K-finger

Candidati:

Alfonso Luciani	0522500929
Antonio Allocca	0522501527

Professore:
Prof. Giuseppe Cattaneo

Implementazione in Scala/Spark

Table of contents

1 **Introduzione**
Introduzione al calcolo degli overlap

2 **K-Finger**
Cos'è e necessità di un'implementazione parallela

3 **Risultati ottenuti**
Risultati ottenuti e performance

4 **Conclusioni**
Considerazioni finali

01 Introduzione

Introduzione al calcolo degli overlap

Introduzione al calcolo degli overlap: allineamento

Motivazioni:

1. Per studiare aspetti evolutivi, somiglianze tra specie
2. Per capire cambiamenti nucleotidici o di riarrangiamento
3. Vaccini, antibiotici

Se abbiamo più di due sequenze genomiche e vogliamo confrontarle dobbiamo ricorrere all'allineamento.

Introduzione al calcolo degli overlap

Gli overlap si riferiscono alla sovrapposizione di sequenze di DNA o geni su uno stesso segmento o su segmenti adiacenti del genoma.
Gli overlap nel contesto della genomica possono presentare sfide nell'interpretazione dei dati e nella predizione delle funzioni genomiche.

Le tecniche di sequenziamento e le analisi bioinformatiche sono importanti per identificare e caratterizzare gli overlap, nonché per studiarne le implicazioni funzionali.

La comprensione degli overlap genomici contribuisce alla nostra conoscenza dei meccanismi regolatori dell'espressione genica e alla comprensione delle complesse reti di interazione all'interno del genoma

Introduzione al calcolo degli overlap

Gli overlap si riferiscono alla sovrapposizione di sequenze di DNA o geni su uno stesso segmento o su segmenti adiacenti del genoma.

Gli overlap nel contesto della genomica possono presentare sfide nell'interpretazione dei dati e nella predizione delle funzioni genomiche.

Le tecniche di sequenziamento e le analisi bioinformatiche sono importanti per identificare e caratterizzare gli overlap, nonché per studiarne le implicazioni funzionali.

La comprensione degli overlap genomici contribuisce alla nostra conoscenza dei meccanismi regolatori dell'espressione genica e alla comprensione delle complesse reti di interazione all'interno del genoma

02

K-Finger

Cos'è e necessità di un'implementazione parallela

Idea proposta*

Approccio **alignment-free** per scoprire gli overlap in un grande insieme di **read** (le quali possono contenere errori legati alla lettura) utilizzando una rappresentazione delle read data dalla lunghezza delle **parole di Lyndon** che compongono la read.

E' stato infatti dimostrato che due stringhe che condividono un overlap comune, condividono anche un insieme di fattori consecutivi nella loro fattorizzazione.

*Bonizzoni, P., Petescia, A., Pirola, Y., Rizzi, R., Zaccagnino, R., Zizza, R.:
KFinger: Capturing Overlaps Between Long Reads by Using Lyndon Fingerprints. Bioinformatics and Biomedical Engineering - 9th International Work-Conference, IWBBIO 2022, Spain, June 27-30, 2022, Lecture Notes in Computer Science, vol. 13347, pp. 436-449.

K-Finger: come funziona

Kfinger è un **algoritmo** utilizzato per il calcolo degli overlap tra **sequenze genomiche**.

È progettato per affrontare le sfide **dell'analisi genomica su larga scala**, in particolare la ricerca di **Sovrapposizioni** (**overlap**) significative tra frammenti di sequenze che possono indicare relazioni evolutive, regioni funzionali o errori di sequenziamento.

L'algoritmo Kfinger utilizza una tecnica basata sui k-mer, che sono sottosequenze di lunghezza fissa prese dalle sequenze genomiche.

Un **k-mer** è una parola di lunghezza k composta da **nucleotidi** (A, C, G, T nel caso del DNA).

K-Finger: come funziona

L'idea fondamentale di **Kfinger** è quella di generare tanti indici basati sui **k-mer** per accelerare la ricerca degli **overlap**.

Per creare gli indici, Kfinger suddivide le sequenze genomiche in **frammenti (READ)** di lunghezza prefissata e genera gli insiemi di k-mer per ogni frammento.

Durante la fase di ricerca degli overlap, Kfinger confronta gli insiemi di k-mer dei frammenti in modo da identificare i k-mer in comune e determinare se esiste un'**overlap significativa** tra i frammenti.

Gli indici vengono utilizzati per avere il riferimento di quel k-mero della nostra READ

K-finger con Spark: Obiettivi

Tuttavia, l'analisi delle sequenze genomiche presenta **sfide uniche** e **complesse**, soprattutto a causa delle dimensioni sempre crescenti dei **dataset genomici**.
Le tecnologie di sequenziamento ad **alta velocità** hanno reso possibile la generazione di enormi quantità di dati genomici, che richiedono **potenti infrastrutture di calcolo** per **l'elaborazione** e **l'analisi**.

La combinazione di **Kfinger** con **Apache Spark** offre un approccio innovativo per affrontare questa sfida.

K-finger con Spark: Obiettivi

Kfinger (che vedremo nel dettaglio in seguito), grazie al suo potere di identificare e caratterizzare gli **overlap** nelle **sequenze genomiche**, si rivela uno strumento efficace per l'analisi dei dati genomici.
L'utilizzo di Apache Spark, con la sua capacità di **calcolo distribuito** e **scalabile**, consente di gestire efficacemente **grandi volumi di dati genomici**, riducendo i tempi di esecuzione e migliorando **l'efficienza** complessiva dell'analisi.
Questo approccio è di fondamentale importanza nell'ambito della genomica, poiché fornisce **strumenti avanzati** per comprendere la struttura e la funzione del genoma, facilitando così la scoperta di informazioni rilevanti per la ricerca biomedica e la pratica clinica.

02

K-Finger

Spiegazione algoritmo e implementazione in Scala/Spark

Spiegazione algoritmo

Spiegazione non legata all'implementazione

Parametri da impostare

- La **dimensione** e la **minima lunghezza** totale dei **k-mers** devono essere stabiliti all'inizio dell'algoritmo.
- **Minimo numero di kmers** (unici) che due reads in overlap devono condividere (in assoluto)
- **Massimo numero di occorrenze di un kmer** nei reads (valore compatibile con la coverage dell'input).
  - Nell'algoritmo sviluppato questo parametro **non** viene valutato.
- Parametri di filtraggio delle regioni comuni e degli overlap in output.

Parametri da impostare

- k = 7
- min_total_length = 40
- min_shared_kmers = 4
- max_kmer_occurrence = -1
- max_diff_region_percentage = 0.0
- min_region_length = 100
- min_region_kmer_coverage = 0.27
- min_overlap_coverage = 0.70
- min_overlap_length = 600

Prima fase: Lettura delle fingerprint

Le **fingerprint** sono salvate su singola riga e gli eventuali frammenti di fingerprint sono separati dal carattere «**|**» (split delle fingerprint sull'intera read)

La prima stringa di una fingerprint ha il formato **ID_BOOL**.

- **BOOL** assumerà valore 1 se la read è quella **originale** e valore 0 se è la sua versione **Reverse and Complement**.
- **ID** è l'identificativo della read (che sarà duplicata)

Seconda fase: calcolo kmer_occurrences

kmer_occurrences è un dizionario che contiene tutti i kmers aventi almeno la minima lunghezza totale.

I kmer che sono presenti il tale dizionario rappresentano la chiave (**KEY**) e il loro valore (**VALUE**) è una lista di coppie (**READ**,**START**)

Lista di coppie (**READ**,**START**)

- **READ**: **ID** della read in cui è stato letto il kmer
- **START**: **indice** del primo valore del kmer all'interno della fingerprint

Terza fase: Dizionari dei leftmost e rightmost kmers comuni

In questa fase vengono memorizzati per ogni coppia di read il **leftmost** e **rightmost kmer** comune e il numero di kmers condivisi.

Dizionario per memorizzare rightmost e leftmost

- **KEY**: (read1, read2)
- **VALUE**: [start1, start2]

Per ogni entry del dizionario, per ogni coppia (**READ**,**START**) presenti all'interno della lista della entry (da cui prendiamo **read1** e **start1**) e per ogni coppia successiva (da cui viene presa **read2**) verifichiamo se nella entry del dizionario con chiave (**read1**,**start2**) il leftmost precedente è più grande si **start1** e se il rightmost è più piccolo di **start1**.

Quarta fase:
Calcolo degli overlap

Il dizionario dei leftmost e rightmost k-mers contiene tutte le **coppie** di **fingerprints** (chiavi del dizionario) che sono candidati a dare un **overlap**.

Per ogni coppia (**r1**, **r2**) nel dizionario si ottiene il leftmost k-mer **L** e il right-most k-mer **R**.

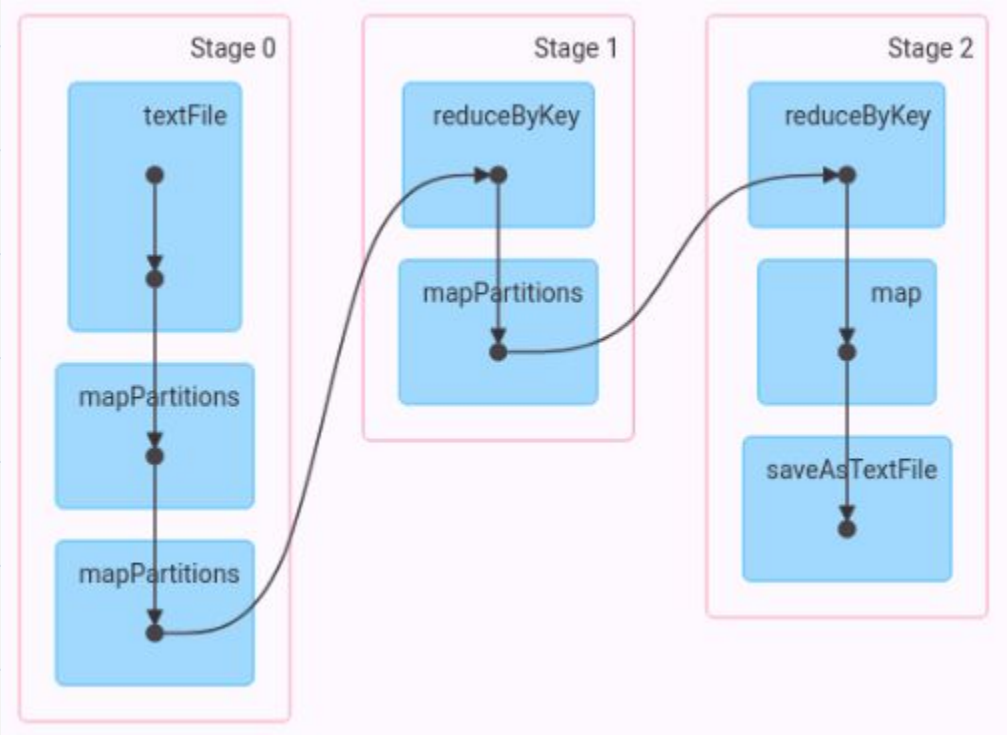
Vengono poi fatte le due seguenti verifiche:

1. **r1** e **r2** devono condividere almeno **min-shared-kmers** (parametro in input) k-mers che sono unici nelle due reads.
2. **L** deve venire prima di **R** in **r2** (per costruzione **L** viene prima di **R** in **r1**).

Implementazione in Scala/Spark

Spiegazione e dettagli implementativi

Stage dell'algoritmo



Stage 0

L'algoritmo comincia con la lettura delle **Fingerprint**. Tali fingerprint possono essere letter dallo **SparkContext**, in particolare utilizzando **sc.textFile(inputPath)** che ci permette di leggere un file dall'**HDFS** (Dove avremo caricato in precedenza il nostro file di input) e ritornerà un **RDD di stringhe**.

Stage 0

Ora che abbiamo **l'RDD** con tutte le stringhe (**Fingerprint**) possiamo procedere ad elaborarle.
Ogni singola riga verrà elaborata in modo **indipendente** dalle altre, quindi andremo a **parallelizzare** l'elaborazione tramite la funzione **map**.

```
.map(line => getreads(line))
```

Stage 0

La funzione **getreads** prende in input una stringa, più in particolare le singole **Fingerprint**, e restituisce una lista di **triple di interi**.

Prima la fingerprint viene **splittata** in presenza del carattere **spazio** " " e successivamente vengono rimosse tutte le occorrenze del carattere "|"

Stage 0

Successivamente **il primo elemento di ogni fingerprint** come descritto anche precedentemente sarà una stringa del tipo **ID_BOOL** e quindi andiamo ad estrapolare tali dati e li andiamo ad immagazzinare in un coppia (**ID**,**BOOL**). In seguito ogni stringa presente **dal secondo elemento** della stringa in poi sarà un **intero** della fingerprint.

Ogni tripla presente nell'array di triple risultante sarà composta come (**ID**,**READ**,**INTERO**). In Scala è presente la **lazy evaluation** quindi **non possiamo conoscere a priori quale read viene elaborata** e di conseguenza non possiamo stabilire a priori quanto vale il valore **READ**. Per determinare tale valore quindi si è pensato di arginare tale problema ricavandolo in **modo deterministico** e con **complessità lineare** tramite un'equazione che usa come variabili il valore ID ed il valore di BOOL ovvero:

```
ID.toInt * 2 + ¬BOOL
```

Stage 0

Una volta che la lettura delle fingerprint è stata completata ci resta **elaborare le triple così ottenute**. In questo punto dell'algoritmo **l'RDD** è un insieme di Array che contengono una sequenza ordinata* di triple.

**Per ordinamento si intende che le triple sono posizionate all'interno dell'array nell'ordine in cui i rispettivi interi all'interno delle fingerprint sono stati letti.*

A questo punto ci interessa elaborare i **k-mer** e per fare ciò ci servirà elaborare gli array precedentemente ottenuti. Anche in questo caso gli array possono essere letti in modo completamente indipendente l'uno dall'altro e questa volta useremo una **flatmap** per ottenere il nuovo RDD. La **flatmap** ci permette di eseguire prima una **map** e poi ottenere singolarmente tutti i k-mer risultando poi in un **unico RDD** che contiene al suo interno **tutti i k-mer ottenuti**.
Ovviamente anche questa volta siccome utilizziamo una flatmap su un RDD l'algoritmo sarà **implicitamente parallelo**.

Stage 0

Per **ottenere i k-mer** è stata utilizzata una funzione che prende il nome di **getkmers** che prende in input una lista di triple di interi ovvero un singolo array contenuto all'interno dell' RDD e restituisce un insieme di coppie (**KEY**,**VALUE**) dove la **chiave** è il **k-mer** e il **valore** è una **lista di coppie** (**READ**,**START**) che sta a rappresentare in quale read e in quale indice appare il k-mer.

Il valore è rappresentato come una **lista di coppie** perché uno stesso k-mer può apparire sia in diversi punti della stessa read che anche in read diverse tra loro.

L'algoritmo scorre una ad una le triple presenti all'interno dell'array. Per ogni tripla, viene in primis creata una chiave che è una stringa che contiene il campo INTERO della tripla e quello delle k – 1 triple successive separate da un underscore "_".

Stage 0

Una volta lette le **k triple** si fa la **somma del loro campo INTERO** e si verica che il valore così ottenuto sia **≥ min_total_length**.

Se questa condizione dovesse verificarsi allora sarà aggiunta all'interno della lista di coppie risultate una coppia del tipo (**KEY**,(**READ**,**START**)).

Da tenere in considerazione è che anche se in questa fase abbiamo delle coppie che hanno per chiave la stessa chiave, esse avranno comunque due entry differenti con una sola coppia all'interno del campo VALUE.

Il motivo sarà spiegato in seguito, ma è intuibile che sia fatto per accorpare tutti gli elementi con la stessa chiave tramite una **reduceByKey** ottenendo altro parallelismo implicito. I campi (**READ**,**START**) sono ottenuti dai primi due valori della tripla che in quel momento stiamo leggendo (Il primo vlore del k-mer).

Stage 0 - Esempio

1_0 1 2 3 4 5 6 7 8 9

↓

(1,1,1)(1,1,2)(1,1,3)(1,1,4)(1,1,5)(1,1,6)(1,1,7)(1,1,8)(1,1,9)

↓

(1_2_3_4,(1,1)),(2_3_4_5,(1,1)),(3_4_5_6,(1,1)),(4_5_6_7,(1,1)),
(5_6_7_8,(1,1)),(6_7_8_9,(1,1))

Stage 1

Siccome abbiamo utilizzato una flatmap l'RDD sarà un insieme di coppie (**KEY**,**VALUE**). Prima di proseguire con l' elaborazione dobbiamo accorpare tutti i risultati precedentemente ottenuti. Precedentemente abbiamo ottenuto una lista di coppie (KEY,VALUE) quindi possiamo andare a **ridurre l'RDD** tramite una **ricerca per chiave**. Qui ci vengono in aiuto 2 cose fondamentali:

1. **reduceByKey**
2. **Le classi mutabili**

reduceByKey ci permette di accorpare tutte le coppie (**READ**,**START**) all'interno del campo **VALUE** dei **k-mer** che hanno la stessa chiave, mentre le classi mutabili ci permettono di poter alterare quest'ultimo valore senza avere la necessità di dover **ricostruire la lista di coppie** ogni volta.
La concatenazione di liste è espressa con l'operatore **++** di Scala.

```
.reduceByKey(_++_)
```

Stage 1

Ora l'RDD è un insieme di coppie (**KEY**,**VALUE**) dove all'interno del campo value sono immagazzinate correttamente le "**coordinate**" dove appaiono i **k-mer**.
Ora seguendo la descrizione dell'algoritmo dobbiamo procedere con la creazione del dizionario **matches_dict**.

```
.flatMap( x => getmatchesdict(x))
```

Viene utilizzata nuovamente la **flatmap** per ottenere parallelismo implicito sull'RDD e per poter operare sui dati così ottenuti.
Per questa fase è stata scritta una funzione che prende il nome di **getmatchesdict**.
Tale funzione prende in **input** una coppia (**KEY**,**VALUE**) presente all'interno dell'RDD e restituisce **un'array** di valori (**KEY**,**VALUE**) dove la **chiave** è la concatenazione di **due READ** separate da un underscore "_" e **value** è un array di 5 valori interi che rappresentano il **leftmost**, il **rightmost** e il numero di volte in cui questa chiave appare.

Stage 1

Seguendo la logica di prima, anche se due entry in questa fase hanno due chiavi uguali, ci saranno comunque due entry diverse per poi andare a raggruppare nuovamente per chiave.
Per ogni entry dell'RDD si prende la parte **VALUE**. All'interno di questo campo è contenuto una lista di coppie (**READ**,**START**) che andrà analizzata.
Il leftmost ed il rightmost sono entrambi coppie di valori interi (**READ**,**START**) che stanno ad indicare la regione che due read hanno in comune.
Vengono analizzate tutte le possibili coppie (**READ1**,**READ2**) ottenibili dalla lista di coppie presenti nell'RDD e per ogni coppia viene creata una entry che avrà una struttura del tipo
`(READ1_READ2,(READ1,START1,READ2,START2,1)).`
L'1 iniziale servirà a contare quante volte la chiave READ1_READ2 è presente nell'RDD risultante.

Stage 2

Ed ora non ci resta altro che usare un'altra **reduceByKey** sempre parallelizzata perché è eseguita sull'RDD che servirà in primis a stabilire a reduce conclusa quali sono i **leftmost** e i **rightmost per ogni coppia di read ottenuta**, inoltre ci sarà dato anche il numero di volte in cui queste coppie vengono contate all'interno dell'RDD.

Il numero di volte in cui queste coppie compaiono all'interno dell'RDD deve essere **≤ min_shared_kmers** in modo tale da essere preso in considerazione, altrimenti le due read condividono troppi pochi k-mer per essere analizzato (Molto probabilmente non è un **Overlap**).

Stage 2

Infine viene effettuato un filtraggio di tutte le coppie che hanno un numero di k-mer condivisi **≤ min_shared_kmers** ma l'operazione **filter** essendo eseguita su un RDD è una delle operazioni implicitamente parallelizzate.

Dopo il filtraggio tramite una map anch'essa parallelizzata per tutto l'RDD possiamo **convertire gli Array** ottenuti **in Tuple** solo per avere una **visualizzazione** più limpida dell'output anche perché quei valori non devono essere più modificati.

Si procede poi con il **salvataggio dell'output sull'HDFS** e la chiusura del contesto di Spark segnando così la **fine** dell'esecuzione dell'algoritmo.

03

Risultati ottenuti

Risultati ottenuti e performance

Risultati ottenuti

Come c'era da aspettarselo, la parallelizzazione di tale algoritmo ha portato a una **migliore scalabilità** del calcolo a fronte di **input sempre e sempre più grandi**.

L'esecuzione locale, però in alcune parti dell'algoritmo (Come ad esempio la parte finale) risulta essere più efficiente in quanto il dataset ottenuto dalla creazione dei dizionari porta ad avere un **piccolo dizionario** degli overlap facilmente computabile da una macchina in locale.

Il problema della **scalabilità** nasce appunto dalla seconda parte dell' algoritmo (Pienamente implementata) in quanto l'algoritmo presenta una **complessità** di **tempo** e di **spazio O(n!)** ed inevitabilmente, per quanto la singola macchina possa essere potente, non basta.
Verrà riportata di seguito la taglia dei dizionari con l'esecuzione su un file di input di taglia 2GB

Risultati ottenuti

Una tale mole di dati che diventa sempre più grande (in modo fattoriale appunto) non può e non portà essere gestito in locale.
Sono proprio i problemi come questo a far nascere la necessità di avere un sistema distribuito in grado di elaborare e conservare questi dataset.

I risultati riportati sopra sono i data raw, ovvero senza averci applicato sopra alcun tipo di filtro, ma se invece aggiungiamo un filtro che in questo caso va a scandire quali sono i dati che effettivamente sono necessari al proseguimento dell'algoritmo noteremo che il dataset siriduce di molto.

```
.filter((tupla) => tupla._2(4) >= min_shared_kmers)
```

```
allocca@masterunisa:~$ hdfs dfs -ls -h outputrnd_input2g
Found 27 items
-rw-r--r--   1 allocca supergroup          0 2023-05-19 12:51 outputrnd_input2g/_SUCCESS
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:51 outputrnd_input2g/part-00000
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:50 outputrnd_input2g/part-00001
-rw-r--r--   1 allocca supergroup    865.0 M 2023-05-19 12:49 outputrnd_input2g/part-00002
-rw-r--r--   1 allocca supergroup    865.0 M 2023-05-19 12:49 outputrnd_input2g/part-00003
-rw-r--r--   1 allocca supergroup    864.9 M 2023-05-19 12:47 outputrnd_input2g/part-00004
-rw-r--r--   1 allocca supergroup    865.1 M 2023-05-19 12:49 outputrnd_input2g/part-00005
-rw-r--r--   1 allocca supergroup    864.9 M 2023-05-19 12:50 outputrnd_input2g/part-00006
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:49 outputrnd_input2g/part-00007
-rw-r--r--   1 allocca supergroup    865.0 M 2023-05-19 12:50 outputrnd_input2g/part-00008
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:50 outputrnd_input2g/part-00009
-rw-r--r--   1 allocca supergroup    864.8 M 2023-05-19 12:49 outputrnd_input2g/part-00010
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:49 outputrnd_input2g/part-00011
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:47 outputrnd_input2g/part-00012
-rw-r--r--   1 allocca supergroup    865.0 M 2023-05-19 12:49 outputrnd_input2g/part-00013
-rw-r--r--   1 allocca supergroup    865.2 M 2023-05-19 12:50 outputrnd_input2g/part-00014
-rw-r--r--   1 allocca supergroup    865.1 M 2023-05-19 12:49 outputrnd_input2g/part-00015
-rw-r--r--   1 allocca supergroup    864.8 M 2023-05-19 12:51 outputrnd_input2g/part-00016
-rw-r--r--   1 allocca supergroup    865.3 M 2023-05-19 12:50 outputrnd_input2g/part-00017
-rw-r--r--   1 allocca supergroup    864.9 M 2023-05-19 12:49 outputrnd_input2g/part-00018
-rw-r--r--   1 allocca supergroup    864.8 M 2023-05-19 12:49 outputrnd_input2g/part-00019
-rw-r--r--   1 allocca supergroup    864.8 M 2023-05-19 12:47 outputrnd_input2g/part-00020
-rw-r--r--   1 allocca supergroup    864.8 M 2023-05-19 12:49 outputrnd_input2g/part-00021
-rw-r--r--   1 allocca supergroup    864.9 M 2023-05-19 12:49 outputrnd_input2g/part-00022
-rw-r--r--   1 allocca supergroup    864.6 M 2023-05-19 12:49 outputrnd_input2g/part-00023
-rw-r--r--   1 allocca supergroup    864.7 M 2023-05-19 12:51 outputrnd_input2g/part-00024
-rw-r--r--   1 allocca supergroup    864.5 M 2023-05-19 12:50 outputrnd_input2g/part-00025
```

Figura: Taglia dizionari con input 2 GB senza filtro

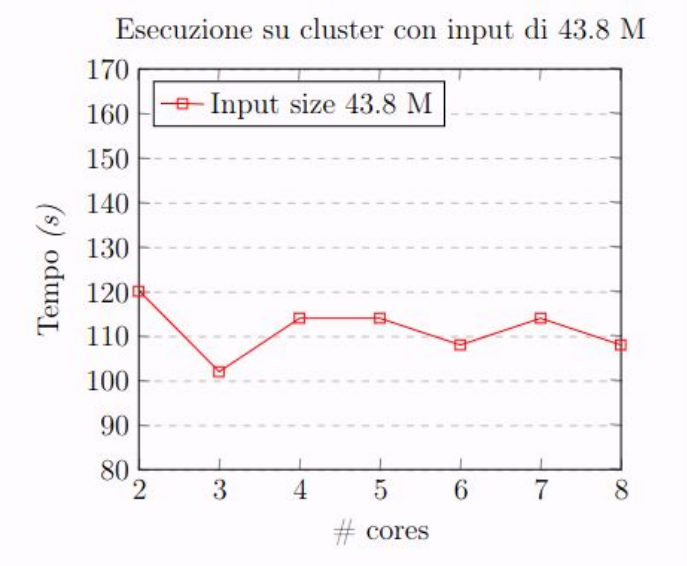
Risultati ottenuti

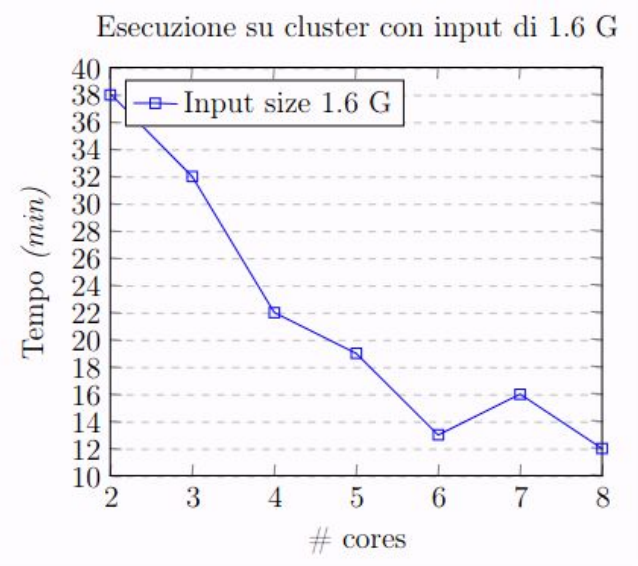
```
allocca@masterunisa:~$ hdfs dfs -ls -h outputrnd_input2g_temp
Found 27 items
-rw-r--r--   1 allocca supergroup          0 2023-05-19 18:27 outputrnd_input2g_temp/_SUCCESS
-rw-r--r--   1 allocca supergroup      1.8 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00000
-rw-r--r--   1 allocca supergroup      1.3 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00001
-rw-r--r--   1 allocca supergroup      1.1 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00002
-rw-r--r--   1 allocca supergroup      1.3 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00003
-rw-r--r--   1 allocca supergroup      1.7 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00004
-rw-r--r--   1 allocca supergroup      1.2 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00005
-rw-r--r--   1 allocca supergroup      1.3 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00006
-rw-r--r--   1 allocca supergroup        994 2023-05-19 18:27 outputrnd_input2g_temp/part-00007
-rw-r--r--   1 allocca supergroup      1.3 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00008
-rw-r--r--   1 allocca supergroup      1.4 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00009
-rw-r--r--   1 allocca supergroup      1.1 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00010
-rw-r--r--   1 allocca supergroup      1.3 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00011
-rw-r--r--   1 allocca supergroup      1.7 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00012
-rw-r--r--   1 allocca supergroup      1.6 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00013
-rw-r--r--   1 allocca supergroup      1.4 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00014
-rw-r--r--   1 allocca supergroup      1.2 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00015
-rw-r--r--   1 allocca supergroup      1.6 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00016
-rw-r--r--   1 allocca supergroup      1.4 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00017
-rw-r--r--   1 allocca supergroup      1.8 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00018
-rw-r--r--   1 allocca supergroup      1.7 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00019
-rw-r--r--   1 allocca supergroup      1.5 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00020
-rw-r--r--   1 allocca supergroup      1.8 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00021
-rw-r--r--   1 allocca supergroup      1.2 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00022
-rw-r--r--   1 allocca supergroup      1.7 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00023
-rw-r--r--   1 allocca supergroup      1.8 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00024
-rw-r--r--   1 allocca supergroup      1.9 K 2023-05-19 18:27 outputrnd_input2g_temp/part-00025
```

Figura : Taglia dizionari con input 2 GB con filtro

A fronte di tali risultati viene naturale considerare l'elaborazione di tale dati in modi sequenziale in locale (occupano pochissima memoria) che risulterà essere molto veloce su questo piccolo dataset

Performance





04

Conclusioni

Considerazioni finali

Conclusioni

La ricerca sull'applicazione dell'algoritmo **Kfinger** con **Apache Spark** per l'analisi degli **overlap genomici** ha evidenziato l'importanza e i vantaggi di questo approccio nell'ambito della genomica.

Attraverso l'utilizzo di Kfinger, siamo in grado di individuare e caratterizzare in modo efficiente gli overlap tra sequenze genomiche, fornendo **informazioni cruciali** sulla struttura e l'organizzazione del genoma.

L'implementazione di Kfinger con Apache Spark ha portato **notevoli vantaggi** in termini di **scalabilità** e **velocità di calcolo**.

Grazie alla natura distribuita di Spark, siamo in grado di elaborare **grandi volumi di dati genomici** in tempi ridotti, migliorando l'efficienza complessiva dell'analisi. Questo ci consente di ottenere risultati più rapidamente e di affrontare **sfide complesse** nell'analisi dei dati genomici.

Grazie per l’attenzione