UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing
Conteggio di bigrammi e trigrammi
Midterm

Massimiliano Mancini

Anno accademico 2019/20

1 Introduzione

Il conteggio dei bigrammi e trigrammi trova applicazione sia nello studio statistico delle lingue [1] che in alcune tecniche di crittoanalisi classica come, per esempio, in presenza del cifrario *playfair* o in generale per l'analisi delle frequenze [2]. Il codice presentato con il seguente elaborato mette in evidenza come poter sfruttare tecniche di programmazione parallela, attraverso l'utilizzo dell'interfaccia applicativa **openmp**, per il conteggio di bigrammi e trigrammi in file di testo opportunamente predisposti.

2 Approccio utilizzato

Per il tipo di problema che ci troviamo ad affrontare, riorganizzeremo la computazione secondo uno schema di scomposizione dei dati (data decomposition). I blocchi di dati (chunk) avranno forma lineare (blockwise) e ogni thread riceverà n/p elementi dell'array oggetto della scomposizione dove n rappresenta il numero di elementi da analizzare, nel nostro caso caratteri alfabetici e p il numero di processori o thread. Il pattern utilizzato è di tipo loop parallelism in un ottica Single Program Multiple Data (SPMD).

3 Ambiente di sviluppo

Per la realizzazione del programma principale e dei programmi di supporto è stato inizialmente utilizzato l'ambiente **mingw** [3] su sistema operativo windows 10 (Figura 1), questo ha consentito una prima stesura in locale su hardware non particolarmente performante, ovvero un Dell Latitude E7250 con un solo processore fisico a due core.
Successivamente, il codice è stato migrato su piattaforma google cloud con caratteristiche superiori: un socket, 4 cores ognuno in grado di eseguire due threads per un totale di 8 CPU (Listato 1).
L'accesso al cloud è garantito attraverso una connessione ssh utilizzata anche in modalità sftp per l'accesso ai file sorgenti al fine di elaborarli con l'editor notepad++.

Listing 1: Caratteristiche gCloud

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU @ 2.30GHz
Stepping:              0
CPU MHz:               2300.000
BogoMIPS:              4600.00
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0-7
```

4 Compilazione dei sorgenti

Il file principale `main` è compilato con i seguenti parametri
`g++ -Wall -Wextra -O3 -fopenmp main.cpp -o main`
oppure, in modalità non parallela
`g++ -Wall -Wextra -O3 main.cpp -o main0`
In entrambi i casi la compilazione avviene senza errori né avvisi (warning) grazie ad alcune macro di tipo `#ifdef _OPENMP` che includono alcune righe di codice e direttive dell'interfaccia openmp solo in caso di presenza dell'apposita libreria (-fopenmp).

5 Utilizzo dei programmi

Il programma `main0`, compilato senza includere la libreria openmp, ovvero nella versione sequenziale, accetta due parametri: il file da analizzare e il numero di caratteri da analizzare. Se il numero di caratteri da analizzare è superiore alla

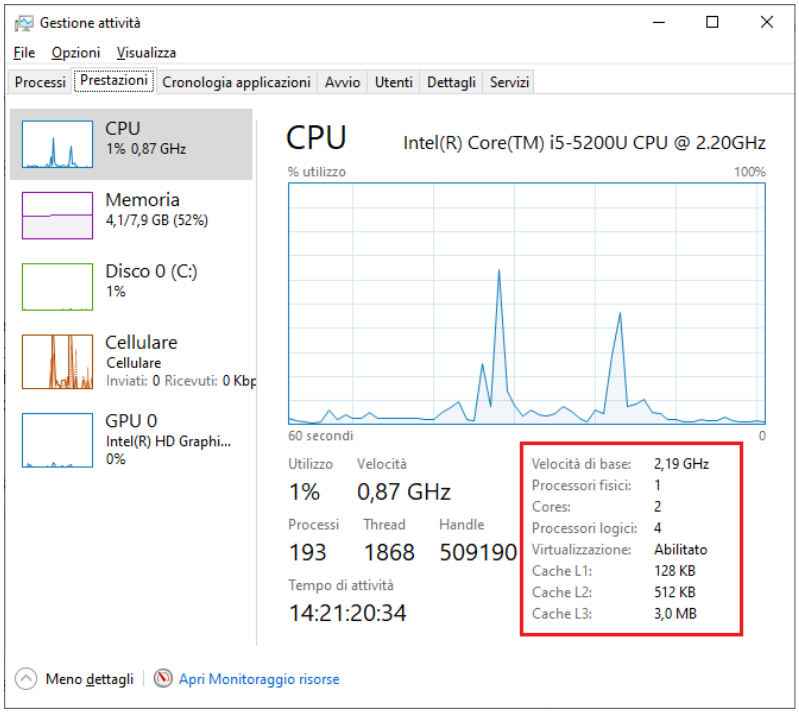

Figura 1: Processori su portatile Dell

dimensione del file, il programma analizzerà tutti i caratteri che compongono il file.
Il programma `main` accetta un ulteriore parametro che rappresenta il numero di thread da utilizzare per il conteggio dei bigrammi e trigrammi.
L'esecuzione del programma, sia in versione sequenziale che in versione parallela, porta alla produzione di tre file di testo:

- `bigrams.txt` che riporta in ordine alfabetico i bigrammi e il relativo numero di occorrenze;
- `trigrams.txt` che riporta in ordine alfabetico i trigrammi e il relativo numero di occorrenze;
- `log.txt` che riporta i tempi della parte del programma parallela indicando il numero di thread utilizzati per l'elaborazione, il numero di caratteri analizzati e il tempo di esecuzione

Per esempio il comando
`./main file1.txt 10000000000 8` esegue il conteggio dei bigrammi e dei trigrammi nel file `file1.txt` per una dimensione massima di 10 GiB utilizzando 8 thread.

6 Tempi e performance

Il tipo di compito affrontato in questo elaborato è caratterizzato da lunghi tempi di lettura del file che se non considerati nel giusto contesto, rischiano di confondere i risultati finali. Proprio per questa ragione si è deciso di conteggiare solo il tempo della componente parallela del programma. L'intero programma è suddivisibile in tre sezioni di cui solo quella centrale è parallela:

1. **Lettura del file da disco**: questa fase è lenta e dipende dall'accesso al disco. La parallelizzazione di questa sezione sarebbe controproducente nel contesto di questo elaborato. Un'ipotesi di parallelizzazione dovrebbe partire da sistemi distribuiti con diversi dischi fisici che leggono file diversi.
2. **Conteggio dei bigrammi e trigrammi**: questa è la fase per la quale ci interessa produrre il massimo speedup. Tutto il contenuto del file è infatti stato caricato in memoria centrale e vogliamo sfruttare l'interfaccia openmp e i core a nostra disposizione per ridurre al massimo il tempo di conteggio. I risultati del conteggio sono posti di nuovo in memoria in due vettori di dimensione n^2 e n^3 (dove n rappresenta il numero di lettere dell'alfabeto, ovvero 26) per il conteggio rispettivamente di bigrammi e trigrammi. Si noti che da un punto di vista della programmazione, il conteggio dei bigrammi e dei trigrammi avviene in un unico ciclo `for`. Un'eventuale implementazione naïf avrebbe potuto implementare due diversi cicli, eventualmente in parallelo ma con performance peggiori.

3. `Scrittura dei risultati`: il contenuto dei due vettori di cui al punto precedente viene scritto su disco. Di nuovo, questa sezione, è sequenziale sia perché particolarmente rapida che per evitare accessi paralleli al disco che sappiamo peggiorare i tempi complessivi.

Listing 2: Output di TOP

```
top - 17:14:23 up  1:35,  2 users,  load average: 1.60, 0.35, 0.18
Threads:  16 total,  16 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s): 64.9 us,  4.4 sy,  0.0 ni, 30.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 20565084 total,   528272 free, 10117360 used,  9919452 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 10128240 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 5036 massimi+  20   0 9908052 9.316g   2976 R 70.7 47.5   0:28.17 main
 5063 massimi+  20   0 9908052 9.316g   2976 R 44.0 47.5   0:01.32 main
 5059 massimi+  20   0 9908052 9.316g   2976 R 41.3 47.5   0:01.24 main
 5064 massimi+  20   0 9908052 9.316g   2976 R 36.3 47.5   0:01.09 main
 5060 massimi+  20   0 9908052 9.316g   2976 R 36.0 47.5   0:01.08 main
 5061 massimi+  20   0 9908052 9.316g   2976 R 36.0 47.5   0:01.08 main
 5062 massimi+  20   0 9908052 9.316g   2976 R 34.3 47.5   0:01.03 main
 5070 massimi+  20   0 9908052 9.316g   2976 R 30.3 47.5   0:00.91 main
 5068 massimi+  20   0 9908052 9.316g   2976 R 30.0 47.5   0:00.90 main
 5058 massimi+  20   0 9908052 9.316g   2976 R 29.7 47.5   0:00.89 main
 5066 massimi+  20   0 9908052 9.316g   2976 R 27.7 47.5   0:00.83 main
 5071 massimi+  20   0 9908052 9.316g   2976 R 27.3 47.5   0:00.82 main
 5072 massimi+  20   0 9908052 9.316g   2976 R 27.3 47.5   0:00.82 main
 5065 massimi+  20   0 9908052 9.316g   2976 R 26.7 47.5   0:00.80 main
 5069 massimi+  20   0 9908052 9.316g   2976 R 26.7 47.5   0:00.80 main
 5067 massimi+  20   0 9908052 9.316g   2976 R 26.3 47.5   0:00.79 main

top - 17:14:26 up  1:35,  2 users,  load average: 1.60, 0.35, 0.18
Threads:  16 total,  16 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 20565084 total,   528188 free, 10117412 used,  9919484 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 10128168 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 5060 massimi+  20   0 9908052 9.316g   2976 R 50.2 47.5   0:02.59 main
 5062 massimi+  20   0 9908052 9.316g   2976 R 50.2 47.5   0:02.54 main
 5067 massimi+  20   0 9908052 9.316g   2976 R 50.2 47.5   0:02.30 main
 5069 massimi+  20   0 9908052 9.316g   2976 R 50.2 47.5   0:02.31 main
 5036 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:29.67 main
 5058 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.39 main
 5059 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.74 main
 5061 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.58 main
 5063 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.82 main
 5064 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.59 main
 5065 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.30 main
 5066 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.33 main
 5071 massimi+  20   0 9908052 9.316g   2976 R 49.8 47.5   0:02.32 main
 5068 massimi+  20   0 9908052 9.316g   2976 R 49.5 47.5   0:02.39 main
 5070 massimi+  20   0 9908052 9.316g   2976 R 49.5 47.5   0:02.40 main
 5072 massimi+  20   0 9908052 9.316g   2976 R 49.5 47.5   0:02.31 main
```

Durante i vari test sono state catturate alcune schermate del comando `top -H -p <pid>` allo scopo di verificare che effettivamente tutti i thread fossero stati avviati come richiesto. Una schermata esemplificativa catturata durante l'esecuzione con 16 thread, è riportata in Listato 2

Per accumulare dati di performance è stato predisposto un piccolo bash script che richiama il programma principale con un numero crescente di caratteri da analizzare, da 1 GiB fino a 10 GiB e un numero crescente di thread da 1 fino a 16.

Si noti come i tempi di esecuzione della versione sequenziale del programma siano leggermente inferiori rispetto alla versione parallela invocata con un solo thread. Le righe seguenti mostrano il file di log che ha registrato i tempi indicando con 0 la versione sequenziale e con 1 la versione con un solo thread, sullo stesso file, analizzando 5 GiB con tempi intorno agli 11.7 secondi.

```
0, 5000000000, 11.6025
```

```
0, 5000000000, 11.5712
0, 5000000000, 11.5506
1, 5000000000, 11.8045
1, 5000000000, 11.7958
1, 5000000000, 11.7889
```

7 Accuratezza

Sebbene il programma accetti qualsiasi tipo di file di testo ed esegua una sanificazione dei caratteri in fase di analisi, se si utilizza un file composto da soli caratteri alfabetici (a-z), l'accuratezza del conteggio è perfetta. Per condurre opportune verifiche di accuratezza, è stato predisposto un secondo eseguibile `genfile` che, ricevuto in input il numero di caratteri, genera sia il file di caratteri casuali che ulteriori due file con il numero di bigrammi e trigrammi in esso presenti. Tali file possono essere utilizzati per un confronto dell'output prodotto dal programma `main`.
Nel caso si intenda sanificare un generico file di testo, può essere utilizzato il programma `preparefile` che produce un file adatto all'utilizzo con `main`.

8 Metodo di rilevazione dei tempi di esecuzione

I tempi di esecuzione della parte parallela del codice sono rilevati attraverso `high_resolution_clock`. Una prima rilevazione è fatta immediatamente prima l'ingresso della sezione parallela e una seconda rilevazione all'uscita della stessa. La classe `std::chrono::duration` fornisce il corretto intervallo.

9 Direttive OPENMP

Nel codice sono utilizzate due sole direttive dell'interfaccia applicativa OPENMP:

- `#pragma omp parallel default(none) shared(buffer, length, freqb, freqt, blocksize)`

  Si utilizza la clausola `default(none)` e si condividono soltanto le variabili necessarie:

  - `buffer, length` che contengono rispettivamente la totalità dei caratteri da analizzare e la sua dimensione;
  - `freqb, freqt` ossia i vettori che conterranno i conteggi (le frequenze) di bigrammi e trigrammi;
  - `blocksize` ossia la dimensione di ogni chunk di buffer da assegnare a ogni thread. Per ottenere la massima accuratezza, tale dimensione viene calcolata come il multiplo di 6 più vicino al quoziente ottenuto dividendo il numero di caratteri da analizzare per il numero di thread. La scelta del numero 6 sarà esposta nella prossima sezione.

- `#pragma omp for reduction (+:freqb,freqt) schedule (static, blocksize)`

  Il blocco di dati da analizzare viene suddiviso tra i diversi thread in un pattern di tipo loop parallelism. Alla fine del ciclo i vettori freqb e freqt sono oggetto di riduzione tramite l'operazione di somma. Per mantenere l'accuratezza del conteggio viene utilizzata la clausola `schedule (static, blocksize)` che garantisce che la suddivisione avvenga solo in punti predeterminati.

10 Algoritmo

L'algoritmo utilizzato consente il conteggio in un unico passaggio di bigrammi e trigrammi componendo un piccolo vettore di 6 caratteri in modo possano essere conteggiati 3 bigrammi e 2 trigrammi contemporaneamente. Da qui dipende la scelta di suddividere il file in multipli di 6 per garantire l'accuratezza dei conteggi. I bigrammi e trigrammi individuati incrementano i relativi contatori nei due vettori `freqb` e `freqt` i cui indici sono linearizzati proprio a partire dal bigramma o trigramma individuato. Sebbene la compilazione con `-O3` sia già sufficiente, per migliorare ulteriormente le performance in questa parte di codice viene fatto un unrolling di un eventuale ciclo interno che scorra i tre bigrammi e i due trigrammi. La parte significativa di codice è riportata in Listato 3.

Listing 3: Algoritmo principale

```
for (long long int j = 0; j <= length; j++) {
    // current char
    cc = buffer[j];

    // sanitization
    if (cc >= 'A' && cc <= 'Z') {
        cc = cc + 32;
    }

    // insert char in c buffer
    if (cc >= 'a' && cc <= 'z') {
        c[i] = cc;
        i++;
    }

    // if c buffer is full, count 3 bigrams and 2 trigrams
    // avoid another for cicle by unrolling
    if (i == 6) {
        ib = (c[0] - 'a') * NL + c[1] - 'a';
        freqb[ib] = freqb[ib] + 1;
        ib = (c[2] - 'a') * NL + c[3] - 'a';
        freqb[ib] = freqb[ib]+1;
```

```
            ib = (c[4] - 'a') * NL + c[5] - 'a';
            freqb[ib] = freqb[ib]+1;
            it = (c[0] - 'a') * NL2 + (c[1] - 'a') * NL + c[2] - 'a';
            freqt[it] = freqt[it]+1;
            it = (c[3] - 'a') * NL2 + (c[4] - 'a') * NL + c[5] - 'a';
            freqt[it] = freqt[it]+1;

            // reset c buffer
            i = 0;
        }
}
```

11 Analisi dei risultati

Sia la versione sequenziale che quella parallela del programma principale producono un file di log su cui vengono registrati i dati significativi di ogni esecuzione, in particolare il numero di thread utilizzati (0 nel caso della versione sequenziale), il numero di caratteri analizzati e il tempo impiegato per il conteggio corrispondente alla porzione parallela del codice. Dall'analisi dei tempi è stata volutamente rimossa la parte sequenziale del programma perché si tratta soltanto di operazioni di lettura e scrittura su disco che hanno tempi di molto superiori a quelli necessari per il conteggio dei bigrammi e trigrammi che rappresenta la parte parallela. Condurremo quindi le nostre analisi considerando nullo il tempo di esecuzione non parallela per concentrarci solo sulla parte parallelizzata.

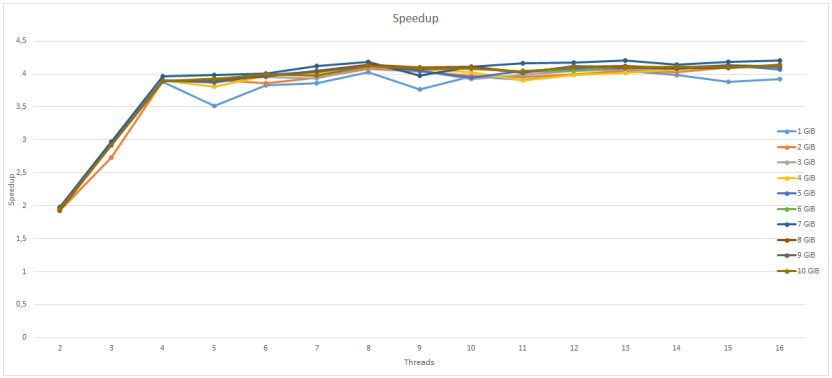

Figura 2: Speedup

Ricordando la formula dello speedup

$$S_P = \frac{t_s}{t_P}$$

dove t_s è il tempo di esecuzione sequenziale e t_P è il tempo di esecuzione parallela e P il numero di processori, come osservabile in Figura 2, questi ha un

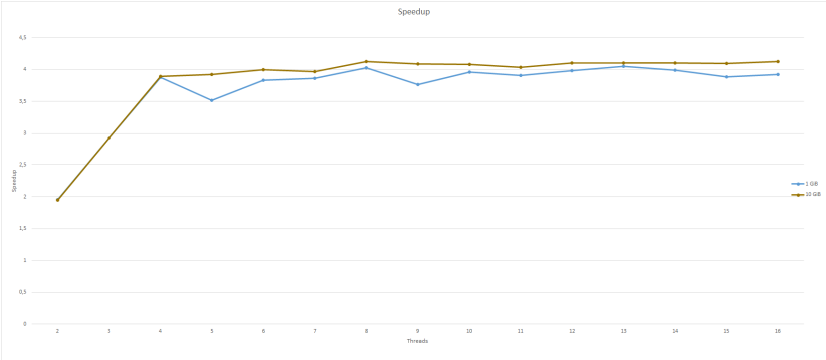

Figura 3: Speedup per 1 e 10 GiB

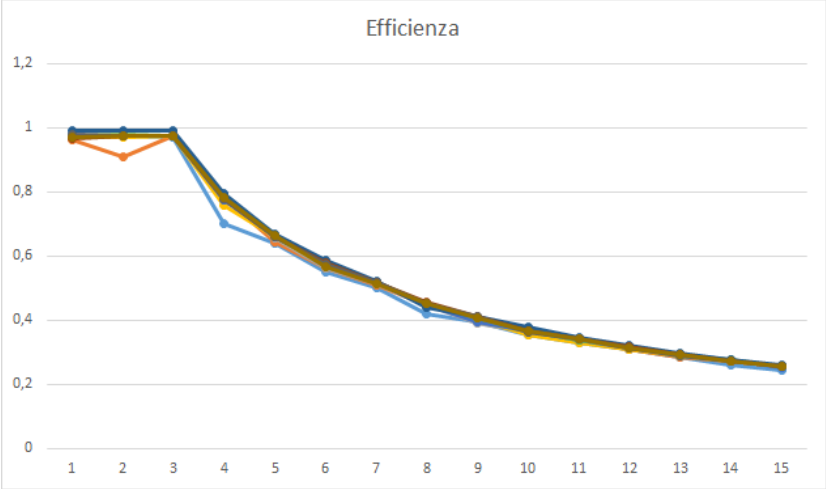

Figura 4: Efficienza

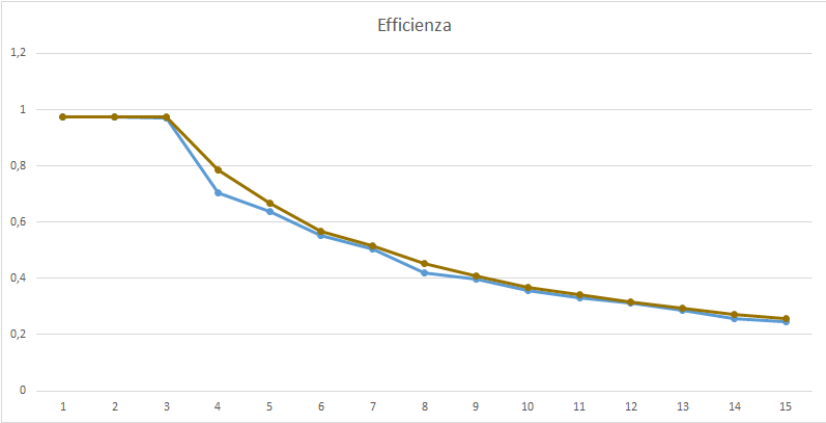

Figura 5: Efficienza per 1 e 10 GiB

andamento lineare in caso di $P < 5$. Con $P > 4$ non si osserva speedup nemmeno in presenza di file di grandi dimensioni (10 GiB). L'andamento è confermato anche dalla rilevazione dell'efficienza visualizzabile in Figura 4. Ricordiamo che l'efficienza è data dallo speedup diviso il numero di processori. Analizzando il comportamento nei due casi estremi, 1GiB e 10 GiB, come riportato in Figura 3 e Figura 5, si osserva che una maggiore quantità di dati favorisce un comportamento più stabile.

Codici

Listing 4: main

```
#include <fstream>
#include <iostream>
#include <cstring>
#include <stdio.h>
#include <chrono>
#include <ctime>
#include <cstdlib>
#include <omp.h>

// Number of letters
#define NL 26
#define NL2 NL*NL
#define NL3 NL*NL*NL


int main(int argc, char *argv[])
{
    // Checking parameters, if openmp is activated then number of threads is allowed
    #if defined(_OPENMP)
    if (argc < 3) {
        std::cout << "Usage: " << argv[0] << " <filename> <size> <num_threads>\n";
        return 1;
    }
    #else
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <filename> <size> \n";
        return 1;
    }
    #endif

    // NL is number of letters
    int freqb[NL2] = {0};
    int freqt[NL3] = {0};

    // Some counters
    char c1 = 'a';
    char c2 = 'a';
    char c3 = 'a';

    // File length
    long long int length = 0;

    #if defined(_OPENMP)
    int nthreads = atoi(argv[3]);

    // check if second argument is
    if (nthreads < 1) {
        std::cout << "Warning: invalid threads number. Default to 4 threads\n";
        nthreads = 4;
    }
    #else
    int nthreads = 0;
    #endif

    std::chrono::high_resolution_clock::time_point t1, t2;
    std::chrono::duration<double> time_span;

    // open up all files input and output
    std::ifstream is (argv[1], std::ifstream::in);
    std::ofstream ob ("bigrams.txt", std::ofstream::out);
    std::ofstream ot ("trigrams.txt", std::ofstream::out);
    std::ofstream log ("log.txt", std::ofstream::app);

    if (is) {
        is.seekg (0, is.end);
        length = is.tellg();
        is.seekg (0, is.beg);
    } else {
        std::cout << "Error in open input file " << argv[1];
    }

    length = std::min(length, atoll(argv[2]));

    // allocate memory for reading file:
    char* buffer = new char [length];

    // read data as a block:
    is.read (buffer,length);

    // start time for parallel part (consumers)
    t1 = std::chrono::high_resolution_clock::now();

    #if defined(_OPENMP)
    // set numeber of threads
```

```
omp_set_num_threads(nthreads);

// set block size to the nearest multiple of 6 in order to avoid bi/trigrams shift
long long int blocksize = ((length/nthreads)/6)*6;
#pragma omp parallel default(none) shared(buffer, length, freqb, freqt, blocksize)
#endif
{
    int c[6] = {0};
    int cc = 0;
    int i = 0;
    int ib = 0;
    int it = 0;

    #if defined(_OPENMP)
    #pragma omp for reduction (+:freqb,freqt) schedule (static, blocksize)
    #endif
    for (long long int j = 0; j <= length; j++) {
        // current char
        cc = buffer[j];

        // sanification
        if (cc >= 'A' && cc <= 'Z') {
            cc = cc + 32;
        }

        // insert char in c buffer
        if (cc >= 'a' && cc <= 'z') {
            c[i] = cc;
            i++;
        }
        // if c buffer is full, count 3 bigrams and 2  trigrams
        // avoid another for  cicle by unrolling
        if (i == 6) {
            ib = (c[0] - 'a') * NL + c[1] - 'a';
            freqb[ib] = freqb[ib] + 1;
            ib = (c[2] - 'a') * NL + c[3] - 'a';
            freqb[ib] = freqb[ib]+1;
            ib = (c[4] - 'a') * NL + c[5] - 'a';
            freqb[ib] = freqb[ib]+1;
            it = (c[0] - 'a') * NL2 + (c[1] - 'a') * NL + c[2] - 'a';
            freqt[it] = freqt[it]+1;
            it = (c[3] - 'a') * NL2 + (c[4] - 'a') * NL + c[5] - 'a';
            freqt[it] = freqt[it]+1;

            // reset c buffer
            i = 0;
        }

    }
}

t2 = std::chrono::high_resolution_clock::now();
time_span = std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);

log << nthreads << ", " << length << ", " << time_span.count() << "\n";


for (int i = 0; i < NL2; i++) {
    if (freqb[i] > 0) {
        ob << c1 << c2 << " " << freqb[i] << "\n";
    }
    c2 = c2 + 1;
    if (c2 > 'z') {
        c2 = 'a';
        c1 = c1 + 1;
    }
}
c1 = 'a';
c2 = 'a';
for (int i = 0; i < NL3; i++) {
    if (freqt[i] > 0) {
        ot << c1 << c2 << c3 << " " << freqt[i] << "\n";
    }
    c3 = c3 + 1;
    if (c3 > 'z') {
        c3 = 'a';
        c2 = c2 + 1;
    }
    if (c2 > 'z') {
        c2 = 'a';
        c1 = c1 + 1;
    }
}

free(buffer);

is.close();
ob.close();
ot.close();
log.close();
```

}

Listing 5: genfile

```
#include <fstream>
#include <iostream>
#include <cstring>
#include <stdio.h>
#include <cstdlib>
#include <time.h>

#define NL 26
#define NL2 NL*NL
#define NL3 NL*NL*NL


int main(int argc, char *argv[])
{
    // Checking parameters, if openmp is activated then number of threads is allowed
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <filename> <size> \n";
        return 1;
    }

    // NL is number of letters
    int freqb[NL2] = {0};
    int freqt[NL3] = {0};

    // Some counters
    char c1 = 'a';
    char c2 = 'a';
    char c3 = 'a';

    // open up all files input and output
    std::ofstream of (argv[1], std::ofstream::out);
        std::ofstream ob ("gen_bigrams.txt", std::ofstream::out);
    std::ofstream ot ("gen_trigrams.txt", std::ofstream::out);

    // File length
        long long int length = 0;
    length = atoll(argv[2]);

        char c[6] = {0};
        int ib = 0;
        int it = 0;
        int i = 0;

        srand(time(NULL));

        for (long long int j = 0; j <= length; j++) {
                c[i] = rand() % 26 + 97;

                i++;

                if (i == 6) {
                        ib = (c[0] - 'a') * NL + c[1] - 'a';
                        freqb[ib] = freqb[ib] + 1;
                        ib = (c[2] - 'a') * NL + c[3] - 'a';
                        freqb[ib] = freqb[ib]+1;
                        ib = (c[4] - 'a') * NL + c[5] - 'a';
                        freqb[ib] = freqb[ib]+1;
                        it = (c[0] - 'a') * NL2 + (c[1] - 'a') * NL + c[2] - 'a';
                        freqt[it] = freqt[it]+1;
                        it = (c[3] - 'a') * NL2 + (c[4] - 'a') * NL + c[5] - 'a';
                        freqt[it] = freqt[it]+1;

                        for (int k = 0; k <= 5; k++) {
                                of << c[k];
                        }
                        i = 0;
                }

        }

    for (int i = 0; i < NL2; i++) {
        if (freqb[i] > 0) {
            ob << c1 << c2 << " " << freqb[i] << "\n";
        }
        c2 = c2 + 1;
        if (c2 > 'z') {
            c2 = 'a';
            c1 = c1 + 1;
        }
    }
    c1 = 'a';
    c2 = 'a';
    for (int i = 0; i < NL3; i++) {
        if (freqt[i] > 0) {
            ot << c1 << c2 << c3 << " " << freqt[i] << "\n";
        }
        c3 = c3 + 1;
        if (c3 > 'z') {
            c3 = 'a';
```

```
                c2 = c2 + 1;
            }
            if (c2 > 'z') {
                c2 = 'a';
                c1 = c1 + 1;
            }
        }


        of.close();
        ob.close();
        ot.close();
    }
```

Listing 6: preparefile

```
#include <fstream>
#include <iostream>

int main(int argc, char *argv[])
{
    if (argc < 1) {
        std::cout << "Usage: " << argv[0] << " <filename>\n";
        return 1;
    }

    std::ifstream is (argv[1], std::ifstream::in);
    std::ofstream os ("preparedFile.txt", std::ofstream::out);

    char cc = 'a';

    while (is >> std::noskipws >> cc) {
        if (cc >= 'A' && cc <= 'Z') {
                cc = cc + 32;
        }

        if (cc >= 'a' && cc <= 'z') {
            os << cc;
        }
    }

    is.close();
    os.close();
}
```

Listing 7: runtests

```
#!/bin/bash
zeros="000000000"

for s in {1..10}
do
    size="$s$zeros"
    ./main0 file1.txt $size
    echo "size_$s,_sequential_mode"
done

for s in {1..10}
do
        for t in {1..16}
        do
                size="$s$zeros"
                ./main file1.txt $size $t
                echo "size_$s,_threads_$t"
        done
done
```

Riferimenti bibliografici

[1] Michael John Collins. ≪A New Statistical Parser Based on Bigram Lexical Dependencies≫. In: *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*. ACL '96. Santa Cruz, California: Association for Computational Linguistics, 1996, pp. 184–191. DOI: 10.3115/981863.981888. URL: https://doi.org/10.3115/981863.981888.

[2] Helen F. Gaines. *Cryptanalysis: A Study of Ciphers and Their Solution*. Dover Publications, apr. 1989. ISBN: 0486200973.

[3] *mingw-w64 GCC for Windows 64 & 32 bits*. http://mingw-w64.org/.