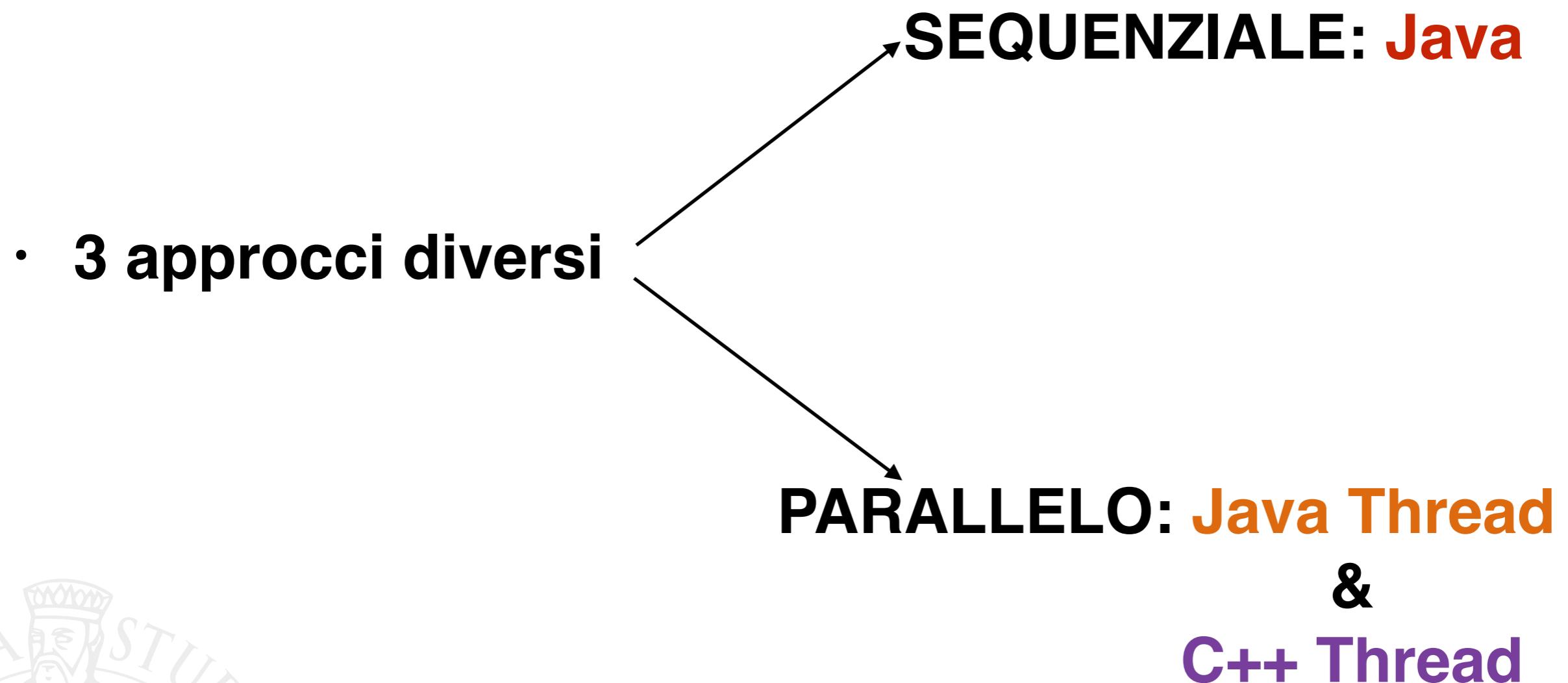




BigramTrigram Generator

Introduzione

- **Obiettivo:** Calcolare le occorrenze dei bigrammi e trigrammi di lettere in un testo



Bigrammi & Trigrammi

- Un ***n-gramma*** è una sottosequenza di n elementi di una data sequenza
- Gli **elementi** possono essere:
 - **Sillabe**
 - **Lettere**
 - **Parole**
- Ci siamo occupati di ***n-grammi*** di **lettere**, in particolare:
 - Una sequenza di due lettere si dice ***Bigramma*** o ***Digramma***
 - Una sequenza di tre lettere si dice ***Trigramma***

Descrizione del Dataset

- Estratto dal database ***Progetto Gutenberg*** che è una biblioteca digitale di eBook di pubblico dominio
- Testo scelto: “*La scienza in cucina e l'arte di mangiar bene*” di **Pellegrino Artusi**
- Esperimenti eseguiti con dimensioni variabili del file

Name	Size
artusi_50kb.txt	50 KB
artusi_100kb.txt	100 KB
artusi_200kb.txt	200 KB
artusi_500kb.txt	500 KB
artusi_1mb.txt	1 MB
artusi_2mb.txt	2 MB
artusi_4mb.txt	4 MB
artusi_8mb.txt	8 MB
artusi_16mb.txt	16 MB
artusi_32mb.txt	32 MB



Versione Sequenziale in Java

- Crea una classe **“LinesExtractor”** che prende in ingresso il nome del file **txt** ed estrae una lista di tutte le righe del testo

```
public List<String> extractLines() {  
    List<String> finalLines = new ArrayList<>();  
    List<String> lines = null;  
    String regex = "[^A-Za-z0-9']";  
  
    try {  
        FileSystem fs = FileSystems.getDefault();  
        lines = Files.readAllLines(fs.getPath("", filename));  
    } catch (IOException e) {  
        System.out.println("Failed to read " + filename);  
    }  
    lines.forEach(line -> {  
        if (!line.isBlank() && !line.isEmpty()) {  
            line = line.toLowerCase().replaceAll(regex, "");  
            finalLines.add(line);  
        }  
    });  
    return finalLines;  
}
```

- Si estrae la **lista di tutte le parole** tramite un'espressione regolare (**\s+**)

```
LinesExtractor extractor = new LinesExtractor(filename);  
List<String> lines = extractor.extractLines();  
  
List<String> words = new ArrayList<>();  
lines.forEach(line -> {  
    words.addAll(Arrays.asList(line.split("\\s+")));  
});  
  
words.removeAll(Collections.singleton(""));
```

Versione Sequenziale in Java

- Vengono calcolati i **bigrammi** e i **trigrammi** per ognuna delle parole estratte
- Il **tempo di esecuzione** viene misurato con la funzione ***currentTimeMillis()*** della libreria standard *System*

```
long start = System.currentTimeMillis();

int n = 2; // 2: bigrammi, 3: trigrammi

List<String> ngrams = new ArrayList<>();
for (String word: words) {
    if (word.length() >= n) {
        for (int i = 0; i < word.length() - (n - 1); i++) {
            ngrams.add(word.substring(i, i + n));
        }
    }
}

long end = System.currentTimeMillis();
```

Versione Parallelia con Java Thread

- Usata la stessa classe “***LinesExtractor***” della versione sequenziale
- Utilizzato un ***AtomicInteger*** per il conteggio dei bigrammi e trigrammi trovati per evitare inconsistenze
- La classe ***NGramThread*** prende in ingresso:
 - *globalCounter*: riferimento alla variabile di tipo *AtomicInteger*, rappresenta il contatore globale del numero di *n-grammi* trovati
 - *words*: l'intera lista di parole
 - *n*: intero che specifica se vengono trattati bigrammi (*n*=2) oppure trigrammi (*n*=3)
 - *start* e *stop*: due interi che indicano gli indici di partenza e arrivo del range di parole su cui ciascun thread lavora

```
public class NGramThread extends Thread {  
  
    private int n, start, stop;  
    private List<String> words;  
    private AtomicInteger globalCounter;  
    private int nGramCounter = 0;  
  
    public void run() {  
        if (stop > words.size()) stop = words.size();  
  
        List<String> ngrams = new ArrayList<>();  
        for (int i=start; i<stop; i++) {  
            String word = words.get(i);  
            if (word.length() >= n) {  
                for (int j = 0; j < word.length() - (n-1); j++) {  
                    ngrams.add(word.substring(j, j+n));  
                    nGramCounter++;  
                }  
            }  
        }  
        globalCounter.addAndGet(nGramCounter);  
    }  
}
```

Versione Parallelia con Java Thread

- Nel main vengono lanciati un numero variabile di thread e ne viene fatto il *join* per aspettare che terminino
- Il **tempo di esecuzione** viene misurato con la funzione ***currentTimeMillis()*** della libreria standard *System*

```
AtomicInteger gCounter = new AtomicInteger(0);

List<NGramThread> threads = new ArrayList<>();
int blockSize = words.size() / numThreads + 1;

int i = 0;
int n = 2; // 2: bigrammi, 3: trigrammi

long start = System.currentTimeMillis();

while (i < numThreads) {
    int s = i * blockSize; // indice partenza
    int e = (i + 1) * blockSize; // indice arrivo
    threads.add(new NGramThread(gCounter, words, n, s, e));
    threads.get(i).start();
    i++;
}

for (NGramThread worker: threads) {
    try {
        worker.join();
    } catch (InterruptedException ignored) {}
}

long end = System.currentTimeMillis();
```



Versione Parallelia con C++ Thread

- Crea una classe **“LinesExtractor”** che prende in ingresso il nome del file **txt** ed estrae una lista di tutte le righe del testo

```
vector<string> extractLines(const string& filename) {  
  
    vector<string> lines;  
  
    string line;  
    ifstream file(filename);  
  
    if (!file) {  
        throw runtime_error("Could not open file!");  
    }  
  
    regex reg("[^A-Za-z0-9']");  
  
    while (getline(file, line)) {  
        transform(line.begin(), line.end(), ::tolower);  
        string cleaned = regex_replace(line, reg, "");  
        if (!cleaned.empty()) {  
            lines.push_back(cleaned);  
        }  
    }  
  
    file.close();  
    return lines;  
}
```

- Si estrae la **lista di tutte le parole** tramite la classe **stringstream** e l'iteratore **istream_iterator**

```
vector<string> lines = extractLines(filename);  
vector<string> words;  
for (auto &line: lines) {  
    stringstream ss(line);  
    istream_iterator<string> begin(ss), end;  
    vector<string> lineWords(begin, end);  
    for (auto &word: lineWords) {  
        words.push_back(word);  
    }  
}
```



Versione Parallelia con C++ Thread

- Utilizzato un ***atomic_int*** per il conteggio dei bigrammi e trigrammi trovati per evitare inconsistenze
- Usati i thread ***std::thread*** lanciati con una funzione *run()* che prende in ingresso:

- *globalCounter*: riferimento alla variabile di tipo *atomic_int*, rappresenta il contatore globale del numero di *n-grammi* trovati
- *words*: l'intero vettore di parole
- *n*: intero che specifica se vengono trattati bigrammi (*n*=2) oppure trigrammi (*n*=3)
- *start* e *stop*: due interi che indicano gli indici di partenza e di arrivo del range di parole su cui ciascun thread lavora

```
void run(atomic_int& counter, int id, int n,
         vector<string> words, int start, int stop) {

    int nGramCounter = 0;

    if (stop > words.size()) stop = words.size();

    vector<string> ngrams;
    for (int i=start; i<stop; i++) {
        string word = words[i];
        if (word.length() >= n) {
            for (int j=0; j<word.length()-(n-1); j++) {
                ngrams.push_back(word.substr(j, n));
                nGramCounter++;
            }
        }
    }
    counter.fetch_add(nGramCounter, memory_order_relaxed);
}
```



Versione Parallelia con C++ Thread

- Nel main vengono lanciati un numero variabile di thread e ne viene fatto il *join* per aspettare che terminino
- Il **tempo di esecuzione** viene misurato con la funzione ***now()*** della classe ***steady_clock*** della libreria ***chrono***

```
int blockSize = words.size() / numThreads + 1;
atomic_int globalCounter(0);

int n = 2; // 2: bigrammi, 3: trigrammi

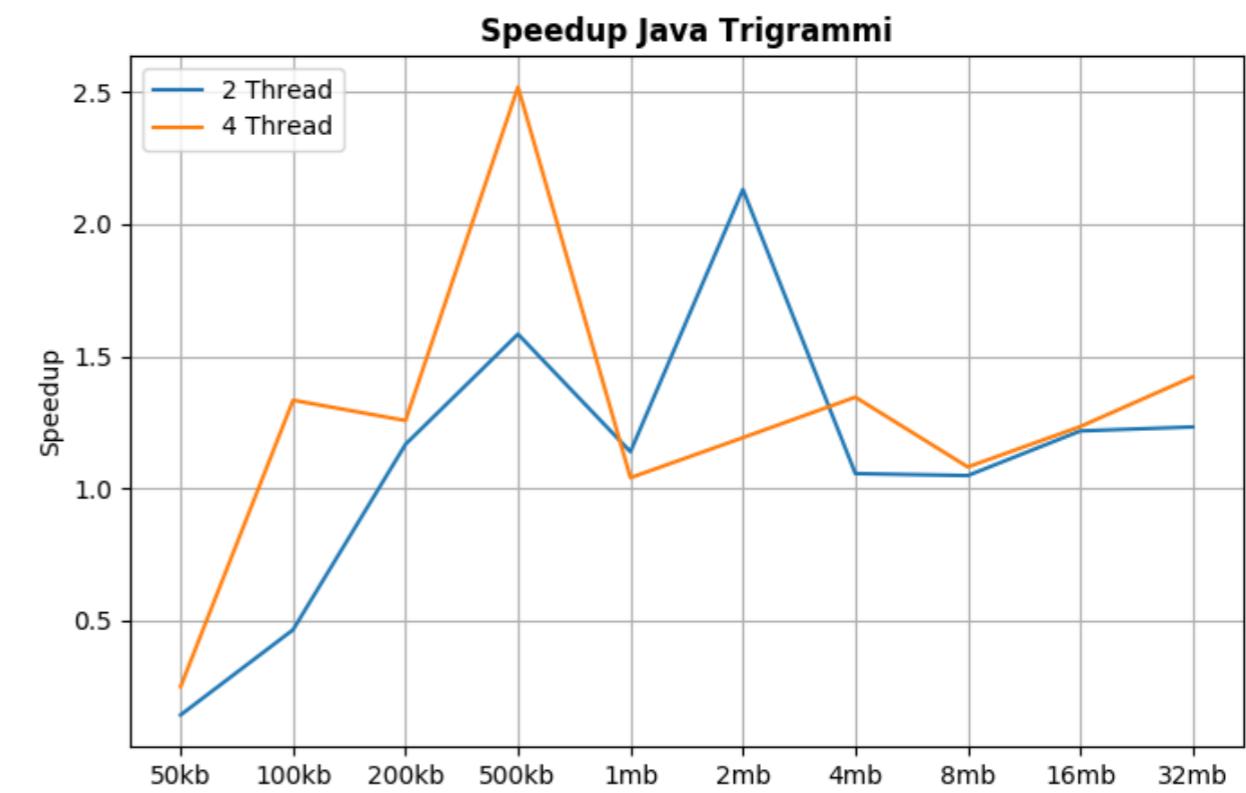
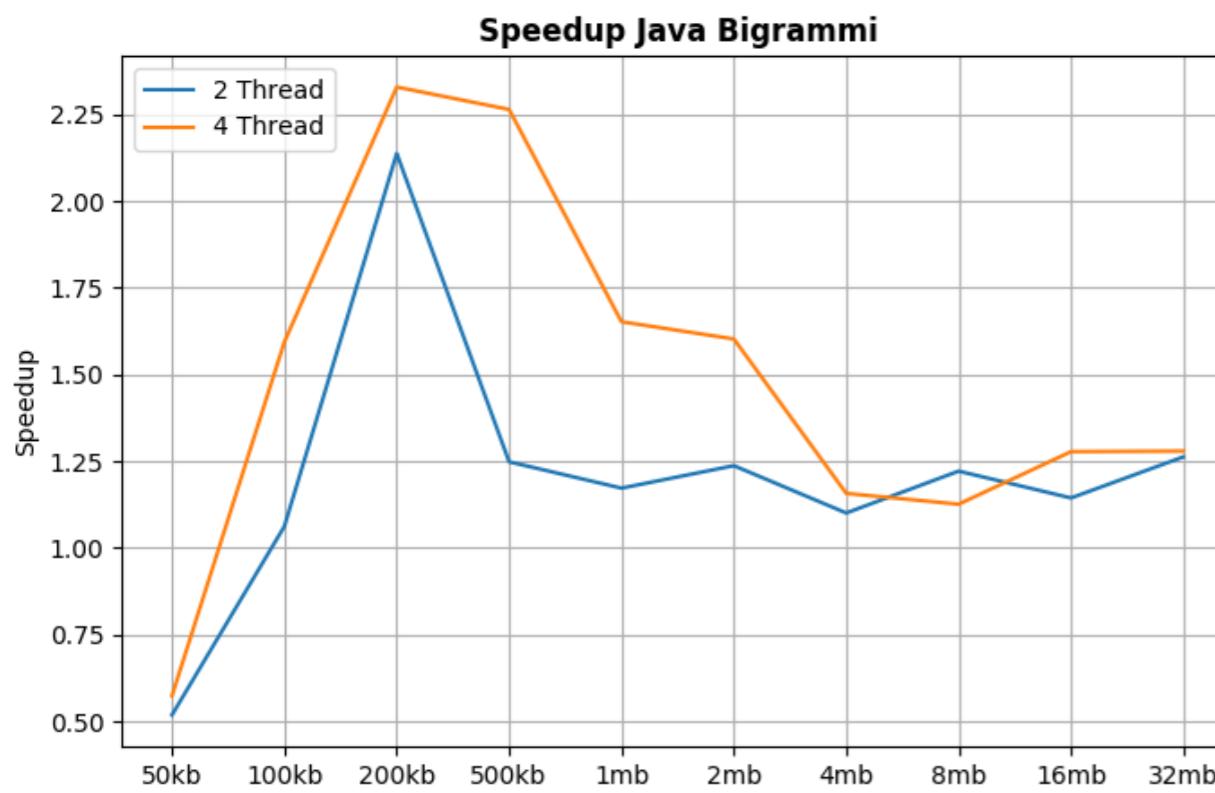
auto start = chrono::steady_clock::now();

std::vector<thread> threads(numThreads);
for (int i = 0; i < numThreads; i++) {
    int s = i*blockSize; // indice partenza
    int e = (i+1)*blockSize; // indice arrivo
    threads[i] = thread(run, ref(globalCounter),
                        i, n, words, s, e);
}
for (auto &th: threads) {
    th.join();
}

auto end = chrono::steady_clock::now();
chrono::duration<double> elapsedSeconds = end - start;
```

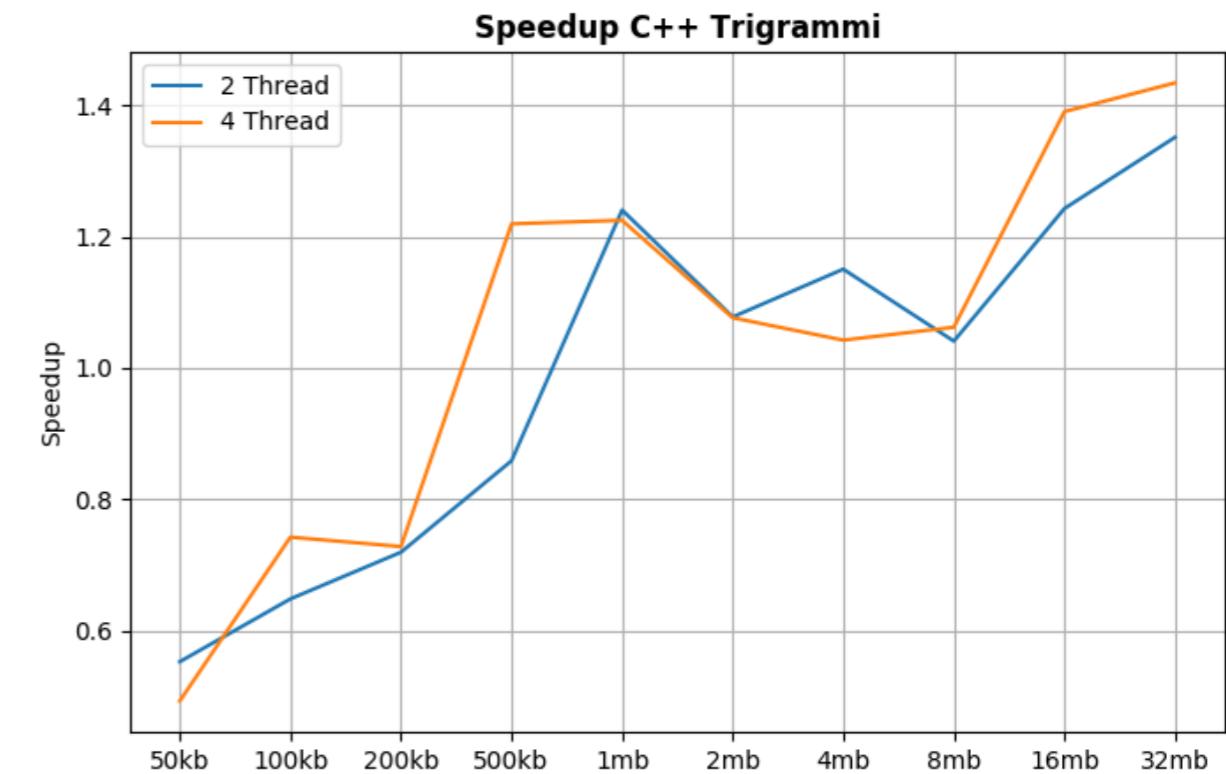
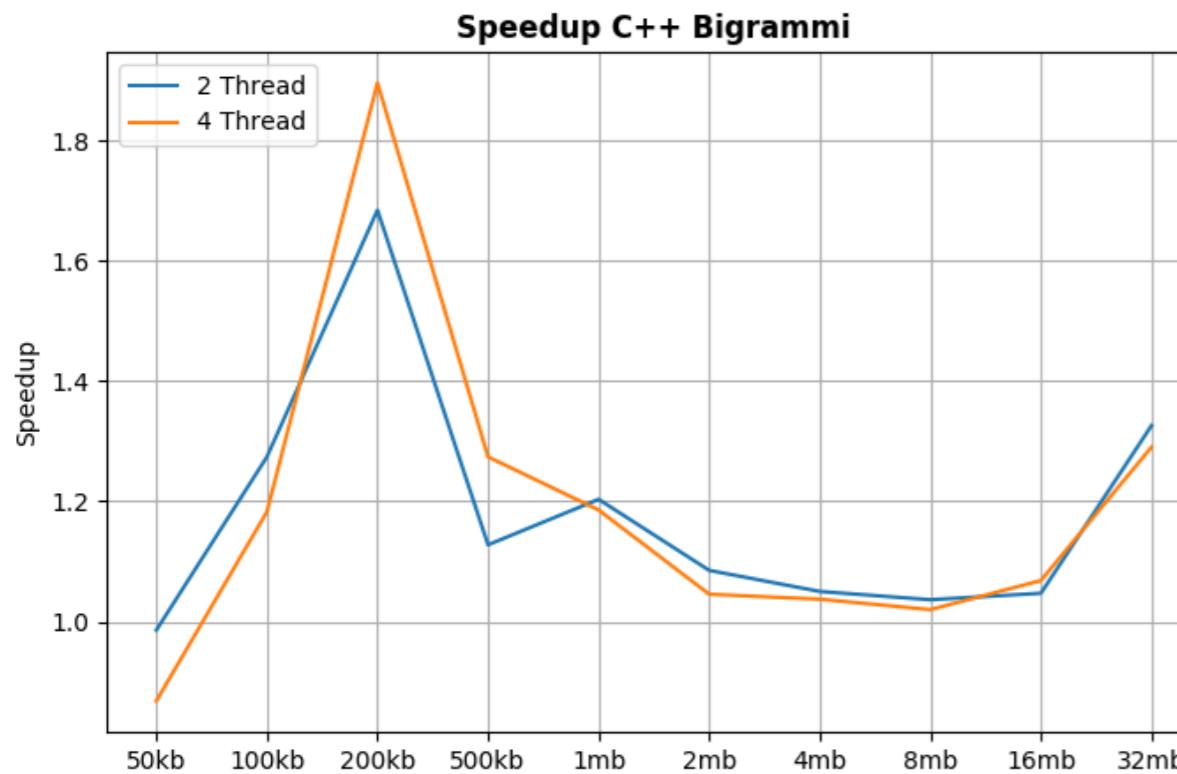
Speedup con Java Thread

- Utilizzati **2 o 4 thread**
- Le **dimensioni dei file** sono:
 - 50 kb, 100kb, 200 kb, 500 kb, 1 mb, 2 mb, 4 mb, 8 mb, 16 mb, 32 mb



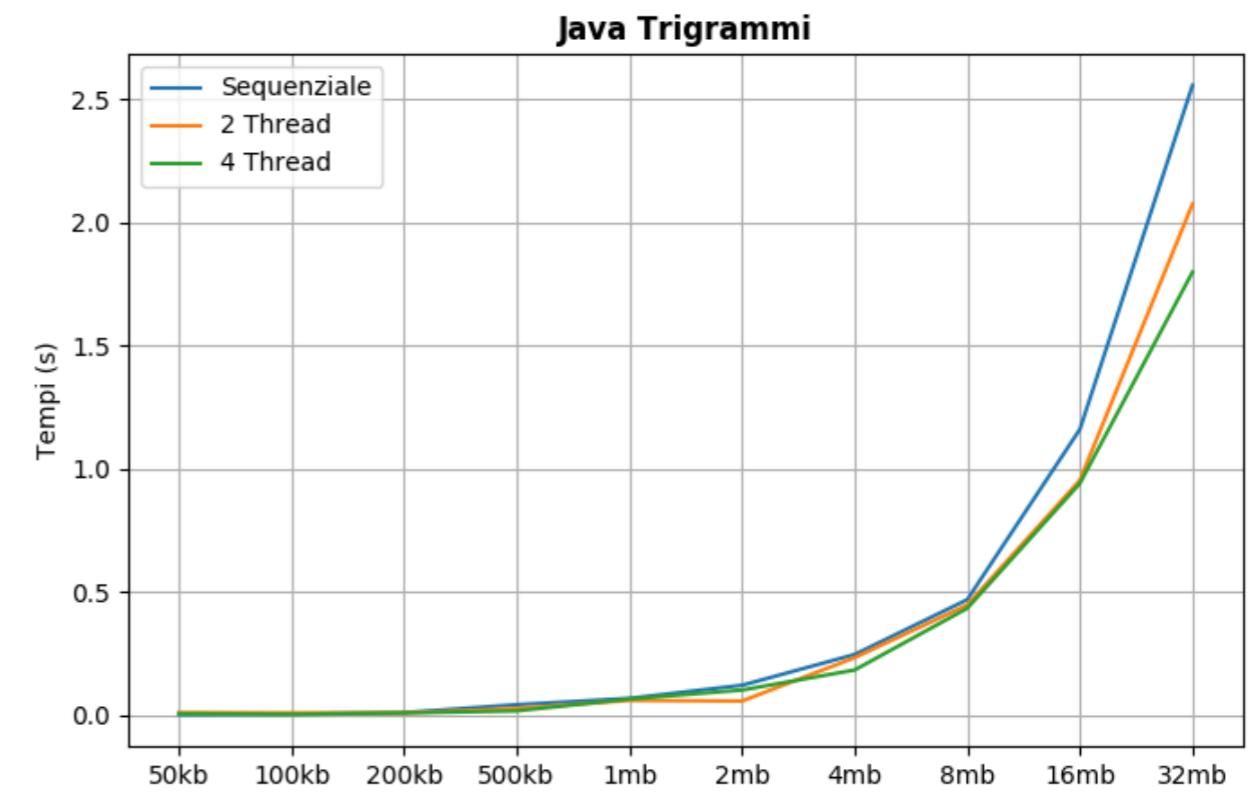
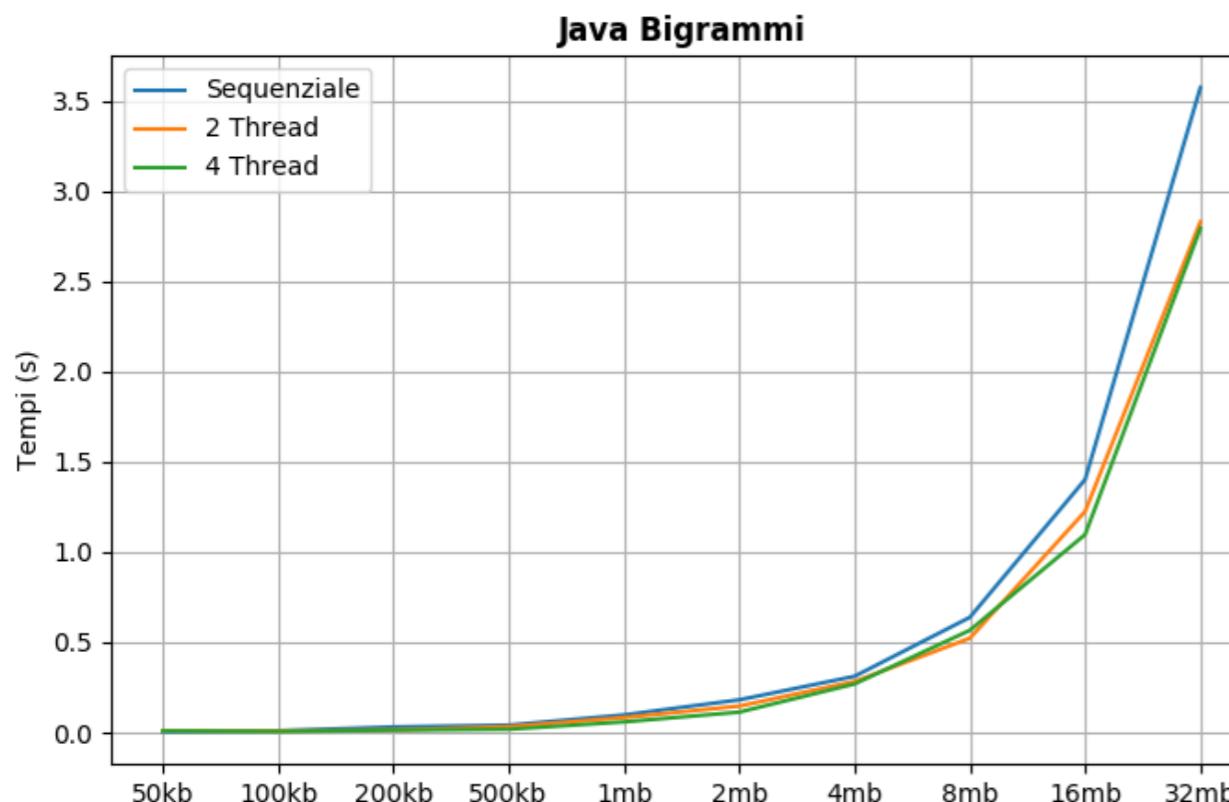
Speedup con C++ Thread

- Utilizzati **2 o 4 thread**
- Le **dimensioni dei file** sono:
- 50 kb, 100kb, 200 kb, 500 kb, 1 mb, 2 mb, 4 mb, 8 mb, 16 mb, 32 mb



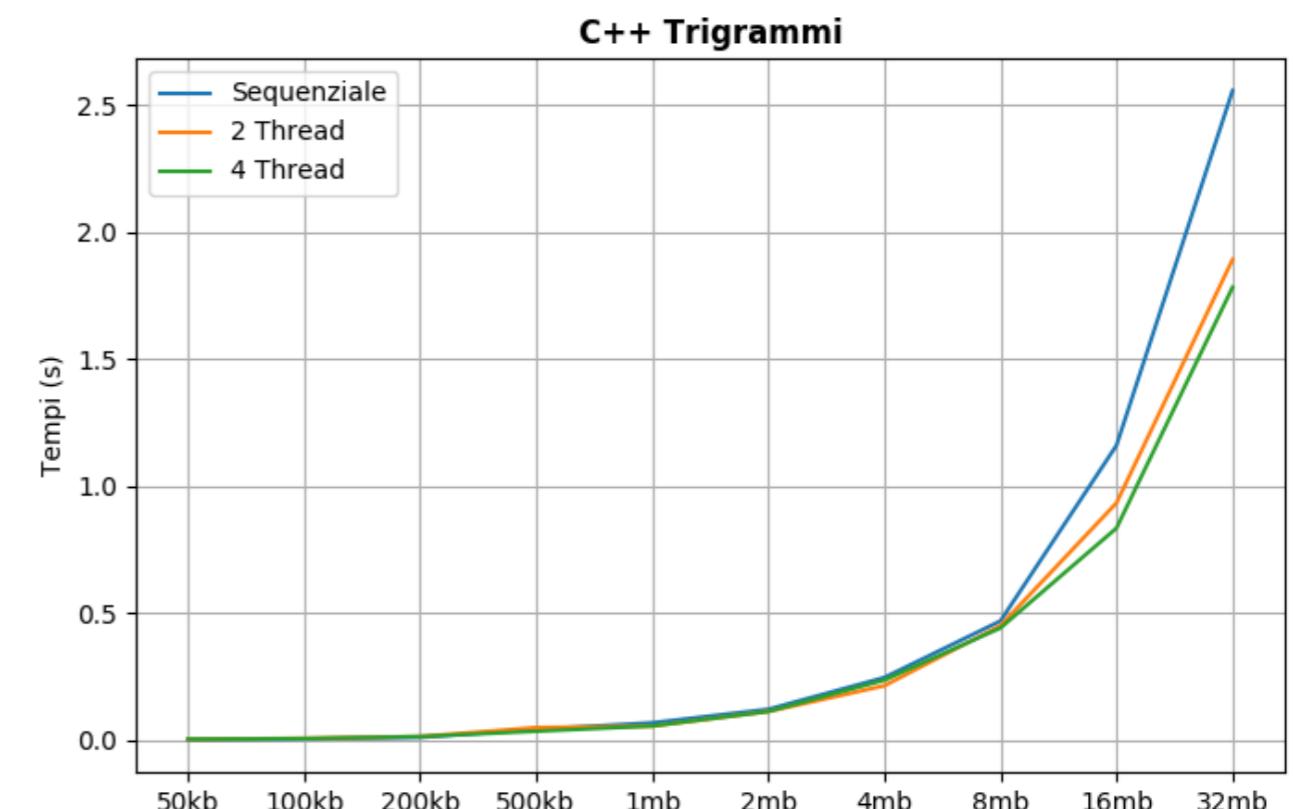
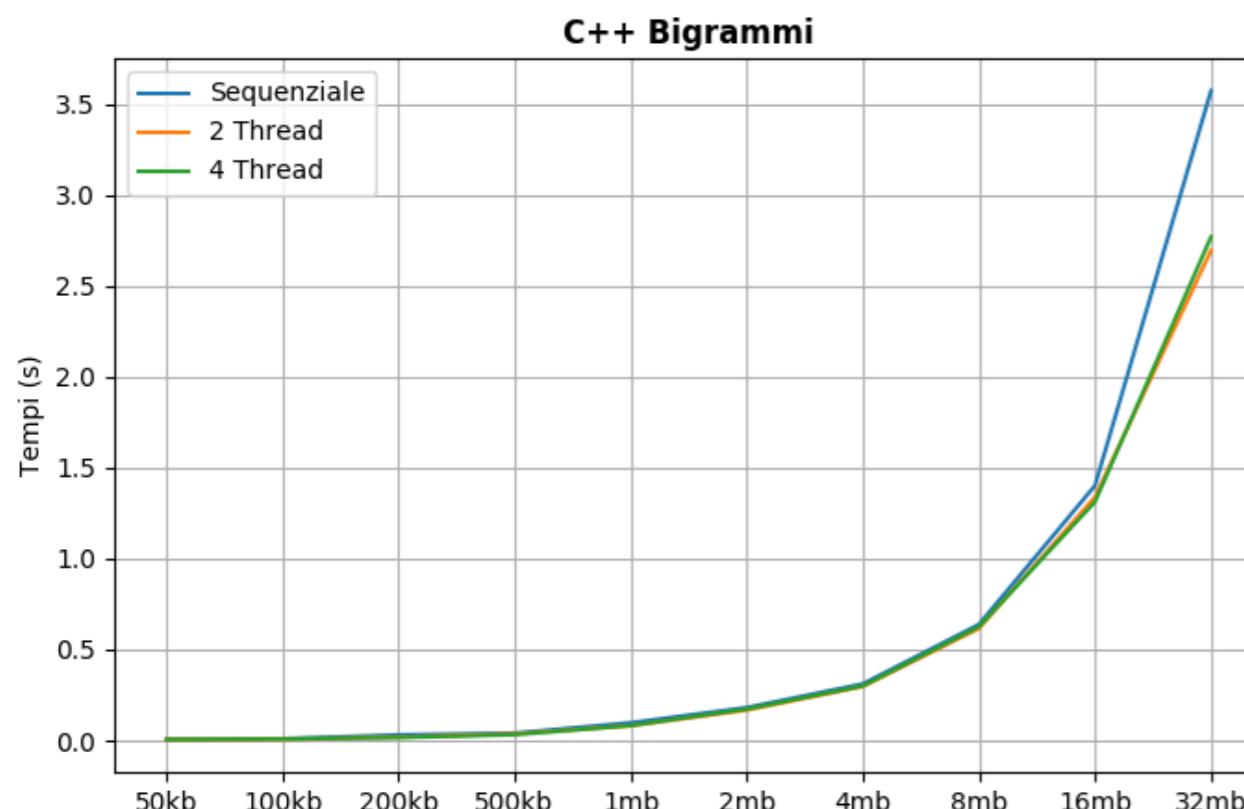
Tempi di Esecuzione con Java Thread

- Utilizzati 2 o 4 thread



Tempi di Esecuzione con C++ Thread

- Utilizzati 2 o 4 thread



Conclusione

- Con file di grosse dimensioni l'**approccio parallelo** è sempre più efficiente rispetto a quello sequenziale
- Il tempo di **estrazione righe e parole** è significativamente maggiore con **C++**
- È ragionevole pensare che aumentando ulteriormente la dimensione del file di testo si possa ottenere uno **speedup** ancora maggiore

