UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing
Bigram and Trigram Computing

Academic Year 2019-2020

Paolo Le Piane

Manuel Natale Sapia

Abstract: *Obiettivo del progetto è l'implementazione di un generatore di bigrammi e trigrammi di lettere estrapolati da un testo selezionato. Sono state elaborate tre versioni del framework: una versione sequenziale in Java e due versioni parallele in Java e OpenMP. In fine è stato effettuato un confronto tra queste tre varianti relativo alle tempistiche di esecuzione e allo speed up.*

1. Introduzione

Un n-gramma è una sotto sequenza di n elementi di una data sequenza.

Il framework proposto elabora e acquisisce n-grammi di lettere da un testo scelto, in particolare n-grammi di lunghezza due e tre chiamati rispettivamente bigrammi e trigrammi.

Nella prima parte del lavoro è stato implementato il framework in tre versioni: una versione sequenziale in Java e due versioni parallele in Java e C++ utilizzando OpenMP [1], andando a calcolare i bigrammi e trigrammi del testo e la frequenza di ogni singolo bigramma e trigramma.

La seconda parte dell'elaborato è stata dedicata alle misurazioni delle performance e al confronto di quest'ultime tra le tre versioni. In particolare sono stati valutati il tempo di esecuzione e lo speed up in funzione della dimensione del file di testo e del numero di thread per le versioni parallele.

Le varie versioni del framework sono state testate su un PC con queste specifiche:

- Processore Intel Core i7-6700HQ da 2.60GHz con turbo boost fino a 3.5GHz
- GPU NVIDIA GeForce GTX 970M con 1280 core e 3GB di memoria dedicata
- 16GB di memoria RAM

2. Implementazione

2.1. Versione Sequenziale: Java

La versione sequenziale è stata implementata in linguaggio Java. Il file di testo scelto per l'analisi è letto tramite la funzione *myReadFile()*. In questa funzione è stata usata la classe Java BufferedReader per salvare il file di testo e leggerlo linea per linea. Tutte le linee del file in ingresso sono memorizzate in un'unica stringa rilasciata in output, in cui i caratteri vengono tutti convertiti in minuscolo.

```
public static String myReadFile(String filename) {

try {

BufferedReader reader = new BufferedReader(new
FileReader(filename));

StringBuilder output = new StringBuilder();

String line;

while ((line = reader.readLine()) != null) {

output.append(line);

output.append(" "); //used to separate each word from the others

}

reader.close();

return output.toString().toLowerCase();

...
```

1. *Estratto funzione myReadFile()*

La funzione che svolge il ruolo principale è *generateNgrams()* che prende in ingresso l'n dell' n-gramma e il testo sottoforma di stringa da analizzare. La stringa viene percorsa interamente e, in base all'n scelto, vengono creati gli n-grammi, salvati nell'array ngrams. Dal conteggio vengono esclusi gli n-grammi con spazi e numeri all'interno.

```
public static List<String> generateNgrams(int n, String str) {
      String letters = "abcdefghijklmnopqrstuvwxyz";
      boolean skip;
      List<String> ngrams = new ArrayList<>();
      for (int i = 0; i < str.length() - n + 1; i++) {
          skip = false;
          for (int k = 0; k < n ; k++) {
              if (letters.indexOf(str.charAt(i+k)) == -1) {
                  skip = true;
                  break;
              }
          }
          if (skip) continue;
          ngrams.add(str.substring(i, i + n));
      }
      return ngrams;
  }
```

2. Funzione generateNgrams()

Per calcolare la frequenza di ogni n-gramma è stata creata la funzione *frequency()* che, tramite l'utilizzo di una linked hash map, conta le occorrenze degli n-grammi unici, dividendo il valore trovato per il totale degli n-grammi. Da questa funzione sono stati ottenuti due file di testo contenenti le frequenze di bigrammi e trigrammi del testo utilizzato per le analisi.

Per ottenere il file con tutti gli n-grammi ricavati dal file di testo in ingresso, è stata implementata la funzione *myWriteFile()* che utilizza la classe BufferedWriter.

In fine nel metodo *main()*, per calcolare le tempistiche dell'elaborazione di bigrammi e trigrammi sono state utilizzate le classi Instant e Duration con le relative funzioni *now()* per calcolare un istante preciso, *between()* e *toMillis()* per calcolare il tempo tra due istanti in millisecondi.

2.2. Versione Parallela: Java

Una versione parallela di questo progetto è stata implementata in Java, in particolare utilizzando l'interfaccia Callable.

Callable rappresenta un task asincrono che può essere eseguito da thread diversi. Essa contiene il metodo *call()* che esegue il task asincrono e ha la particolarità di poter generare un'eccezione in caso di fallimento del task durante l'esecuzione e di poter ritornare in output un oggetto Future. Quest'ultimo rappresenta il risultato di una computazione asincrona. Un oggetto Callable viene eseguito tramite Java ExecutorService.

Il file di testo scelto è letto, come nella versione sequenziale dalla funzione *myReadFile()* che riporta in output il testo sottoforma di unica stringa.

È stata creata una classe CallMain che implementa Callable e il suo costruttore prende in ingresso: *n* dimensione dell'n-gramma, *str* stringa contenente il testo, *start* e *end* che indicano la dimensione dei blocchi di testo da passare ai thread.

Per ottenere gli intervalli per partizionare in blocchi la stringa con il testo, è stata implementata la funzione *create_chunks()* che prende in ingresso il numero di thread utilizzati e la stringa contenente il testo. Vengono creati tanti blocchi quanti sono i thread e se la fine di un blocco si posiziona dove è presente un carattere lettera, questa viene fatta avanzare fino a ricoprire una posizione con un carattere spazio. In output viene rilasciato una lista di interi con gli intervalli di testo.

```
public static List<Integer> create_chunks(int numb_th, String str) {
        List<Integer> result = new ArrayList<>();
        int pos;
        result.add(0);
        int chunk = str.length() / numb_th;
        for (int k = 1; k < numb_th; k++ ) {
            pos = chunk * k;
            while (str.charAt(pos) != ' ') {
                pos += 1;
            }
            result.add(pos);
        }
        result.add(str.length());
        return result;
    }
```

3. Funzione create_chunks()

Il metodo *call()* esegue il task asincrono e in particolare genera gli n-grammi relativi all'intervallo del testo delimitato da start e end. In caso di fallimento dell'esecuzione genera un'eccezione Exception. In output si ha per ogni blocco di testo l'array di n-grammi, il quale è reso accessibile tramite un oggetto Future.

```
public List<String> call() throws Exception{
        String letters = "abcdefghijklmnopqrstuvwxyz";
        boolean skip;
        List<String> ngrams = new ArrayList<>();
        assert start <= end;
        for (int i = start; i < end - n + 1; i++) {
            skip = false;
            for (int k = 0; k < n ; k++) {
                if (letters.indexOf(str.charAt(i+k)) == -1) {
                    skip = true;
                    break;
                }
            }
            if (skip) continue;
            ngrams.add(str.substring(i, i + n));
        }
        return ngrams;
    }
```

4. Funzione Call()

Come nella versione parallela sono presenti le funzioni *frequency()* per calcolare la frequenza degli n-grammi e *myWriteFile()* per ottenere un file con tutti gli n-grammi del file di testo in ingresso.

Nel metodo *main()* viene creato un executor di tipo ThreadPoolExectuor che genera e gestisce un threadPool formato da un determinato numero di thread che eseguono il task asincrono. Viene poi effettuato da parte del ThreadPoolExectuor il submit del task Callable, il quale viene eseguito dal threadPool. L'output sarà una lista di oggetti Future da cui verranno prelevati i blocchi di stringhe contenenti il testo.

Come nella versione sequenziale il tempo di esecuzione è stato ottenuto tramite le funzioni *now()*, *between()* e *toMillis()* delle classi Instant e Duration.

```
public static void main(String[] Args) throws InterruptedException {

        ...

        ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(4);

        List<Future<List<String>>> resultList = new ArrayList<>();

        Instant start_t = Instant.now();

        for (int i = 0; i < number_th; i++) {

            CallMain prova = new CallMain(2, book, ck.get(i), ck.get(i+1));

            Future<List<String>> result = executor.submit(prova);

            resultList.add(result);

        }

        for (Future<List<String>> future : resultList) {

            try

            {

                count = count + future.get().size();

            }

            ...
```

5. Estratto main()

2.3. Versione Parallela: C++ con OpenMP

La seconda versione parallela è stata implementata utilizzando il linguaggio C++ e l'api OpenMP per la gestione della parallelizzazione. OpenMP offre un modo molto facile di utilizzare i thread, utilizzando un modello fork&join senza rendere necessario che la creazione e la gestione dei task debba essere effettuata manualmente.

In questa versione sono presenti un metodo per la lettura da file di testo e uno per la scrittura su file di testo. In particolare, la funzione che effettua la lettura da file è *myReadFile()* nel quale il file viene letto per righe e per ogni riga vengono sostituiti con spazi i caratteri non ammessi mediante l'uso di una espressione regolare. Successivamente, utilizzando la classe *stringstream*, la riga viene spezzata in varie parole e ognuna di queste viene trasformata in minuscolo e inserita in un vettore di stringhe,

```
regex letter("[^[:alpha:]]");
//apertura file
while (getline(inputFile, line)) {
	line = regex_replace(line, letter, " ");
	stringstream line2(line)
	while (getline(line2, word, ' ')) {
		//rendi word minuscolo e inseriscilo nel vettore di stringhe
	}
}
```

6. Estratto dal metodo myReadFile()

Il metodo *myWriteFile()* invece utilizza un iteratore per leggere i dati presenti nel vettore di stringhe e scriverli su un nuovo file il cui nome viene passato come parametro.

La funzione responsabile della creazione dei thread e nella quale vengono effettivamente utilizzati i costrutti di OpenMP è la funzione *generateNgramsParallel()* che riceve la dimensione degli n-grammi che dovrà andare a creare, la lista delle parole presenti nel file di testo (un vettore di stringhe) e il puntatore ad un vettore di vettori di stringhe in cui verranno memorizzati i vari n-grammi.

Quest'ultimo vettore è composto da un numero di vettori pari al numero di thread per far sì che nella memorizzazione degli n-grammi ogni thread lavori sulla propria porzione di spazio.

Nella prima regione parallela di questa funzione, la direttiva *parallel* serve a definire la parte di codice da parallelizzare, la clausola *private* in cui sono specificate le variabili *thread_index* e *word* specifica che quelle variabili sono private per ogni thread, cioè, che ogni thread lavorerà sulla sua porzione di memoria relativa a quelle variabili. Infine, la clausola *num_threads* serve a specificare il numero di thread che OpenMP dovrà creare.

All'interno della zona parallela viene memorizzato il numero del thread che verrà usato nella funzione *compute()* responsabile dell'effettiva generazione degli n-grammi. Infine, l'ultima regione parallela serve a permettere la parallelizzazione di un ciclo for e al suo interno viene richiamata la funzione *compute()* alla quale viene passato come parametro anche l'indice del thread.

```
#pragma omp parallel private(thread_index,word) num_threads(num_threads)
{
	thread_index = omp_get_thread_num();
	#pragma omp for
	for (int j = 0; j <strList.size(); j++) {
		word = strList[j];
		compute(thread_index, word, n, ngrams);
	}
}
```

7. Estratto dal metodo generateNgramsParallel()

Il metodo responsabile della creazione degli n-grammi da parte di ogni thread è *compute()* che riceve come parametri l'indice del thread, la parola da processare, la dimensione (*n*) degli n-grammi che dovrà andare a creare e il puntatore al vettore di n-grammi. La sua funzione è abbastanza semplice e lavora in modo simile alle altre versioni ma inserisce gli n-grammi nel vettore associato a quel determinato thread mediante l'uso del suo indice. Se m è la dimensione della parola esso genera

$m - n + 1$ n-grammi.

```
if(!word.empty())
	limit = word.length() - n + 1;
	for(int i = 0; i < limit; i++){
	    ngrams[thread_index].push_back(word.substr(i,n));
	}
```

8. Estratto dal metodo compute()

Infine è presente anche un metodo *frequency()* usato per calcolare la frequenza di ogni n-gramma all'interno del testo, similmente a quello che è stato fatto nelle altre versioni.

3. Dataset

Il file utilizzato è stato prelevato dalla libreria Project Gutenberg contenente più di 60.000 eBooks accessibili gratuitamente [2] ed è il testo del romanzo "Pride and Prejudice" [3], romanzo della scrittrice inglese Jane Austin pubblicato nel 1813.

Il file di testo è stato manipolato per ottenere dimensioni diverse utili per un'analisi più precisa e approfondita, in particolare: 75 Kb, 150 Kb, 250 Kb, 500 Kb, 1 Mb, 2 Mb, 4 Mb, 8 Mb, 16 Mb, 32 Mb, 64 Mb.

4. Esperimenti e Risultati

Per ogni versione del framework sono state effettuate delle valutazioni relative alle tempistiche di esecuzione e allo speed up. Quest'ultimo rappresenta il rapporto tra il tempo di esecuzione sequenziale e il tempo di esecuzione parallelo. Sono stati svolti degli esperimenti sia per la ricerca di bigrammi che di trigrammi. Nei grafici utilizzati per l'analisi dei risultati sono stati messi in relazione il tempo (in millisecondi) con la dimensione del file di testo e lo speed up sempre con la dimensione del file di testo.

Le dimensioni del testo scelte per l'analisi sono state di 75 Kb, 150 Kb, 250 Kb, 500 Kb, 1 Mb, 2 Mb, 4 Mb, 8 Mb, 16 Mb, 32 Mb, 64 Mb.

Per rendere più accurata la valutazione, ogni fase sperimentale è stata eseguita tre volte per poi calcolare la media dei tempi rilevati. Tutti i grafici sono stati generati su Excel.

4.1. Valutazione prestazioni in Java

Il primo esperimento riguarda la valutazione dello speed up nella versione parallela in Java. Nei grafici relativi sono state confrontate le prestazioni variando il numero di thread da 4 a 8. Come si può notare, al variare della dimensione del file di testo, lo speed up della parallela con otto thread risulta sempre sopra l'1 e quasi sempre maggiore rispetto alla versione a 4 thread sia nella generazione dei bigrammi che nella generazione dei trigrammi.

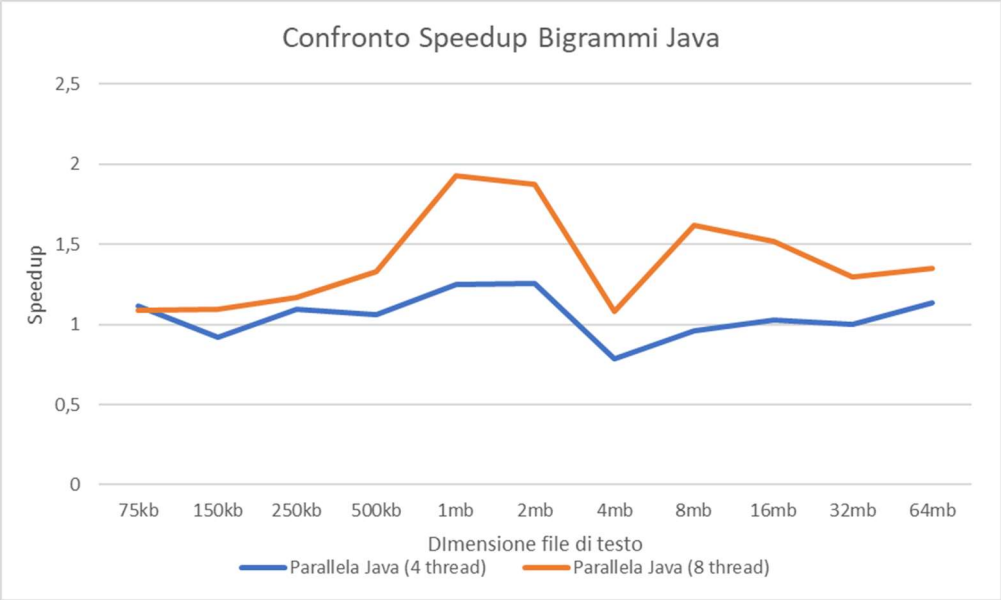

Figura 1 Confronto Speed up bigrammi Java

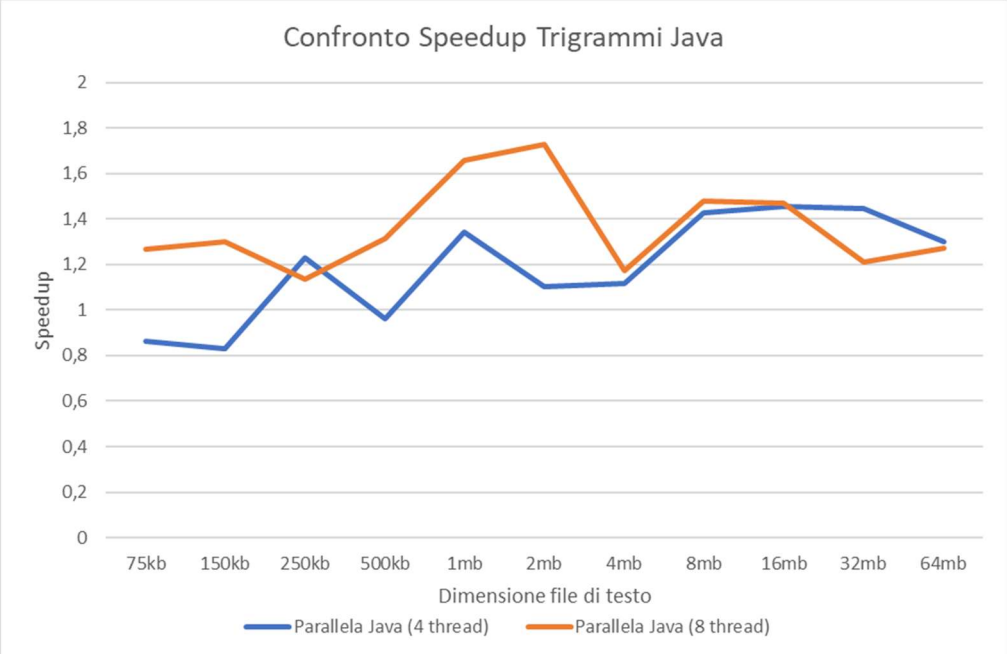

Figura 2 Confronto Speed up trigrammi Java

4.2. Valutazioni prestazioni in C++ con OpenMP

In questo secondo esperimento è stato valutato lo speed up nella versione parallela in C++ con OpenMP. Nei grafici relativi sono state confrontate le prestazioni variando il numero di thread da 4 a 8. Come si può notare, lo speed up assume dei valori molto alti con dimensioni di testo ridotte (75 Kb, 150 Kb) per poi scendere fino al valore 3 verso i 64 Mb. Questo si verifica in entrambe le varianti a 4 e a 8 thread sia nel calcolo dei bigrammi che dei trigrammi.

Nel grafico dei trigrammi si osserva un picco massimo a 150 Kb con 4 e con 8 thread.

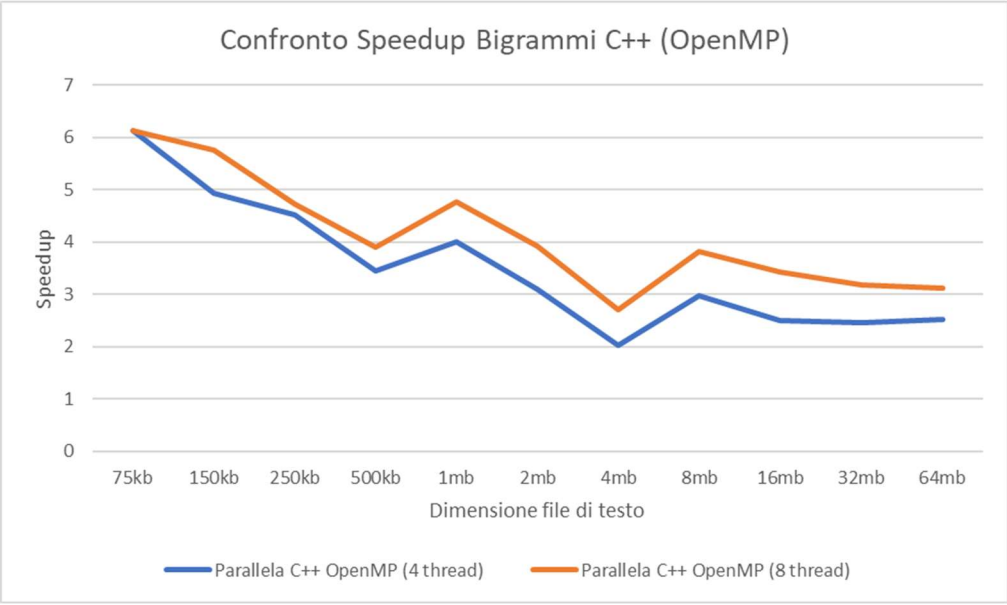

Figura 3 Confronto Speed up bigrammi c++ (OpenMP)

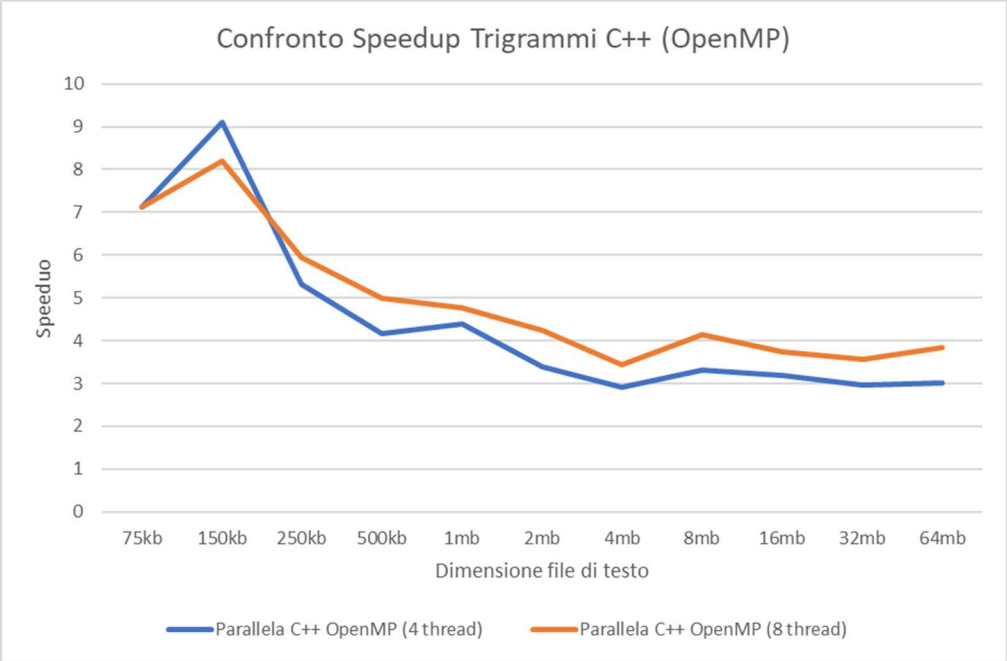

Figura 4 Confronto Speed up trigrammi C++ (OpenMP)

4.3. Valutazione tempi di esecuzione Java

Un altro elemento di misura delle performance della versione parallela in Java è il tempo di esecuzione per la generazione di bigrammi e trigrammi. Nei grafici a seguire sono stati confrontate le tempistiche della versione sequenziale, della versione parallela con 4 thread e della versione parallela a 8 thread.

Nel grafico della generazione dei bigrammi si può notare fino ai 500 Kb un andamento identico delle tre varianti, dai 4 Mb in poi troviamo un distacco in positivo per la versione a 8 thread mentre la versione a 4 thread assumerà delle tempistiche più performanti della sequenziale dalla dimensione dei 32 Mb in poi. A 64 Mb la differenza tra la versione sequenziale e quella parallela a 8 thread è di quasi 1.5 secondi.

Nel grafico della generazione dei trigrammi si nota un andamento simile al grafico precedente ma verso i 32 Mb i tempi di esecuzione della versione parallela a 4 thread migliorano rispetto alla versione a 8 per poi eguagliarli a 64 Mb.

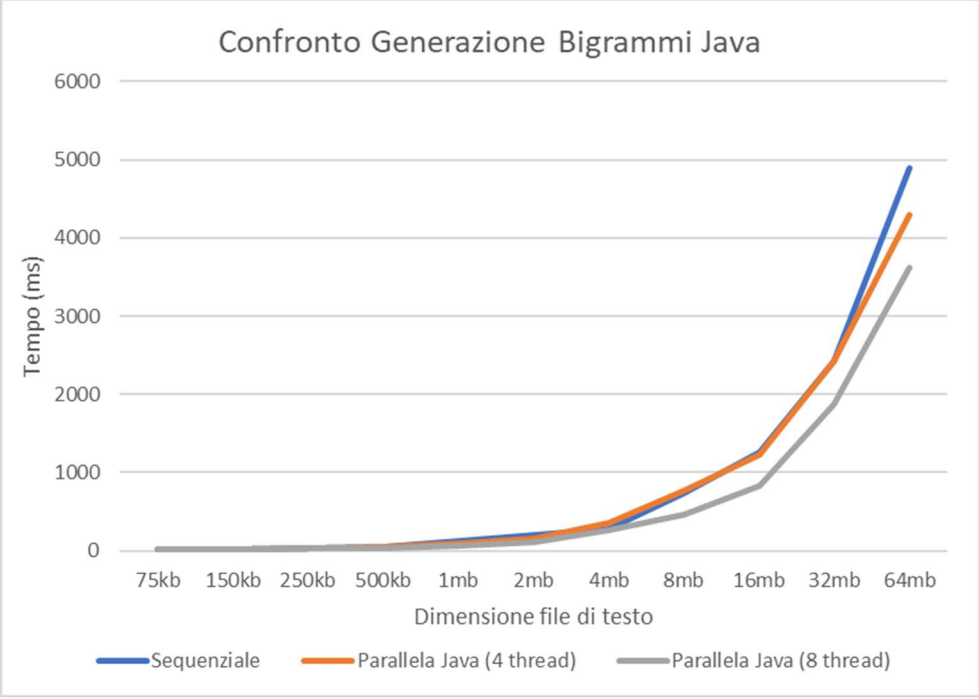

Figura 5 Confronto generazione bigrammi Java

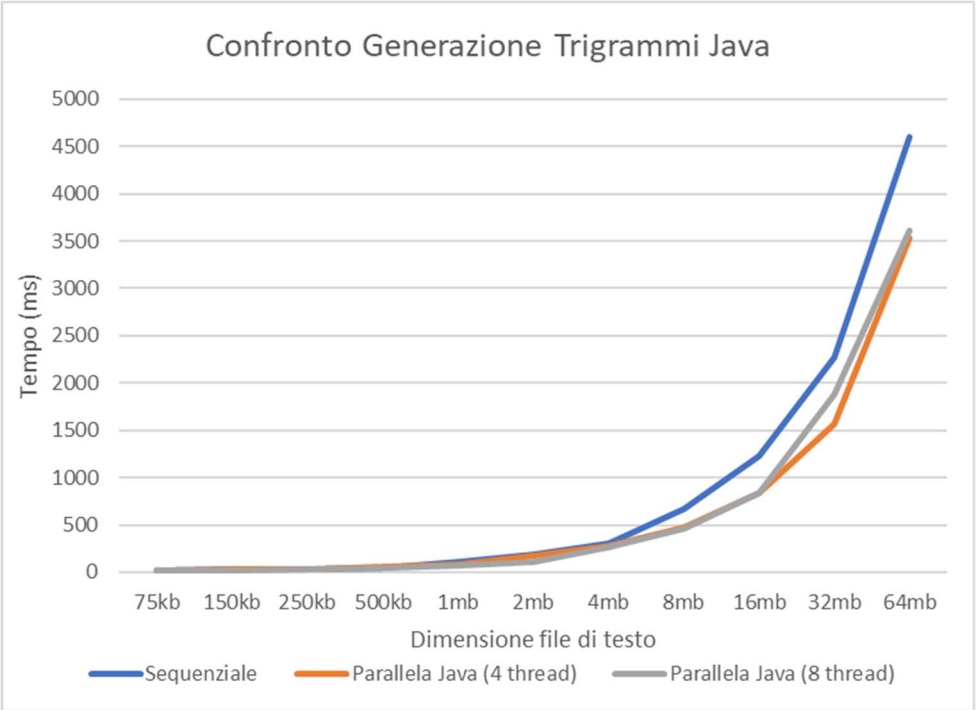

Figura 6 Confronto generazione trigrammi Java

4.4 Valutazione tempi di esecuzione C++ con OpenMP

Il tempo di esecuzione per la generazione di bigrammi e trigrammi è stato analizzato anche per la versione parallela in C++ con OpenMP. Nei grafici a seguire sono stati confrontate le tempistiche della versione sequenziale (in Java), della versione parallela con 4 thread e della versione parallela a 8 thread.

In entrambi i grafici troviamo un miglioramento delle performance delle versioni parallelle rispetto alla sequenziale verso i 500 Kb. Dai 4 Mb in poi il distacco tra le parallelle e la sequenziale diventa netto. Le due varianti parallele assumono un andamento simile in entrambi i grafici arrivando ad avere a 64 Mb una differenza di tempo di esecuzione rispetto alla sequenziale di circa 3 secondi.

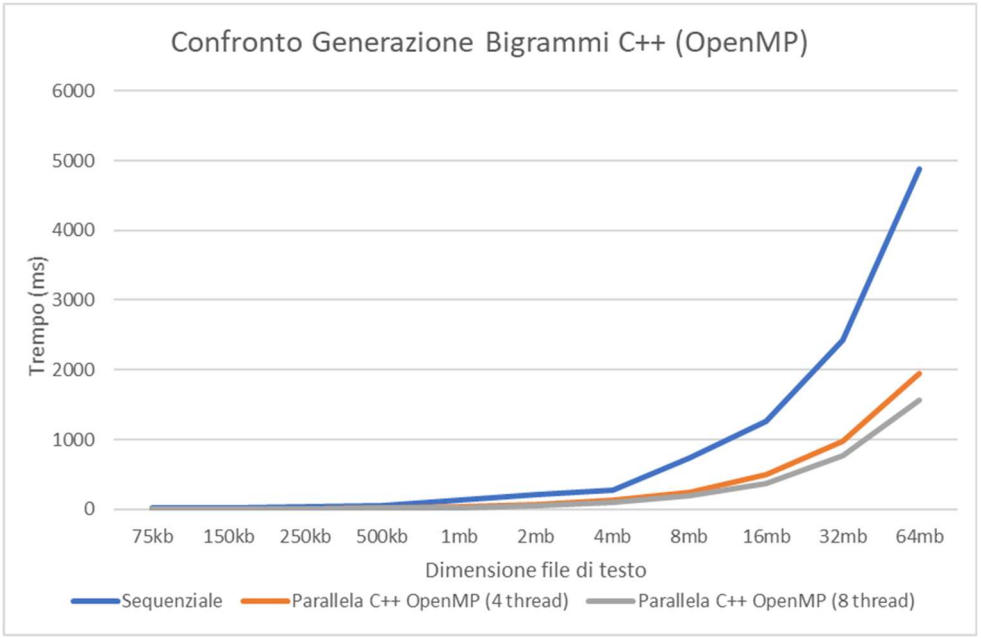

Figura 7 Confronto generazione bigrammi C++ (OpenMp)

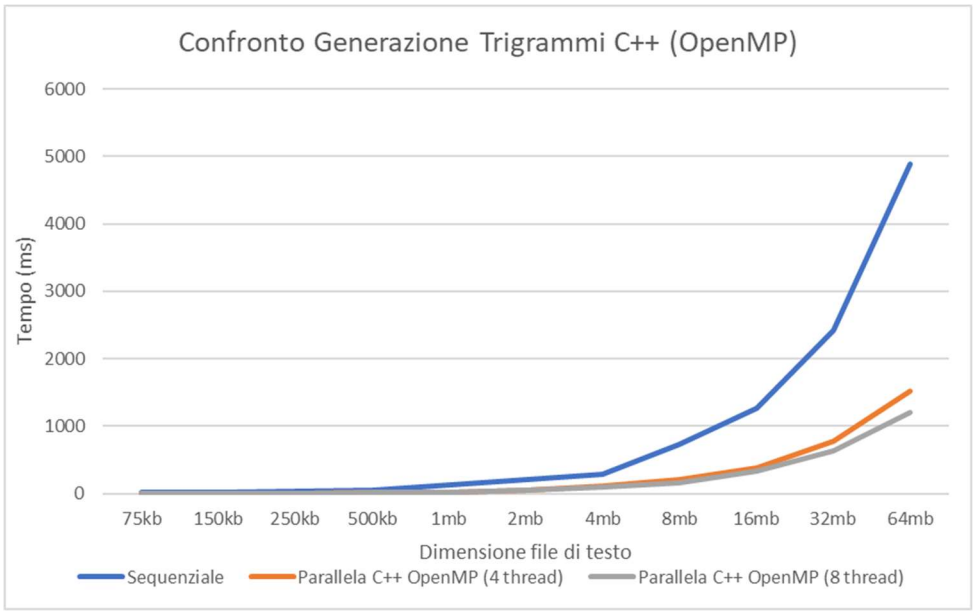

Figura 8 Confronto generazione trigrammi C++ (OpenMp)

5. Conclusioni

In questo progetto è stato implementato un framework per la generazione di bigrammi e trigrammi da un file di testo. È stata sviluppata una versione sequenziale in Java e due versioni parallele in Java e in OpenMP. Gli esperimenti e i relativi risultati hanno mostrato un miglioramento delle performance e delle tempistiche nelle versioni multithreading rispetto alla sequenziale, confermando le aspettative.

Bibliografia

[1] https://www.openmp.org/

[2] Progetto Gutenberg: https://www.gutenberg.org

[3]Romanzo Pride and Prejudice : https://en.wikipedia.org/wiki/Pride_and_Prejudice