PROGETTO STRUMENTI FORMALI PER LA BIOINFORMATICA

KmerCo: Tecnica del conteggio dei K-mer basata su Bloom Filter

Studente
Chiara Puglia
Matricola: 0522501984

Docenti
Prof.ssa Rosalba Zizza
Prof. Rocco Zaccagnino
Prof.ssa Clelia De Felice

Anno Accademico: 2025/2026

1 Introduzione

Il conteggio dei K-mer è una fase fondamentale nelle pipeline di bioinformatica, in particolare per l'assemblaggio del DNA, la correzione degli errori e la costruzione di grafi de Bruijn compatti. Negli anni sono state proposte numerose tecniche per affrontare le sfide computazionali legate alla natura estremamente intensiva in dati e memoria di questo processo. Tra le strutture dati più utilizzate troviamo i Bloom Filter e le loro varianti di conteggio, che hanno permesso di ridurre l'impronta di memoria pur garantendo prestazioni elevate.

Nelle sezioni successive ci concentreremo sulle varie tecniche che sono state introdotte per il conteggio dei K-mer, volgendo particolare attenzione su K-merCO in quanto risultata più efficace rispetto alle altre tecniche per poi proseguire con la struttura e il funzionamento di quest'ultimo, a seguire la compilazione in ambiente Windows ed infine, si volgerà particolare attenzione sui risultati ottenuti, prendendo in considerazione tra i file loxodonta.fastq, balaenoptera.fastq, galeopterus.fastq e microcebus.fastq il file balaenoptera.fastq dato che rappresenta il dataset più grande.

2 Stato dell'arte

In questa sezione, presenteremo le varie tecniche di conteggio dei K-mer basate su Bloom Filter:
Jellyfish: è una tecnica leggera di conteggio dei K-mer, multi-threaded e senza lock, basata su una tabella hash. La tabella hash mantiene il conteggio dei K-mer. Lo schema lock-free consente l'elaborazione parallela dei K-mer. Ogni voce della tabella hash contiene due valori: il K-mer e la sua frequenza. Quando la tabella hash diventa satura, i dati vengono scritti su disco sotto forma di coppie K-mer/frequenza invece di aumentare la dimensione della tabella.

BFCounter: utilizza sia Bloom Filter sia tabella hash per il conteggio dei K-mer. La tabella hash mantiene il conteggio dei K-mer. Implementa un Bloom Filter standard. La sequenza di DNA viene attraversata due volte.

1. **Nella prima scansione, i K-mer vengono verificati nel Bloom Filter: se assenti, vengono inseriti nel Bloom Filter; altrimenti, nella tabella hash. Se un K-mer non è presente nella tabella hash, viene inserito; altrimenti, il suo contatore viene incrementato.**

2. **La seconda scansione serve a determinare la frequenza esatta dei K-mer. Alla fine, tutti i K-mer unici vengono eliminati. La seconda scansione impiega metà del tempo della prima, poiché la ricerca nella tabella hash è più veloce dell'inserimento. Tuttavia, BFCounter risulta lento per dataset di grandi dimensioni e la doppia scansione aumenta i tempi di elaborazione.**

Mcvicar: una tecnica di conteggio dei K-mer basata su FPGA e Bloom Filter. Il Bloom Filter genera piccole operazioni ideali per l'esecuzione su FPGA. Il CBF (Counting Bloom

Filter) mantiene il conteggio dei K-mer. La tecnica utilizza 4 FPGA, ognuno con un proprio Bloom Filter (CBF). Le letture vengono analizzate per generare K-mer, che sono salvati in piccoli blocchi e messi in una coda. Da lì, i K-mer vengono sottoposti a hashing tramite la funzione Shift-And-Xor (SAX). Il valore hash viene inviato a un selettore che decide quale FPGA processerà i K-mer. Ogni FPGA ha inoltre una coda che memorizza i K-mer inviati al rispettivo CBF. Tutti i CBF lavorano in parallelo. Le prestazioni della tecnica sono indipendenti dal valore di K. Tuttavia, se una Hybrid Memory Cube (HMC) riceve troppe operazioni, le prestazioni diminuiscono a causa della mancanza di elaborazione parallela. Questa tecnica è più adatta a sequenze di DNA di piccole dimensioni.

Squeakr: è una tecnica di conteggio dei K-mer in memoria basata su Bloom Filter che implementa il CQF (Counting Quotient Filter). Il CQF mantiene la frequenza dei K-mer. È una tecnica multi-thread che utilizza due tipi di CQF: uno globale e uno locale. Esiste un solo CQF globale, mentre ogni thread possiede un CQF locale. I thread competono per ottenere il lock sul CQF globale. Il thread che ottiene il lock inserisce direttamente il K-mer nel CQF globale, mentre gli altri lo inseriscono nel proprio CQF locale. Quando un CQF locale si satura, i dati vengono trasferiti al CQF globale. Questa tecnica utilizza una coda lock-free e un'implementazione thread-safe del CQF per parallelizzare il parsing dei file, migliorando la scalabilità con un numero crescente di thread. Tuttavia, il CQF non scala efficientemente con dataset di grandi dimensioni e fortemente sbilanciati, in quanto questi contengono regioni con K-mer molto ripetuti che causano un eccessivo contenzioso sui lock tra i thread.

SWAPCounter: è una tecnica distribuita di conteggio dei K-mer basata su Bloom Filter. Una tabella hash mantiene il conteggio dei K-mer. Implementa un CBF in cui ogni slot ha una lunghezza di contatore pari a $\log(\vartheta)$, dove ϑrappresenta la frequenza massima tra i K-mer. Il CBF esegue il conteggio dei K-mer. La tecnica presenta quattro componenti principali:

1. **I/O parallelo delle sequenze**
2. **Estrazione e distribuzione dei K-mer**
3. **Filtraggio dei K-mer**
4. **Conteggio e statistiche**

I primi tre componenti sono i più intensivi in termini di tempo, che viene ridotto implementando pipeline.

1. **L'I/O parallelo partiziona la sequenza di DNA.**
2. **Nell'estrazione dei K-mer, la sequenza viene analizzata per generare i K-mer, impacchettati in blocchi.**
3. **Nella distribuzione dei K-mer, ciascun K-mer viene sottoposto a doppio hashing per determinare il processo e la posizione in memoria. Il processo selezionato è responsabile dell'elaborazione del K-mer, che viene poi memorizzato nella posizione determinata. Successivamente, il processo interroga il CBF con il K-mer.**

I K-mer affidabili vengono memorizzati in un contenitore. Dopo il filtraggio, viene costruita una tabella hash con i K-mer affidabili. Il modulo di I/O basato su message passing interface (MPI) effettua caching e pooling dei dati per massimizzare le prestazioni di I/O.

2.1 KmerCo: Struttura e Funzionamento

In questa sezione si andrà ad analizzare nel dettaglio il funzionamento dei singoli algoritmi implementati, ognuno adibito ad una determinata funzionalità. Tra i file implementati, distinguiamo:

1. **InitBF: Rappresenta la fase di inizializzazione di Counting Bloom Filter, in cui viene calcolata la dimensione ottimale delle matrici bidimensionali x ed y. Ogni cella di countBF contiene più contatori (sub-slot), su cui si effettua il conteggio delle occorrenze dei K-mer. Lo scopo è creare una struttura compatta, veloce da aggiornare e con bassa probabilità di falsi positivi.**
2. **KmerCo: si occupa della creazione del filtro, dell'inserimento dei Kmer (non canonici e canonici) e della successiva verifica della frequenza.**
3. **Murmur: fornisce una funzione di hash rapida e affidabile utilizzata per mappare i K-mer all'interno di countBF.**
4. **Prime: si occupa della scelta delle dimensioni della matrice dei Bloom Filter e permette di derivare proporzioni tra righe e colonne in modo che la matrice sia bilanciata.**
5. **Mask: Sono maschere binarie usate nelle operazioni bit a livello di contatore. Distinguiamo in particolare, due tipi di operazioni utilizzate:**
   - **Extract mask (Me): isola i bit relativi a un contatore specifico.**
   - **Reset mask (Mr): azzera i bit relativi a quel contatore, per poter inserire il nuovo valore aggiornato senza interferenze.**

   Funzionano con operazioni logiche AND, OR, SHIFT per mantenere i contatori corretti in ogni cella del countBF.

Per quanto riguarda la compilazione dei file precedenti, è stato implementato il seguente makefile:

```
KmerCo.exe: KmerCo.o murmur.o prime.o initBF.o mask8.o
  gcc KmerCo.o murmur.o prime.o initBF.o mask8.o -o KmerCo.exe


KmerCo.o: KmerCo.c
  gcc -c KmerCo.c -o KmerCo.o


murmur.o: murmur.c
  gcc -c murmur.c -o murmur.o
```

```
11
12
13 prime.o: prime.c
14   gcc -c prime.c -o prime.o
15
16 initBF.o: initBF.c
17   gcc -c initBF.c -o initBF.o
18
19
20 mask8.o: mask8.c
21   gcc -c mask8.c -o mask8.o
22
23
24 clean:
25   del KmerCo.o murmur.o initBF.o mask8.o KmerCo.exe
```

Il makefile precedente, suggerisce la creazione degli header file che contengono le chiamate delle funzioni implementate nei corrispondenti file sorgente (file.c) per garantire una migliore leggibilità del codice e mantenere una struttura coerente.

2.2 Fasi di Funzionamento

In questa sezione si andrà ad analizzare in dettaglio come viene inizializzata la matrice dei k-mer e il criterio con cui vengono scelti quest'ultimi e successivamente inseriti nella matrice. A tal proposito, distinguiamo le sueguenti fasi:

1. Fase di Inserimento in K-merCO
2. Fase di Classificazione
3. Fase di inserimento in countBF

Nella fase di inserimento, vengono effettuati i seguenti step descritti dallo pseudocodice sottostante:

1. Viene letto il kmer dal file di DNA
2. Si calcola il reverse complement, dato che una sequenza di DNA può essere letta sia a partire da sinistra verso destra e viceversa
3. Con QcountBF-I si decide quale sia il K-mer canonico (quello con hash più piccolo)
4. Se non è presente in countBF, lo si inserisce sia in countBF che nel file Distinct.txt, altrimenti se presente, si incrementa il contatore in countBF.

Il primo k-mer viene costruito leggendo sequenzialmente i primi K caratteri dal file di input (che è stato mappato in memoria). Dopo il primo, ogni k-mer successivo viene generato a partire dal precedente in questo modo:

- Scorrimento: Il k-mer precedente viene fatto "scorrere" di una posizione a sinistra (il primo carattere viene scartato).

- Nuovo Carattere: Viene letto un solo nuovo carattere dal file di input (la sequenza di DNA) e aggiunto alla fine del k-mer.

Il seguente snippet di codice rappresenta il comportamento appena descritto:

```
ch = buff[bindex++];
kmer[--kcount] = ch;
```

Questo processo continua scorrendo il file di input carattere per carattere, finché non si raggiungono tutti i k-mer totali, calcolati come: $file\_length - Kmer\_length$.

```
[xleftmargin=0cm]

Algorithm 1 Insertion phase of KmerCo.

Input
  DNAfile: A DNA sequence file
  Cx,y : countBF
  K : Length of K-mer
  kh : Number of hash functions

Output
  Distinct file: A file containing all the distinct K-mers present in DNAfile

 procedure InsertKmerCo(DNAfile, Cx,y, K, kh)
   while Read != EOF do  EOF : End of file
      Read <- K-mer of length K
      ReadRC <- Reverse complement of Read
      if QcountBF-I(Cx,y, K, Read, ReadRC, kh, Result) = 0 then
          if Result = 0 then
              Insert Read into Distinct file
              IcountBF(Cx,y, Read, kh)
          else
              Insert ReadRC into Distinct file
              IcountBF(Cx,y, ReadRC, kh)
          end if
      else
          if Result = 0 then
              IcountBF(Cx,y, Read, kh)
          else
              IcountBF(Cx,y, ReadRC, kh)
          end if
      end if
  end while

end procedure
```

Una volta individuato il k-mer, bisogna verificare quante volte quest'ultimo compare in countBF. In tal caso, sono stati eseguiti i seguenti step rappresentati dall'algoritmo sottostante:

1. Si calcola l'hash sia del K-mer che del suo reverse complement.
2. Viene scelto il k-mer con hash più piccolo (in questo caso rappresenta quello canonico).
3. Si recuperano i valori dai contatori corrispondenti.
4. Se almeno un contatore è pari a 0, allora il K-mer non è presente, altrimenti restituisce il minimo dei contatori.

```
[xleftmargin=0cm]
Algorithm 4 Query-I operations of countBF.

1: Input
2:  Cx,y : countBF
3:  Read : Query K-mer
4:  ReadRC : Reverse complement of Read
5:  kh : Number of hash functions

6: Output
7:  Frequency
8:  Result <- {0 if Read is selected; 1 if ReadRC is selected}

9: procedure QcountBF-I(Cx,y, Read, ReadRC , kh, Result)
10:  for a : 1 to kh do
11:      h <- Ha(Read) \triangleleft Ha() is a hash function
12:      h1 <- Ha(ReadRC)
13:      if h < h1 then
14:      Result <- 0
15:      else
16:          h <- h1
17:          Result ← 1
18:      end if
19:      i <- h % x, j <- h % y, l <- h % η \triangleright{} η: number of counters in each cell
20:      value <- Cx,y \wedge Me_l \triangleright{} Me_l is the extract mask
21:      value <- value >> (α * l) \triangleright{} α: counter bit length
22:      if value = 0 then
23:          return 0
24:      else
25:          counta <- value
26:      end if
27:  end for
28:  return min(counta) \triangleleft return minimum value within count array
29: end procedure
```

CountBF è una matrice o un array bidimensionale di contatori. Nel codice sorgente KmerCo.c, countBF è implementata come una matrice dinamica di contatori: unsigned long long **aBF. Il filtro è diviso in $\boldsymbol{k}$ sottocontenitori (o "hash functions") in cui la variabile k specifica il numero di hash functions, con un massimo gestito di 8. Ogni sottocontenitore ha una dimensione specifica, indicata da m, che definisce il numero di celle totali. In seguito, distinguiamo le fasi di inizializziazione della matrice CountBF:

- Inizializzazione: Tutti i contatori nella matrice aBF vengono inizializzati a zero
- Fase di Inserimento dei K-mer: Questa è la fase in cui countBF "apprende" i k-mer dal dataset e costruisce la struttura.
- Estrazione del K-mer: Viene estratto un k-mer dalla sequenza di DNA (utilizzando il metodo a finestra scorrevole).
- Hashing: Il k-mer viene sottoposto a $\boldsymbol{k}$ funzioni di hash indipendenti. Nel codice KmerCo.c, si utilizza la funzione di hash MurmurHash2 con $\boldsymbol{k}$ diversi valori di seed.
- Mappatura della Posizione: Ogni funzione di hash produce un indice (posizione) all'interno del sottocontenitore corrispondente:

  $$\text{Posizione}_i = \text{hash}_i(\text{k-mer}) \pmod{m}$$

  dove i va da 1 a k.
- Incremento del Contatore: Per ciascuna delle $\boldsymbol{k}$ posizioni calcolate, il contatore in quella cella specifica di countBF viene incrementato di 1.

Al termine della fase di inserimento, i contatori nelle celle del countBF rappresentano una stima della frequenza dei k-mer che sono stati mappati su di esse. Successivamente il k-mer viene sottoposto alle stesse $\boldsymbol{k}$ funzioni di hash. Vengono lette le $\boldsymbol{k}$ posizioni all'interno del countBF. Il conteggio stimato del k-mer, chiamato C_{est}, è dato dal valore minimo tra i $\boldsymbol{k}$ contatori letti:

$$C_{\text{est}}(\text{k-mer}) = \min(\text{Contatore}_1, \text{Contatore}_2, \ldots, \text{Contatore}_k)$$

Questa è la logica implementata nella funzione _test_ in KmerCo.c. Se un k-mer è presente X volte, incrementerà tutte le k posizioni X volte. Se un'altra posizione è stata incrementata solo 1 volta (a causa di collisioni), il valore minimo tra i k contatori darà la stima più vicina al conteggio reale.

Nella fase di classificazione, si utilizza la matrice countBF per verificare se un k-mer è "affidabile" (trustworthy) o "erroneo" (erroneous) in base a una soglia fornita dall'utente. La classificazione avviene confrontando il conteggio stimato C_{est} con una soglia (η) fornita dall'utente.

- Se $C_{\text{est}} \geq \eta$: il k-mer è classificato come affidabile (trustworthy).
- Se $C_{\text{est}} < \eta$: il k-mer è classificato come erroneo (erroneous).

Questo comportamento, viene rappresentato dagli pseudocodici sottostanti:

```
[xleftmargin=0cm]
Algorithm 2 Classification phase of KmerCo.

Input
  Cx,y : countBF
  kh : Number of hash functions
  Distinct file: A file containing all the distinct K-mers present in the input DNA sequence file

Output
  Trustworthy File: A file containing K-mers having frequency more than τ
  Erroneous File: A file containing K-mers having frequency less than or equal to τ

 procedure QueryKmerCo(Cx,y, kh , Distinct file)
   while Distinct file do
      K-mer <- K-mer read from Distinct file
      if QcountBF-II(Cx,y, K-mer, kh) > τ then
          Insert into Trustworthy File
      else
          Insert into Erroneous File
      end if
   end while
 end procedure
```

```
Algorithm 5 Query-II operations of countBF.

1: Input
2:  Cx,y : countBF
3:  K-mer: Query K-mer
4:  kh : Number of hash functions

5: Output
6:  Return: Frequency

7: procedure QcountBF-II(Cx,y , K-mer, kh)
8:   for a : 1 to kh do
9:       h <- Ha(K-mer) \triangleleft Ha() is a hash function
10:      i <- h % x, j <- h % y, l <- h % η \triangleleft η: number of counters in each cell
11:      value <- Cx,y \wedge Me_l \triangleleft Me_l is the extract mask
12:      value <- value >> (α * l) \triangleleft α: counter bit length
13:      if value == 0 then
14:          return 0
15:      else
16:          counta <- value
```

```
17:      end if
18:   end for
19:   return min(counta) \triangleleft return minimum value within count array
20: end procedure
```

Successivamente, si analizzerà nel dettaglio il modo con cui viene calcolato il nuovo contatore e la cella in cui verrà inserito il k-mer individuato utilizzando i criteri esposti nel paragrafo precedente. Il seguente algoritmo descrive i seguenti step:

1. Si calcola la cella (i,j) e il contatore l
2. Viene estratto il valore del contatore utilizzando la mask
3. Il contatore viene incrementato di 1, verificando che non superi il massimo rappresentabile
4. Si reinserisce il nuovo valore nella cella di riferimento

```
[xleftmargin=0cm]
Algorithm 3 Insertion operation of countBF

1: Input
2:  Cx,y : countBF
3:  K-mer: A Read of length K
4:  kh : Number of hash functions

5: procedure IcountBF(Cx,y , K-mer, kh)
6:   for a : 1 to kh do
7:       h <- Ha(Read) \triangleright{} Ha() is a hash function
8:       i <- h % x, j <- h % y, l <- h % η \triangleright{} η: number of counters in each cell c
9:       value <- C(i,j) \wedge Me_l \triangleright{} Me_l is the extract mask
10:      value <- Cl >> (α * l) \triangleright{} α : counter bit length
11:      value <- value + 1
12:      if value = MAX then
13:          Counter Overflow
14:          return
15:      end if
16:      value <- value << (α * l) \triangleright{} Left-shift bit operation
17:      value <- C(i,j) \wedge Mr_l \triangleright{} Mr_l is the reset mask
18:      C(i,j) = C(i,j) \vee value
19:   end for
20: end procedure
```

3 Compilazione ed esecuzione in ambiente Windows

Per quanto riguarda la compilazione in ambiente windows, sono state apportate le seguenti modifiche:

Sostituzione della libreria <sys/mman.h> compatibile per Linux con una libreria equivalente compatibile con Windows riportata in seguito:

```
#ifdef _WIN32
    #include <windows.h>
#else
     #include <unistd.h>
#endif
#include <fcntl.h>
#include <sys/types.h>
```

sys/mman.h è una libreria che fornisce funzionalità come mmap, munmap, mprotect, ecc, per la gestione della memoria a basso livello.

Durante la compilazione con makefile, è stato riscontrato un errore per il comando wc, il quale consente di contare il numero di linee, parole e k-mer presenti all'interno del file FASTQ. Tale comando risulta essere compatibile con Linux, per questo motivo, anche in questo caso, è stata installata la libreria scoop, compatibile con Windows, eseguendo i seguenti comandi a catena:

```
    - # Installa scoop >> iwr -useb get.scoop.sh | iex
  - # Installa coreutils >> scoop install coreutils
```

Questa libreria contiene tutte le variabili d'ambiente di Linux, compreso il comando wc.

4 Risultati ottenuti

I risultati ottenuti fanno riferimento, in particolare, al file Balaenoptera.fastq di dimensioni 400.4 MB dal momento che rappresenta il dataset più grande. La dimensione della struttura dati Bloom Filter di KmerCo, ovvero countBF, dipende dal numero totale di K-mer presenti nel dataset di input. Sono stati effettuati alcuni esperimenti per osservare le prestazioni di KmerCo aumentando e riducendo il numero totale di K-mer. La lunghezza del contatore di countBF è di 8 bit.

La seguente tabella fornisce dettagli riguardanti il numero di K-mer considerati per la costruzione di countBF e la dimensione di countBF in megabyte. Tutti i KmerCo con diverse dimensioni di countBF, inseriscono lo stesso numero di K-mer e il rapporto tra K-mer inseriti e ignorati è zero. Pertanto, in questa sezione verrà effettuato il confronto tra vari KmerCo con diverse dimensioni (28-mers e 55-mers) di countBF in base al tempo di inserimento, al numero di inserimenti al secondo e al tasso di affidabilità. La scelta di questi due valori per k è stata effettuata principalmente per validare la robustezza e le prestazioni di KmerCo in diverse condizioni di input.

Table 1: Riepilogo del Conteggio K-mers e delle Dimensioni di CountBF per la versione canonica.

Total K-mers	**28 mers**	**55 mers**	**size CountBF (MB)**
total_kemrs/8	50055418	50055414	22.07
total_kmers/4	100110836	100110829	43.32
total_kmers/2	200221673	200221659	86.44
total_kmers	400443346	400443319	44.12

La tabella precedente è stata costruita sulla base dei risultati ottenuti facendo variare ad ogni esecuzione il numero totale dei k-mer. I risultati vengono salvati all'interno del file result.txt costruito nel seguente modo:

```
total_kmers/8

############## Results of dataset balaenoptera.fastq with write (Canonical) ##############
Kmer length: 28
Total kmers: 50055418
Total insertion: 50055418
Elapsed Time of insertion: 20.887000
Number of Insertion/Second: 2396486,714
Required memory size in bits and MB: 185173568 22.07 MB

Elapsed Time of query: 19.567000
Total queries: 25311149
Total number of distinct kmers: 22508105
Total number of erroneous kmers: 22982937
Total number of trustworthy kmers: 2328213

total_kmers/4

############## Results of dataset balaenoptera.fastq with write (Canonical) ##############
Kmer length: 28
Total kmers: 100110836
Total insertion: 100110836
Elapsed Time of insertion: 43.614000
Number of Insertion/Second: 2295383,042
Required memory size in bits and MB: 363433792 43.32 MB

Elapsed Time of query: 36.022000
Total queries: 48673524
Total number of distinct kmers: 42778800
Total number of erroneous kmers: 44970747
Total number of trustworthy kmers: 3702778
```

```
total_kmers/2

############ Results of dataset balaenoptera.fastq with write (Canonical) ############
Kmer length: 28
Total kmers: 200221673
Total insertion: 200221673
Elapsed Time of insertion: 79.237000
Number of Insertion/Second: 2526870,944
Required memory size in bits and MB: 725114048 86.44 MB

Elapsed Time of query: 76.909000
Total queries: 94363822
Total number of distinct kmers: 82050867
Total number of erroneous kmers: 89542273
Total number of trustworthy kmers: 4821550

total_kmers

############ Results of dataset balaenoptera.fastq with write (Canonical) ############
Kmer length: 28
Total kmers: 400443346
Total insertion: 400443346
Elapsed Time of insertion: 132.386000
Number of Insertion/Second: 3024816,416
Required memory size in bits and MB: 370168384 44.12 MB

Elapsed Time of query: 64.876000
Total queries: 82578904
Total number of distinct kmers: 71774552
Total number of erroneous kmers: 66086027
Total number of trustworthy kmers: 16492878
```

Il file precedente prende in input la lunghezza del k-mer, ovvero una sottosequenza nucleotidica di una molecola di DNA/RNA e calcola il numero totale dei k-mer che è dato da: $total\_kmer = file\_length - kmer\_len$. Per quanto riguarda $total\_insertion$, esso rappresenta il numero totale di kmer che sono stati inseriti nel set di dati, mentre $Elapsed\_time\_of\_Insertion$ rappresenta la quantità di tempo, misurata in secondi, impiegata dal software per completare l'inserimento di tutti i k-mer contati in $total\_insertion$. $Number\_of\_Insertion/Second$ è dato dal rapporto tra $total\_insertion$ e $Elapsed\_time\_of\_Insertion$. Alla fine dell'esecuzione del codice, viene fatta una distinzione tra i kmer che sono stati classificati come kmer reali e affidabili e salvati nel file trustworthy.txt e i kmer che sono stati classificati come kmer che contengono rumore e salvati del file Erroneous.txt.

In relazione all'output ottenuto, è necessario fare una distinzione tra la modalità "without canonical" e la modalità "with canonical". La differenza sostanziale tra le due modalità riguarda il modo con cui vengono trattati i k-mer e il loro reverse complement che influenzano

l'accuratezza biologica e l'efficienza computazionale.

- Nella modalità "without canonical", i K-mer e i rispettivi revese complement vengono trattati come entità separate, risultando errata per l'analisi del DNA; in questa modalità viene effettuato soltanto il conteggio dei k-mer senza effettuare il confronto con i rispettivi reverse complement, di conseguenza il costo computazionale risulta piuttosto basso.
- Nella modalità "with canonical", i k-mer e i rispettivi reverse complement vengono considerati come un'unica entità, di conseguenza, a differenza della modalità precedente, il costo computazionale risulterà alto, dal momento che vengono gestiti i confronti con il reverse complement e il doppio hashing e, di conseguenza, anche il tempo di inserimento dei k-mer nella matrice dei Bloom-Filter risulterà alto.

In seguito ai risultati ottenuti dal file precedente, sono stati costruiti i seguenti grafici:

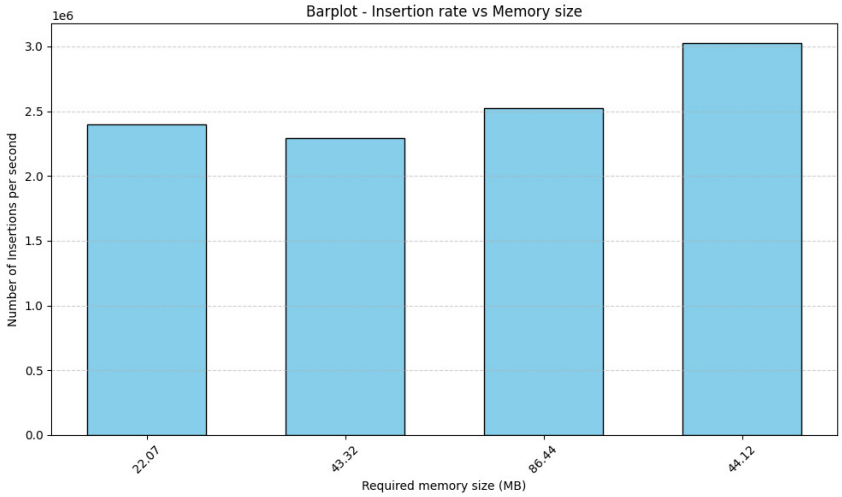

Figure 1: Insertion Rate vs Memory Size (MB).

Il grafico precedente mostra il tasso di inserimento nella modalità "with canonical". Esso misura quanti K-mer il sistema riesce ad elaborare al secondo. Il tasso è inversamente proporzionale al tempo di inserimento. Il picco di performance è per 44.12 MB che corrisponde al punto in cui l'algoritmo elabora il maggior numero di K-mer al secondo, e i tassi più bassi per le memorie più grandi (43.32 MB e 86.44 MB). I tassi di inserimento sono visibilmente inferiori (circa il 40-50% in meno) rispetto alla modalità "without canonical" ($2.3 - 3.0 \times 10^6$ vs $4.0 - 4.4 \times 10^6$).

Il tasso di inserimento inferiore è la diretta conseguenza del maggiore tempo impiegato: il lavoro computazionale aggiuntivo riduce il numero di K-mer che il sistema può processare al secondo, giustificando la differenza tra le due modalità. Nonostante l'inferiore performance in termini di velocità, la modalità "with canonical" è preferita nell'analisi genomica, in quanto riduce a metà il numero di K-mer distinti da memorizzare, portando a un minor tasso di falsi positivi e a una maggiore accuratezza del conteggio biologico (come evidenziato

anche nel supplemento che raccomanda la versione con countBF da 70.92 MB e 8 bit per il compromesso ottimale tra performance, memoria e accuratezza).

Il seguente grafico mostra il tempo di inserimento quando l'algoritmo deve calcolare il K-mer canonico ("with canonical"). E' possibile osservare una tendenza all'aumento del tempo con l'aumento della memoria allocata. Tutti i tempi di inserimento in questa modalità sono significativamente più alti, con un picco a 128 s (il picco del "without canonical" era a 91 s), questo perchè, questa modalità, introduce una maggiore complessità computazionale per ogni K-mer inserito, che include:

- Calcolo della sequenza reverse complement.
- Calcolo dell'hash della reverse complement.
- Confronto tra l'hash del K-mer originale e l'hash della reverse complement per selezionare la K-mer canonica.

Queste operazioni extra aumentano il tempo di elaborazione per ogni singolo K-mer, aumentando il tempo totale di inserimento.

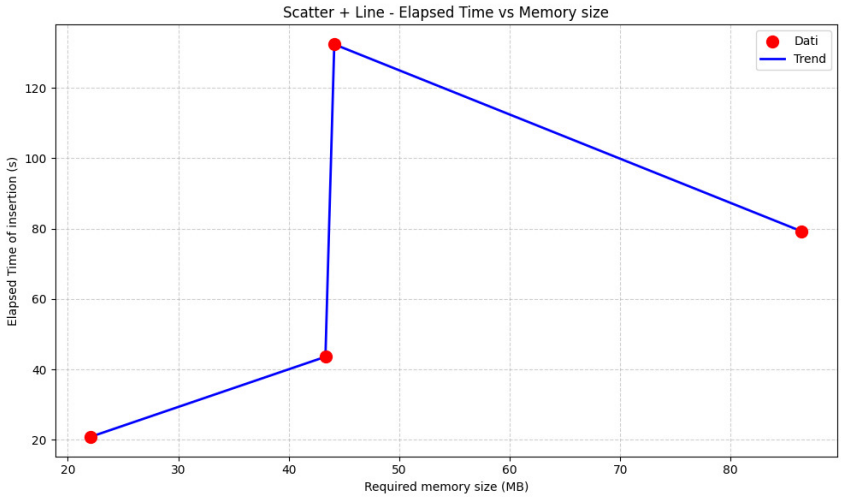

Figure 2: Elapsed Time vs Memory Size (MB).

Nel grafico seguente, il tasso di inserimento è inversamente proporzionale al tempo. Il tasso di inserimento più alto è $\mathbf{1.51 \times 10^6}$ ins/s per 44.13 MB(un dato che, per la relazione inversa con il tempo, è inaspettato se non si considera che, il numero di Total Insertions non è costante per tutte le allocazioni in questo test, o se il picco di 266 s è errato e dovrebbe corrispondere al tasso più basso). La performance migliore in termini di velocità si osserva per 22.07 MB ($\approx 1.25 \times 10^6$ ins/s). La modalità canonical per i 55-mers è molto più lenta (max $\approx 1.5 \times 10^6$ ins/s) rispetto ai 28-mers (max $\approx 3.0 \times 10^6$ ins/s).

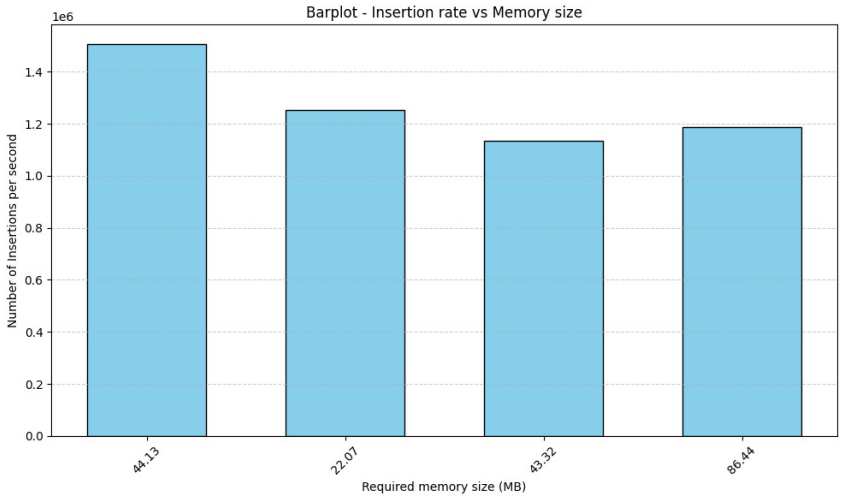

Figure 3: Insertion Rate vs Memory Size (MB).

Nel grafico seguente è possibile notare un aumento del tempo con la memoria, con il tempo più basso per 22.07 MB ($\approx$ 40 s). Il picco a 44.13 MB ($\approx$ 266 s) è il più alto tra tutti i grafici, indicando un'estrema lentezza in quel punto. La penalità di tempo per l'uso del canonical è più severa per i 55-mers. Il tempo di inserimento aumenta drasticamente in confronto alla modalità "without canonical" per i 55-mers ($\approx$ 40 s vs $\approx$ 18 s), dal momento che il calcolo del reverse complement e il confronto degli hash sono operazioni più costose per i K-mers più lunghi, che amplificano il sovraccarico computazionale già presente nel K=28, causando un maggiore rallentamento.

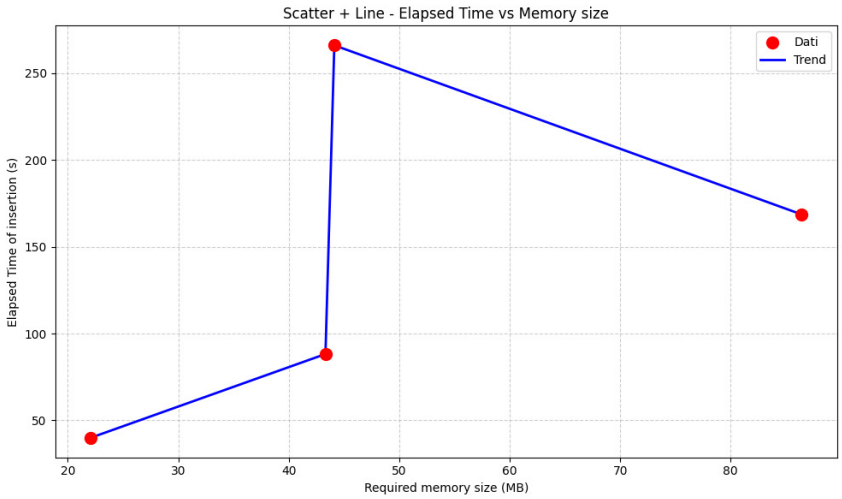

Figure 4: Insertion Rate vs Memory Size (MB).

Table 2: Riepilogo del Conteggio K-mers e delle Dimensioni di CountBF per la versione non canonica.

Total K-mers	**28 mers**	**55 mers**	**size CountBF (MB)**
total_kmers/8	50055418	50055414	22.07
total_kmers/4	100110836	100110829	43.32
total_kmers/2	200221673	200221659	86.44
total_kmers	400443346	400443319	44.12

Il grafico seguente illustra l'efficienza di inserimento in funzione della memoria nella modalità "without canonical". Il tasso di inserimento è inversamente correlato al tempo di inserimento. Il picco di performance si osserva con la memoria più piccola (22.07 MB) con il tasso più alto, mentre i tassi più bassi corrispondono alle memorie più grandi (43.32 MB e 86.44 MB).

Il tasso di inserimento è Total Insertions/Elapsed Time. Poiché il numero totale di inserzioni è costante, il barplot è un'immagine speculare dello scatterplot del tempo: meno tempo è impiegato, maggiore è il tasso di inserimento/s. La modalità "without canonical" è la più veloce e con il tasso di inserimento più alto in generale, perché omette il costoso calcolo del reverse complement e il confronto tra gli hash, rendendo ogni singola operazione di inserimento più rapida.

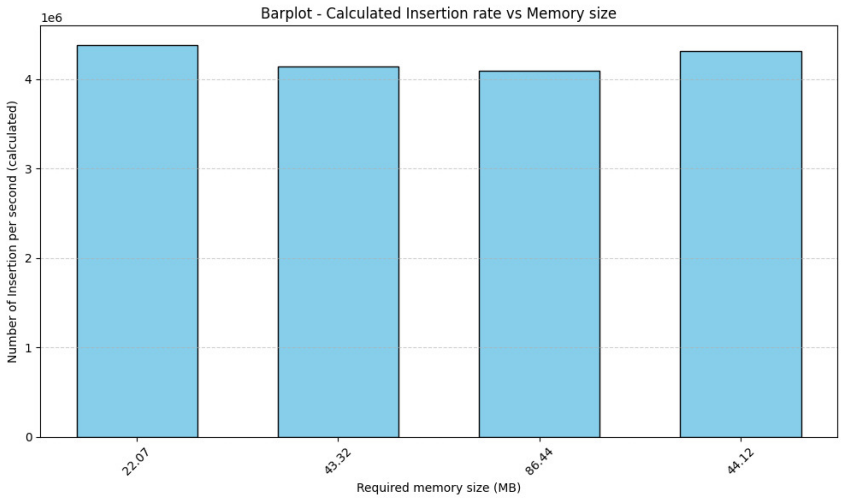

Figure 5: Insertion Rate vs Memory Size (MB).

Il seguente grafico mostra la relazione tra la memoria allocata e il tempo impiegato per inserire i K-mer senza considerare il reverse complement e il confronto degli hash. E' possibile notare che il tempo di inserimento tende ad aumentare con la dimensione della memoria allocata, con l'eccezione di un picco inatteso a 44.12 MB. In questa modalità (without canonical), l'operazione di base è più semplice. In generale, con l'aumentare della dimensione del Bloom Filter (countBF), i tempi di accesso alla memoria (lettura/scrittura) tendono a salire leggermente, aumentando il tempo totale. Il punto più veloce è 22.07 MB.

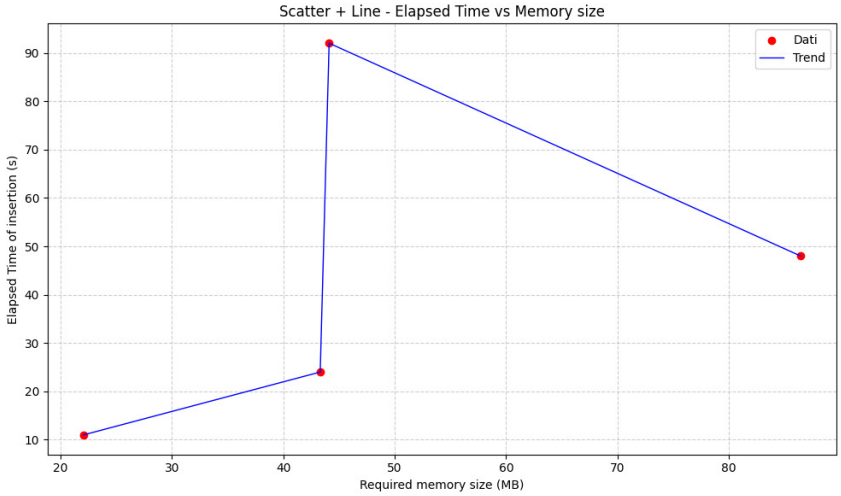

Figure 6: Elapsed Time vs Memory Size (MB).

Nel seguente grafico invece, il tasso di inserimento più alto è per la memoria 22.07 MB ($\approx 2.7 \times 10^6$ ins/s), corrispondente al tempo più basso. Il tasso più alto è 2.7×10^6 ins/s per 22.07 MB. I 55-mers hanno un tasso di inserimento massimo inferiore (2.7×10^6 ins/s) rispetto ai 28-mers (4.4×10^6 ins/s). Questa differenza conferma che, a parità di modalità (senza il costo canonical), l'elaborazione di K-mers più lunghi richiedono più cicli CPU per operazione, riducendo la velocità massima di inserimento.

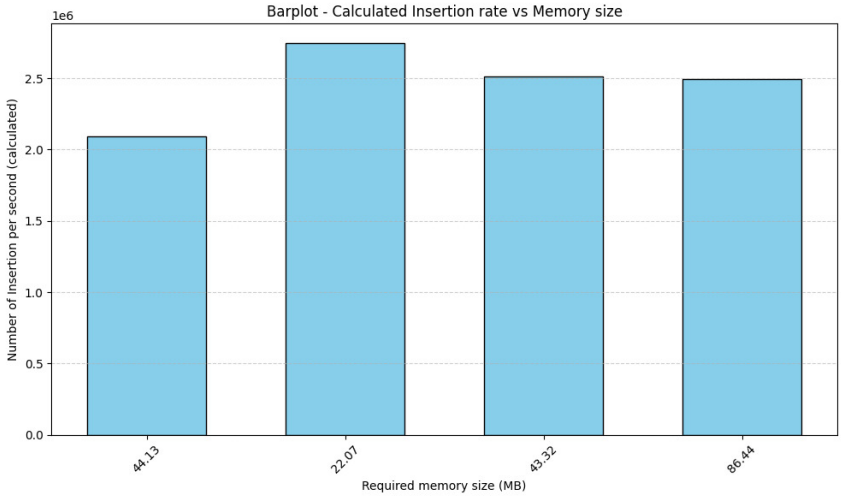

Figure 7: Insertion Rate vs Memory Size (MB).

Similmente al grafico precedente con 28-mers, il tempo di inserimento tende ad aumentare con la dimensione della memoria allocata, con un picco anomalo a 44.13 MB (≈ 187 s). Minore è la memoria utilizzata, minore sarà il tempo impiegato: 22.07 MB (≈ 18 s). I 55-mers risultano più lenti rispetto ad altri. Ad esempio, per 22.07 MB, i 55-mers impiegano ≈ 18 s, mentre i 28-mers impiegano ≈ 11 s, questo perchè l'utilizzo di K-mers più grandi, comporta

un maggiore lavoro computazionale all'interno della funzione hash e un maggiore sovraccarico per le operazioni di manipolazione delle stringhe per ogni inserimento, risultando in un tempo trascorso più lungo.

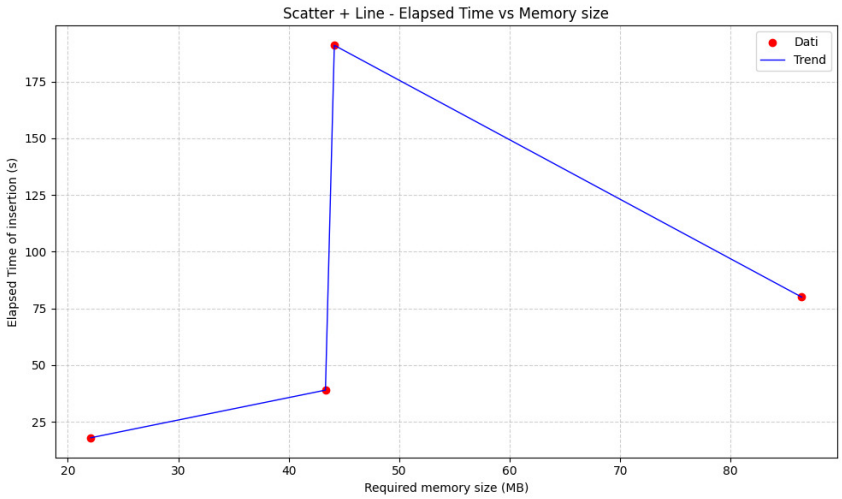

Figure 8: Elapsed Time vs Memory Size (MB).

5 Conclusioni

L'analisi delle performance di KmerCo su 28-mers e 55-mers per il dataset balaenoptera, evidenzia un fondamentale compromesso tra accuratezza biologica ed efficienza computazionale. I risultati mostrano che la velocità di inserimento è dominata da due fattori: la lunghezza del K-mer (K) e la gestione della canonizzazione.

In generale, l'aumento della lunghezza del K-mer da K=28 a K=55 impone una significativa penalità di tempo e un calo del tasso di inserimento in entrambe le modalità. Questo perché i K-mer più lunghi richiedono inevitabilmente più lavoro computazionale per l'hashing e la manipolazione delle stringhe ad ogni singola operazione, riducendo la velocità massima di processamento (ad esempio, il tasso di inserimento in modalità "without canonical" per K=55 è notevolmente inferiore rispetto a K=28). Di conseguenza, l'efficienza di KmerCo è inversamente correlata alla lunghezza K.

Per quanto riguarda invece la modalità "with canonical", essa è cruciale per l'analisi genomica, poiché tratta i K-mer e i loro reverse complement come un'unica entità biologica. Tuttavia, questa correttezza biologica ha un costo:

- Il tempo di inserimento aumenta drasticamente (soprattutto per i 55-mers), a causa delle operazioni aggiuntive necessarie per calcolare il reverse complement e confrontare gli hash per stabilire la forma canonica.
- Il tasso di inserimento si dimezza rispetto alla modalità "without canonical".

6 Link di Riferimento

1. Documenti di riferimento per la stesura della documentazione: KmerCo2023.
2. Repository GitHub di riferimento per il lavoro svolto: KmerCo-Main