco. In questa operazione il collo di bottiglia non è la CPU bensì la velocità del disco stesso.