Università degli studi di Salerno

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Tesina in
Sistemi Operativi Avanzati

Kfinger - Implementazione in Scala/Spark

Candidati:
Alfonso Luciani
Matr.: 0522500929
Antonio Allocca
Matr.: 0522501527

Relatore:
Prof. Giuseppe Cattaneo

Anno Accademico 2022/23

Indice

Abstract

L'algoritmo Kfinger è un metodo innovativo per il calcolo degli Overlap tra lunghe reads genomiche, che mira a identificare e quantificare le regioni sovrapposte tra sequenze genomiche. L'obiettivo principale dell'algoritmo è quello di fornire una valutazione precisa e veloce degli overlap tra sequenze genomiche per supportare l'analisi e l'interpretazione dei dati genomici, compito fondamentale, ad esempio, per costruire l'overlap graph. L'overlap tra sequenze genomiche può fornire informazioni importanti sull'evoluzione, sulla struttura genica, sulle regioni conservate e sulle relazioni funzionali tra geni o elementi genomici. La tecnica proposta utilizza le fattorizzazioni di Lyndon. E' ben noto che la fattorizzazione di Lyndon di una stringa 'e una sequenza non crescente di fattori, che sono parole di Lyndon. Tale fattorizzazione 'e unica e pu'o essere calcolata in tempo lineare. Viene utilizzata una rappresentazione delle reads come sequenza di interi, precisamente data dalla lunghezza delle parole di Lyndon che compongono la fattorizzazione della read (Lyndon Fingerprints). E' stato dimostrato che due stringhe che condividono un overlap comune, condividono anche un insieme di fattori consecutivi nella loro fattorizzazione. L'implementazione dell'algoritmo Kfinger offre numerosi vantaggi, tra cui un'elaborazione ad alta velocità e un basso consumo di risorse computazionali. Inoltre, l'accuratezza e la robustezza dell'algoritmo ne fanno uno strumento affidabile per il calcolo degli Overlap, contribuendo alla comprensione delle relazioni genomiche, all'identificazione di regioni conservate e all'analisi di eventi di duplicazione o frammentazione del DNA. In questo progetto di tesi si è lavorato su una implementazione Scala che si basa sull'algoritmo Kfinger, flessibile , robusta e collaudata da utilizzare in Spark un framework per applicazioni distribuite per stimare velocemente la distanza fra n sequenze, in particolare fra n genomi, inoltre per verificarne la correttezza è stata effettuata una sperimentazione su diversi genomi reali e confrontando l'output con altri software che si sono basati su questo algoritmo.

Capitolo 1

Introduzione

La bioinformatica ha reso possibile l'analisi delle sequenze genomiche su larga scala, portando a importanti scoperte e comprensioni nel campo della biologia molecolare. Tuttavia, l'aumento esponenziale del volume dei dati genomici richiede l'utilizzo di algoritmi efficienti e scalabili per elaborare e analizzare queste informazioni in tempi ragionevoli. In questo contesto, l'algoritmo Kfinger si configura come una soluzione promettente per il calcolo degli overlap tra sequenze genomiche in un ambiente distribuito con Apache Spark. [30]

L'obiettivo di questa tesi è esplorare l'applicazione dell'algoritmo Kfinger in un contesto distribuito con Apache Spark e analizzarne le prestazioni e l'efficacia nell'elaborazione di grandi volumi di dati genomici. L'utilizzo di Spark come framework di elaborazione distribuita consente di sfruttare la scalabilità e la potenza di calcolo offerte da un cluster di macchine, consentendo una maggiore velocità di calcolo e la gestione efficiente di dataset di grandi dimensioni. [3]

Nella prima parte della tesi, verrà fornita una panoramica sulla bioinformatica e sull'importanza dell'analisi delle sequenze genomiche in un contesto distribuito. Saranno affrontate le sfide specifiche che sorgono nell'elaborazione e nell'analisi di grandi volumi di dati genomici e sarà sottolineata l'importanza di algoritmi efficienti e scalabili per affrontare tali sfide.

Successivamente, verrà presentato l'algoritmo Kfinger e saranno esplorate le sue caratteristiche chiave. Sarà illustrato come Kfinger possa essere adattato e implementato nell'ambiente distribuito di Apache Spark, sfruttando le funzionalità di Spark per l'elaborazione parallela e distribuita dei dati genomici.

Nella parte successiva della tesi, verranno affrontate le questioni di prestazioni e scalabilità. Saranno presentati i risultati di prove sperimentali che valutano le prestazioni dell'algoritmo Kfinger in un ambiente distribuito con Apache Spark, confrontando le prestazioni con altri approcci di calcolo degli overlap. Saranno considerati fattori come la velocità di calcolo, l'utilizzo delle risorse di calcolo e la gestione efficiente della memoria per valutare l'efficacia di Kfinger in un contesto distribuito.

Successivamente, verranno esplorate le diverse applicazioni dell'algoritmo Kfinger in ambito genomica e bioinformatica. Saranno descritti casi di studio specifici in cui l'utilizzo di Kfinger in un ambiente distribuito ha fornito risultati significativi e sarà discussa la sua rilevanza per l'analisi delle sequenze genomiche su larga scala.

Infine, verranno tratte le conclusioni dalla ricerca condotta. Saranno riassunti i principali punti trattati nella tesi, sottolineando l'importanza dell'algoritmo Kfinger nell'elaborazione distribuita delle sequenze genomiche e fornendo suggerimenti per sviluppi futuri e ulteriori ricerche nel campo.

Attraverso questa tesi, si mira a fornire una comprensione approfondita dell'applicazione dell'algoritmo Kfinger in un ambiente distribuito con Apache Spark e ad esplorarne le potenzialità nel contesto dell'analisi delle sequenze genomiche.

1.1 Analisi delle sequenze genomiche

L'analisi delle sequenze genomiche riveste un ruolo fondamentale nel comprendere i processi biologici, identificare relazioni evolutive e individuare regioni funzionalmente significative nel genoma.

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.

In un contesto distribuito, come quello fornito da Apache Spark, è possibile affrontare le sfide dell'analisi genomica su larga scala. La distribuzione del carico di lavoro su un cluster di macchine consente di sfruttare la potenza di calcolo parallela e la capacità di gestire grandi volumi di dati. Ciò è particolarmente rilevante nell'analisi delle sequenze genomiche, in cui l'elaborazione di intere sequenze o frammenti richiede tempo e risorse considerevoli.

La scalabilità è un aspetto chiave nell'analisi delle sequenze genomiche in un contesto distribuito. Gli algoritmi utilizzati devono essere in grado di gestire efficacemente grandi dataset, adattandosi alle dimensioni variabili delle sequenze genomiche e alle richieste di calcolo. L'utilizzo di algoritmi efficienti e scalabili è cruciale per garantire tempi di esecuzione ragionevoli e un utilizzo efficiente delle risorse di calcolo disponibili.

Nel contesto della tesi, si sottolinea l'importanza di algoritmi come Kfinger per affrontare le sfide specifiche dell'analisi delle sequenze genomiche in un ambiente distribuito. Kfinger è progettato per l'elaborazione efficiente degli overlap tra sequenze genomiche e la sua implementazione in Apache Spark offre la possibilità di sfruttare la potenza di calcolo distribuita per l'analisi di grandi dataset genomici.

In conclusione, l'importanza dell'analisi delle sequenze genomiche in un contesto distribuito evidenzia le sfide specifiche che sorgono nell'elaborazione e nell'analisi di grandi volumi di dati genomici.

L'adozione di algoritmi efficienti e scalabili, come Kfinger in ambiente distribuito, si configura come una soluzione promettente per affrontare tali sfide e ottenere risultati di analisi genomiche accurati e tempestivi.

1.2 Overlap: cosa sono

Nel contesto della genomica, gli overlap si riferiscono alla sovrapposizione di sequenze di DNA o geni su uno stesso segmento o su segmenti adiacenti del genoma. Gli overlap possono avere diverse cause e implicazioni a livello funzionale.

Gli overlap possono essere classificati in diverse categorie, tra cui:

Overlap di geni: si verifica quando i confini di due o più geni si sovrappongono. Questo può avvenire sia nella stessa direzione (sovrapposizione di geni consecutivi) che in direzioni opposte (sovrapposizione di geni antisenso). Gli overlap di geni possono influenzare la regolazione dell'espressione genica, il processo di trascrizione e la produzione di proteine.

Overlap di trascrizione: si verifica quando due o più trascritti (RNA) si sovrappongono parzialmente o totalmente. Gli overlap di trascrizione possono essere il risultato di meccanismi complessi, come l'uso di diverse regioni promotori o la produzione di varianti di trascritti tramite lo splicing alternativo. Gli overlap di trascrizione possono influenzare la regolazione dell'espressione genica e la diversità dei trascritti.

Overlap di regioni regolatorie: si verifica quando le regioni regolatorie, come i promotori o gli enhancer, di due o più geni si sovrappongono. Questo può implicare un'integrazione delle vie di segnalazione e una regolazione coordinata dell'espressione genica.

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.

1.3 KFinger

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 significative tra frammenti di sequenze che possono indicare relazioni evolutive, regioni funzionali o errori di sequenziamento [24].

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). L'idea fondamentale di Kfinger è quella di generare un indice basato sui k-mer per accelerare la ricerca degli overlap [13].

Per creare l'indice, Kfinger suddivide le sequenze genomiche in frammenti di lunghezza prefissata e genera gli insiemi di k-mer per ogni frammento. Successivamente, gli insiemi di k-mer vengono organizzati in una struttura dati ad albero chiamata k-tree, che consente una ricerca efficiente degli overlap [27].

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. Questo processo è eseguito in modo efficiente utilizzando strutture dati ottimizzate come gli alberi di suffissi o le strutture dati di hashing [17].

L'implementazione di Kfinger in un ambiente distribuito, come Apache Spark, consente di sfruttare la potenza di calcolo parallela e distribuita per accelerare ulteriormente il calcolo degli overlap su grandi dataset genomici [31]

L'utilizzo di Kfinger in un ambiente distribuito con Spark può portare a notevoli miglioramenti delle prestazioni rispetto a metodi tradizionali basati su algoritmi sequenziali. L'elaborazione distribuita permette di suddividere il carico di lavoro su più macchine, riducendo il tempo di calcolo complessivo e consentendo l'analisi di dataset genomici di dimensioni considerevoli [22].

Kfinger è un algoritmo efficace e scalabile per il calcolo degli overlap tra sequenze genomiche. La sua implementazione in un ambiente distribuito come Apache Spark offre notevoli vantaggi in termini di velocità di calcolo e gestione efficiente dei grandi volumi di dati genomici. L'utilizzo di Kfinger in combinazione con Spark rappresenta un'opzione promettente per l'analisi genomica su larga scala [7].

1.4 Lyndon Fingerprint

Le Lyndon Fingerprints possono essere utilizzate insieme alla fattorizzazione per ottenere una rappresentazione efficiente delle sequenze genomiche. La fattorizzazione delle sequenze genomiche consiste nel suddividere una sequenza in sottostringhe chiamate fattori, che rappresentano le unità di base della sequenza [10].

Nel contesto delle Lyndon Fingerprints, la fattorizzazione viene utilizzata per generare le sottostringhe di Lyndon. Questo processo coinvolge la scomposizione della sequenza in una serie di fattori, che vengono poi combinati in modo specifico per ottenere le sottostringhe di Lyndon. Ogni sottostringa di Lyndon corrisponde a una concatenazione di uno o più fattori della sequenza originale [2].

La combinazione dei fattori per generare le sottostringhe di Lyndon segue regole specifiche basate sull'ordinamento lessicografico. Le sottostringhe di Lyndon vengono generate selezionando i fattori in base al loro valore lessicografico più basso. Questa selezione garantisce che le sottostringhe di Lyndon siano irriducibili rispetto all'ordinamento lessicografico, poiché non è possibile sostituire una sottostringa con una sua sottosequenza in modo che l'ordinamento lessicografico rimanga invariato [18].

Una volta ottenute le sottostringhe di Lyndon utilizzando la fattorizzazione, è possibile calcolare le Lyndon Fingerprints per ciascuna di esse. Le fingerprints sono calcolate utilizzando algoritmi di hashing che generano una rappresentazione univoca per ciascuna sottostringa di Lyndon [8].

Le Lyndon Fingerprints consentono di ridurre lo spazio di archiviazione richiesto per rappresentare le sequenze genomiche, grazie alla loro natura compatta. Utilizzando la fattorizzazione e le Lyndon Fingerprints, è possibile rappresentare le sequenze genomiche in modo efficiente e identificare facilmente le sottostrutture irriducibili all'interno delle sequenze [9].

1.5 KFinger con Lyndon Fingerprint

Kfinger, insieme alle Lyndon Fingerprints, è un algoritmo utilizzato per il calcolo degli overlap tra sequenze genomiche. Le Lyndon Fingerprints sono una struttura dati basata sui concetti di sottostringhe di Lyndon e di fingerprinting, che vengono combinati per ottenere una rappresentazione compatta delle sequenze genomiche.

Le sottostringhe di Lyndon sono una classe speciale di sottostringhe di una sequenza, che hanno la proprietà di essere "irriducibili" rispetto all'ordinamento lessicografico. Questo significa che non è possibile suddividere una sottostringa di Lyndon in sottostringhe più piccole, o in altre parole, una sottostringa di Lyndon non può essere rappresentata come concatenazione di copie di una stessa sottostringa.

Le Lyndon Fingerprints vengono generate utilizzando le sottostringhe di Lyndon delle sequenze genomiche. Queste sottostringhe vengono ordinate in modo lessicografico e viene calcolato un hash per ciascuna sottostringa. L'hash rappresenta in modo univoco la sottostringa e viene utilizzato come fingerprint per quella specifica sottostringa di Lyndon.

L'utilizzo delle Lyndon Fingerprints con Kfinger consente di ridurre lo spazio di archiviazione richiesto per rappresentare le sequenze genomiche e di migliorare l'efficienza dell'algoritmo. Invece di memorizzare l'intera sequenza, è sufficiente memorizzare le Lyndon Fingerprints e le informazioni associate ad esse.

Durante la fase di ricerca degli overlap, Kfinger utilizza le Lyndon Fingerprints per confrontare le sequenze genomiche e identificare gli overlap significativi. L'algoritmo confronta le fingerprints delle sottostringhe di Lyndon delle sequenze e determina se esiste un'overlap basato sulle fingerprints corrispondenti.

L'utilizzo delle Lyndon Fingerprints consente una riduzione significativa dello spazio di archiviazione richiesto, poiché le fingerprints sono rappresentazioni più compatte delle sequenze genomiche. Inoltre, l'utilizzo di sottostringhe di Lyndon garantisce che gli overlap identificati siano basati su sottostrutture irriducibili delle sequenze, fornendo informazioni più rilevanti per l'analisi genomica.

In sintesi, l'utilizzo delle Lyndon Fingerprints con Kfinger consente di ottimizzare l'efficienza e la gestione dello spazio di archiviazione nel calcolo degli overlap tra sequenze genomiche. L'uso combinato di queste tecniche rappresenta un approccio promettente per l'analisi genomica su larga scala, consentendo di affrontare in modo efficiente le sfide legate alla gestione e all'elaborazione di grandi volumi di dati genomici.

Capitolo 2

Strumenti utlizzati

2.1 GARR

La rete accademica e di ricarca **GARR** nacque negli anni '80 e porta con se una unga storia di sperimentazione di protocolli di rete. Attualmente la **GARR** prende il nome di **GARR-B—GARR Broadband Service** [1] è basata su una struttura **ATM** che connette **4** nodi. La rete mesh che li interconnette costituisce la rete **backbone**.

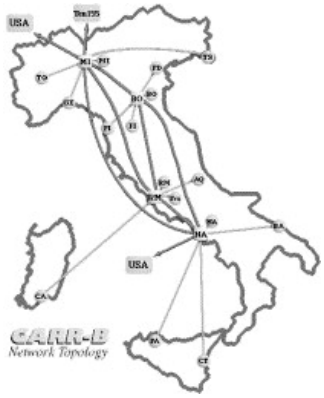


Figura 2.1: Topologia della rete **GARR-B**

Gli **obiettivi** del GARR possono essere divisi in due gruppi principali:

i.) Aumento della banda a fronte della richiesta per comunicare con il **General Internet** [19]

ii.) Aumento della disponiblilità a fronte di una sempre maggiore richiesta di risorse dedicate a servizi scientifici e di ricerca come **Computing GRID** [16] o la creazione di centri di calcolo.

2.1.1 GARR-G

La continua richiesta di **banda** da parte degli utenti sta portando a molte istituzioni accademiche a valutare l'idea di accedere alla rete **GARR** con velocità nell'ordine dei **Gigabit** dando così vita alla **GARR-G**. [1]

2.2 Apache Spark

Apache Spark è un framework open-source pensato per **Computing GRID** [16] e l'elaborazione di enormi moli di dati che è riuscito a superare il MapReduce di Hadooop.

Figura 2.2: Logo di **Apache Spark**

Come specificato in [23] **Apache Spark** permette di estendere le potenzialità del **Computing GRID** in molti modi tra i quali troviamo:

- Rende le applicazioni fino a **100** volte più veloci.
- Facile da programmare in quanto fornisce **API** per molti linguaggi come **Python**, **Java**, **Scala** ed **R**
- Astrazione dei dati
- Data Frame distribuito [4]

Mentre in **ambito applicativo** abbiamo computazioni complesse come:

- Machine learning
- Streaming
- Graph processing
- ...

Come spiegato in [15] da solo **Apache Spark** non permette di usufruire di sevizi di **Data Storage** in quanto si occupa di eseguire computazioni sulla **Spark JVM**. Per questo, come nel nostro caso, **Apache Spark** viene utilizzato in concomitanza con sistemi di archiviazione distribuiti (**HDFS** nel nostro caso) e cluster manager (nel nostro caso **Hadoop YARN**).

Apache Spark si è rilevato un framework fondamentale per il proseguimento del progetto in quanto come anche specificato in [11] è possibile utilizzare Apache Spark in ambito **bioinformatico** portando a prestazioni elevate e tenendo testa ai **requisiti** di **spazio** e **potenza** di calcolo per portare a termine l'elaborazione.

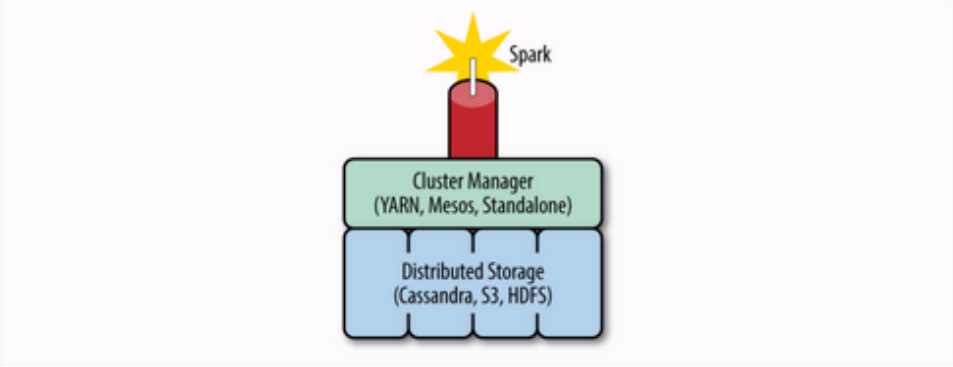


Figura 2.3: Data Processing Ecosystem

2.2.1 Spark Components

Tra le componenenti presenti all'interno dell'ecosistema di Spark possiamo trovare lo **Spark Core** che fornisce **API** per **Python**, **Java**, **Scala** ed **R** fornendo così un query language ad alto livello per elaborare i dati.

Un'altra componente fondamentale sono gli oggetti **RDD** che sono **oggetti distribuiti** con due importanti caratteristiche:

i.) Lazy evalutated [29]

ii.) Tipizzati staticamente

Tali oggetti possono essere elaborati tramite funzioni di **map** e **reduce**, oltre che a fornire un canale **I/O** tra l'**HDFS** e la **Spark JVM**.

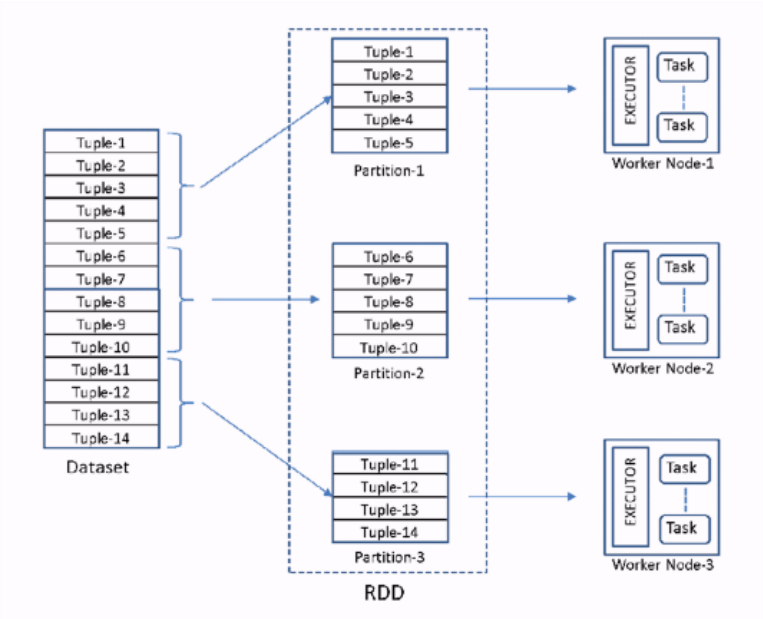


Figura 2.4: Archiviazione e Trasformazione **RDD**

2.3 Hadoop YARN

Hadoop YARN (**Y**et **A**nother **R**esource **N**egotiator)[26] è una piattaforma di **gestione delle risorse** che si occupa di gestire le risorse di calcolo all'interno di un **cluster** e di utilizzarle per gestire lo **sviluppo di codice** e per l'**elaborazione dei dati**.

Figura 2.5: Logo di **Hadoop YARN**

Separando la gestione delle risorse al modello di programmazione, **YARN** affida la gestione dello **scheduling** delle risorse a **componenti dedicate**. Così facendo il modello **MapReduce** è solo una delle applicazioni che vengono eseguite da YARN fornendo inoltre **flessibilità** per la scelta del **framework di prograzione** (nel nostro caso è stato scelto **Spark**). Tale **framework** verrà eseguito su **YARN** e si occuperà di coordinare:

- IPC [28]
- Flusso di esecuzione
- Ottimizzazioni dinamiche delle risorse in base alla necessità

In un'architettura distribuita **YARN** si pone tra l'**HDFS** e il **framework** utilizzato per eseguire le applicazioni.

2.4 HDFS

L'HDFS (**H**adoop **D**istributed **F**ile **S**ystem) è un **file system distribuito** progettato per poter essere ospitato su **hardware low-cost** mantenendo un alto gradi di **tolleranza ai guasti**. Fornisce un **alto throughput** per i dati della applicazioni ed è pensato per applicazioni con **grandi data set**.

Figura 2.6: Logo di **HDFS**

È un'implementazione open-source del **GFS** (**G**oogle **F**ile **S**ystem)[14], HDFS suddivide i dati in **blocchi di 128 MB** tranne l'ultimo ma può essere impostato a piacimento. Il cluster HDFS è supportato da un'architettura **master/slave** con un singolo **Name Node** che è il **master** e gestisce il namespace del file system e regola l'accesso ai file da parte dei client. Gli **slave** prendono il nome di **Data Node** che gestiscono la memoria del dispositivo su cui sono montati.

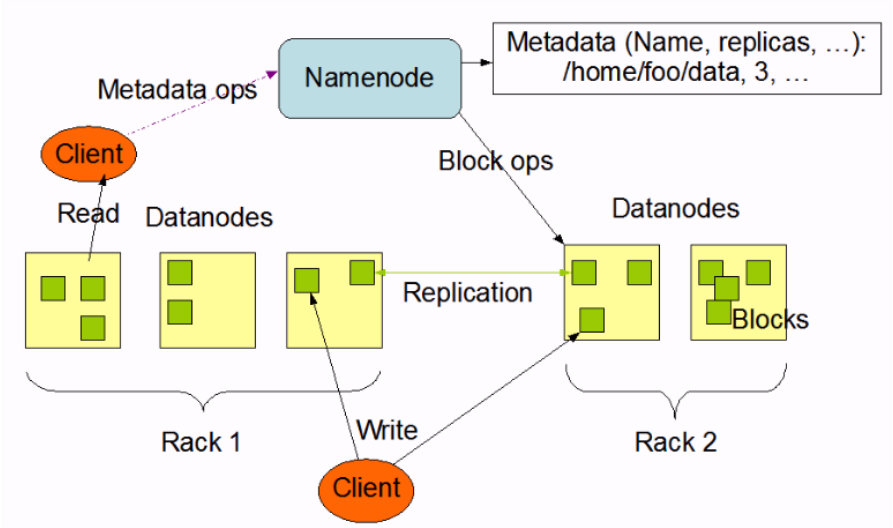


Figura 2.7: Architettura **HDFS**

2.5 Scala

Come spiegato in [20] Scala unisce i concetti della **programmazione object-oriented** [21] e quello della **programmazione funzionale** [12] in un linguaggio **tipizzato staticamente** ed ha come obiettivo la creazione di componenti e sistemi di essi. Tra i **requisiti** di Scala troviamo e il concetto di **codice scalabile** e la sua interoperabilità by design con i linguaggi **Java** e **C#**, infatti adotta la sintassi e il sistema di tipizzazione da questi linguaggi.

Figura 2.8: Logo si **Sala**

La documentazione che è stata utilizzata per sviluppare tale progetto è stata ricavata dai seguenti siti:

- *https://docs.scala-lang.org/*
- *https://index.scala-lang.org/apache/spark*
- *https://spark.apache.org/docs/3.4.0/api/scala/org/apache/spark/index.html*

2.6 Zabbix

Figura 2.9: Logo si **Zabbix**

Zabbix come introdotto da [25] è un software **open-source** per il **monitoraggio** di sistemi supportato da **UNIX**, **Linux**, **BSD**, **Mac OS X** e **Windows** sviluppato da **Alexei Vladishev** e la sua azienda **Zabbix SIA**. Tra le **features** più importanti offerte da tale servizio troviamo:

- Distributed monitoring
- Client per Linux, BSD, Windows, Mac OS X e versioni di UNIX commericali
- Web-based interface
- Notifiche via e-mail, SMS o Jabber
- SNMP
- Grafici

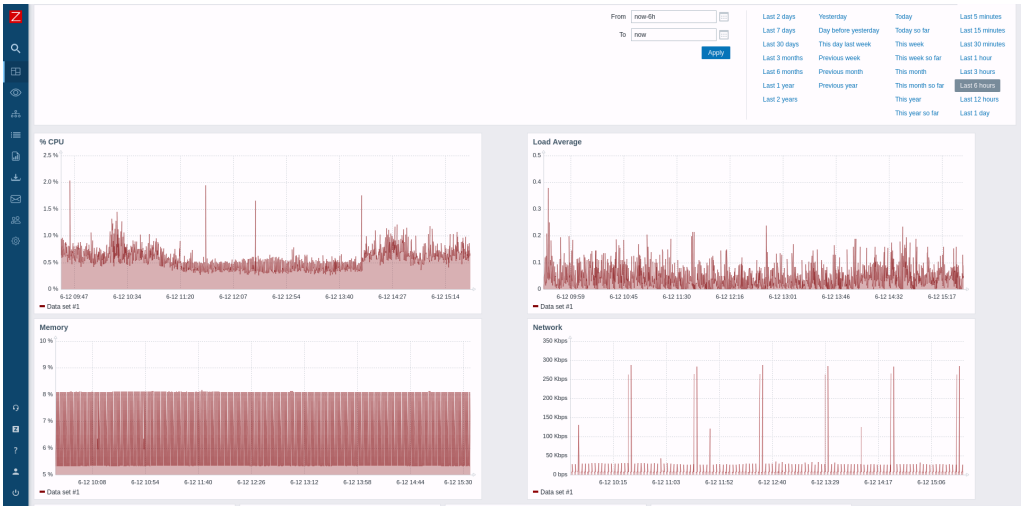


Figura 2.10: Esempio Dashboard di **Zabbix**

Capitolo 3

Spiegazione algoritmo

L' algoritmo proposto in [6] si divide in quattro fasi principali:

- Lettura delle fingerprint
- Popolazione del dizionario *dict_ occ_ kmers*
- Popolazione dei dizionari *min_ sharing_ dict* e *matches_ dict*
- Calcolo e riconciliazione degli overlaps

Nella prima parte dell'algoritmo si impostano i parametri che andranno poi ad influire sulla precisione e sulla velocità dell'algoritmo.

Inizialmente verranno lette le fingerprint contenute all'interno di un file testuale ottenute tramite l'elaborazione di un file FASTA[5]. Questo ci permetterà di trasformare ogni losta di interi in una lista di triple (**VALORE**,**READ**,**START**) che successivamente ci porteranno alla costruzione dei k-fingers su cui possiamo lavorare. L'aggiunta di ridondanza è fondamentale in quanto ogni riga è valutata in modo lazy. Ciò a lungo andare porterà anche un aumento delle performance ma è un effetto collaterale.

Di ogni read sarà presente sia la sua versione originale che quello **R**everse_**A**nd_**C**omplement e verranno fattorizzate distintamente. Ciò porta che ad ogni read vengano associate 2 righe (le due **fingerprint**) che avranno come **ID** lo stesso **ID** ma le fingerprint di una read in **R**everse_**A**nd_**C**omplement avrà come valore booleano associato a quell'**ID** pari ad **1**. La fattorizzazione viene fatta ogni 300 basi che verranno poi accumulate nella fingerprint. Ciò viene fatto per limitare la lunghezza dei fattori che conterrà la fingerprint

Successivamente iene popolato il dizionario *dict_ occ_ kmers* in modo tale che ad ogni entry che ha come chiave un k-mer viene associata una lista di coppie (**READ**,**START**) che corrispondono rispettivamente all'**ID** della read in cui appare il k-mer e l'**indice** in cui appare il k-mer all'interno della **fingerprint** (read).

Si procede poi con la creazione di altri 2 dizionari:

- *min_ sharing_ dict*: contiene il numero di volte in cui due read ID sono presenti all'interno delle coppie (READ,START) relative ad un k-mer
- *matches_ dict*: Rightmost e Leftmost k-mer in comune, per ogni coppia di read.

Il primo dizionario nelle ultime versioni sviluppate (in **C** e in **Scala**) stato accorpato all'altro.

Le chiavi nei dizionari sono liste di stringhe per adattarci alla sintassi del linguaggio e poter operare successivamente operazioni di **map** e **reduce** di con elementi con **chiave**. Le strutture dati prima nominate saranno risultato implicito dell'esecuzione del programma ma si potrà avere anche una dichiarazione esplicita di tali strutture a piacimento. Sono tutte ottenute tramite **map**, **reduce** e **filter** dei risultati precedenti.

Vengono poi prodotte in output le regioni comuni e gli overlap, in questo modo. Una volta calcolate le posizioni dei k-mer leftmost L e rightmost R, per ogni coppie di reads, si deve verificare che L ed R diano origine ad una regione comune. Questo si verifica solo se lo start di L della seconda read è $\leq$ dello start di R nella seconda read. Da una regione comune viene prodotto un overlap se la regione copre a sufficienza l'overlap. Al termine viene effettuato il calcolo e la riconciliazione degli overlap. La riconciliazione è causata dal fatto che le reads sono state duplicate e quindi serve per ricondurre gli overlap all'effettivo strand dei reads in input.

L'ultima fase dell'algoritmo proposto **non** viene sviluppata in Scala in quanto gli Overlap ottenuti tramite l'esecuzione del programma sono pochi ed il **dataset** per quest'ultima fase **non giustifica** l'impiego di un HDFS o di un'esecuzione distribuita.

```
val result = sc.textFile(inputPath)
        .map(line => getreads(line))
        .flatMap(triple => getkmers(triple))
        .reduceByKey(_++_)
        .flatMap(x => getmatchesdict(x))
       .reduceByKey((a,b) =>{
          if(a(0) > b(0)){
            a(0)=b(0)
            a(1)=b(1)
          }
          if(a(2) < b(2)){
            a(2)=b(2)
            a(3)=b(3)
          }

          if (a(4) < min_shared_kmers) {a(4) = a(4)+b(4)}

          a

        })
        //.filter((tupla) => tupla._2(4) >= min_shared_kmers)
        .map(y => (y._1,(y._2(0),y._2(1),y._2(2),y._2(3),y._2(4))))
```

3.1 Parametri

I parametri considerati in entrambe le implementazioni sono i seguenti:

- Dimensione e minima lunghezza totale dei k-mers: `k = 7, min_total_length = 40`.

- Minimo numero di k-mers (unici) che due reads in overlap devono condividere (in assoluto): `min_shared_kmers = 4`.

- Massimo numero di occorrenze di un k-mer nei reads (valore compatibile con la coverage dell'input): `max_kmer_occurrence = -1`. Un valore pari a -1 indica assenza del controllo.

- Parametri di filtraggio delle regioni comuni in output:

  - massima differenza percentuale tra le lunghezze (bp) delle due stringhe della regione comune (mettere 0.0 nel caso di reads senza errore), `max_diff_region_percentage = 0.0`.
  - minima lunghezza (bp) delle due stringhe della regione comune, `min_region_length = 100`.

- Parametro di filtraggio delle regioni comuni:

  - minima percentuale di copertura dei k-mers contigui rispetto alla lunghezza della regione comune. Una regione lunga L bp deve contenere almeno L/min_total_length di k-mers comuni, `min_region_kmer_coverage = 0.27`

- Parametri di filtraggio degli overlap in output:

  - minima copertura percentuale (bp) della regione comune rispetto all'overlap, `min_overlap_cove = 0.70`
  - minima lunghezza dell'overlap da produrre in output (1200 per read senza errore), `min_overlap_length = 600`

3.2 Lettura delle fingerprint

Nel file testuale ogni fingerprint è su un'unica riga e le fingerprints sono rappresentate come sequenze

ID_BOOL 45 7 9 1 1 | 7 65 2 3 54 |,

dove ogni | era stato introdotto per separare i segmenti della fattorizzazione e ID è semplicemente n_x, dove ID identifica l'n-esima read e $BOOL$ è un flag che indica se la fingerprint é del read originale ($x = 0$) oppure del read dopo reverse and complement ($x = 1$).

Per la realizzazione di liste di fingerprint, per ogni riga del file in input vengono innanzitutto rimosse tutte le occorrenze del carattere |. Successivamente dalla prima stringa presente all'interno della riga vengono estratti i dati della read. In particolare otteniamo un intero che rappresenta il suo **ID** e un valore booleano.

Se il valore booleano assume valore 0, allora la read è originale, altrimenti è la sua versione duplicata a di cui ne viene calcolato il r&c.

```
def getreads(line:String): Array[(Int,Int,Int)] ={

    val l_line = line.split("␣").filter(_ != "|")
    val read_id=l_line.head.split("_")
    val fingerprint_list=l_line.slice(2,l_line.size)


    val res_fin_list = new ArrayBuffer[(Int,Int,Int)]()

    var temp = 1
    if(read_id(1)=="1") temp = 0

    for(i <- 0 until fingerprint_list.size by 1){
      res_fin_list +=((fingerprint_list(i).toInt,read_id(0).toInt*2+temp,i
    }
    return res_fin_list.toArray

}
```

3.3 Popolazione del dizionario *dict_occ_kmers*

Per la creazione delle occorrenze dei k-mer, si deve leggere ciascuna delle fingerprint. Per ogni intero presente nella fingerprint si procede a comporre in k-mer che è formato dall'intero e, quando possibile, dai $k-1$ interi successivi.

Si può inoltre, se impostato, controllare l'unicità del k-mer all'interno delle reads, nel senso che vengono memorizzate per un dato read solo le occorrenze di k-mers che appaiono una sola volta nel read stesso.

Ogni k-mer (sequenza di interi) è associato ad una sottostringa di read (concatenazione dei fattori corrispondenti). Per evitare k-mer che supportano sottostringhe troppo corte (che porterebbero a trovare overlap errati), si controlla se la somma dei valori che compongono il k-mer sia maggiore o uguale della grandezza minima totale, parametro fissato all'inizio. Solo in questo caso viene creata all'interno del dizionario una entry che avrà per chiave una rappresentazionne del $k$mer come stringa e come valore una lista di coppie (READ,START). Precisamente, START è la posizione di inizio dell'occorrenza del k-mer chave all'interno della fingerprint della READ. Per costruzione, le tuple sono ordinate per valore crescente del valore READ e poi per START. Vengono poi eliminati i k-mer che occorrono una sola volta nell'insieme in input o troppe volte.

```
def getkmers(triple: Array[(Int,Int,Int)]):
    Array[(String,ArrayBuffer[(Int,Int)])]={

    var res = ArrayBuffer[(String,ArrayBuffer[(Int,Int)])]()

    for (i <- 0 until triple.size-k by 1){
      var somma=0
      var kmer = ""
```

```
      for(j <- 0 until k by 1){
        var temp_v =triple(i+j)._1
        somma=somma+temp_v
        kmer+=temp_v.toString+"_"
      }
      if (somma >= min_total_length)
        res += ((kmer,ArrayBuffer((triple(i)._2,triple(i)._3))))
    }

    return res.toArray

}
```

```
1       [8 3 26 6 2 2 8 ]=(6017,95)
     1 [24 5 3 24 13 3 1 ]=(2696,78)[5 7 26 6 2 2 9 ]=(3413,706)(
       7401,670)(14113,3271)(17041,57)(17173,1079)(17547,100)
     2 [1 3 26 6 2 2 10 ]=(1729,338)(3475,649)(11505,274)[2 11 26
        6 2 2 10 ]=(5922,1405)(16156,550)
     3 [6 3 26 6 2 2 11 ]=(1651,904)(6827,1579)(9915,1355)(12141,
       111)(14103,823)(15729,1382)[1 3 26 6 2 2 11 ]=(5868,941)[7
        3 11 9 7 9 3 ]=(7760,594)(9944,1156)(12014,645)
     4 [10 70 58 6 2 2 12 ]=(3903,49)(4245,901)
     5 [8 4 8 30 3 8 29 ]=(2588,1339)(9676,582)(12878,235)[2 4 8
       30 3 8 29 ]=(9266,1924)
     6 [10 70 58 6 2 2 14 ]=(5077,604)(15839,538)
     7 [1 3 26 6 2 2 15 ]=(9394,872)(11824,304)(11866,615)(12106,
       1007)(19560,614)
     8 [8 3 26 6 2 2 17 ]=(7501,593)(14399,286)[1 1 3 24 13 3 9 ]
       =(17242,29)(17972,90)
     9 [5 24 9 62 3 9 2 ]=(13172,1338)(13854,416)(15234,1352)(174
       04,276)
    10 [3 5 13 54 16 12 3 ]=(2945,567)(5065,108)(11017,752)(16569
       ,263)(18781,532)(19539,1052)
    11 [3 11 20 7 10 18 4 ]=(1725,2006)(7113,1631)[8 3 26 6 2 2 2
       0 ]=(5155,1074)(7433,740)(16733,254)(19423,445)
    12 [4 3 2 11 6 64 21 ]=(30,250)(1014,452)(2702,65)(3670,549)(
       4044,460)(4222,204)(4946,1299)(6974,406)(10438,498)(12350,
       205)(13518,171)(15564,247)(16338,2089)(16402,824)(17794,14
       14)[1 3 2 11 6 64 21 ]=(10554,108)
    13 [1 6 1 18 2 22 8 ]=(7501,1228)
       @
       @
N…  dict_occorrenze.log                                  0% ln:1 %:1
"dict_occorrenze.log" 1704560L, 239844256B
```

Figura 3.1: Esempio risultato del **Dizionario delle occorrenze**

3.4 Popolazione dei dizionari *min_sharing_dict* e *matches_dict*

In questa fase vengono trovati quelli che sono i leftmost e i rightmost k-mer per ogni coppia di reads. Inoltre viene creato un dizionario per tenere conto di quanti k-mers sono condivisi da due reads,

Questa è sicuramente la fase più onerosa in quanto abbiamo dei for innestati.

Dai dizionari dei leftmost e rightmost k-mers si inferiscono gli overlap per il set dei reads duplicati. Subito dopo gli overlap vengono riconciliati per avere gli overlap per il set di reads originali.

```
def getmatchesdict(t_entry : (String,ArrayBuffer[(Int,Int)])):
    Array[(String,Array[Int])] ={

      val matchesdict = new ArrayBuffer[(String, Array[Int])]()

      for (i <- 0 until t_entry._2.size by 1){
        var first_occ = t_entry._2(i)._1

        for (j <- i+1 until t_entry._2.size by 1){
          var second_occ = t_entry._2(j)._1
          var key = first_occ+"_"+second_occ
          var value = new Array[Int](5)

          value(0)=t_entry._2(i)._2
          value(1)=t_entry._2(j)._2
          value(2)=t_entry._2(i)._2
          value(3)=t_entry._2(j)._2

          value(4) = 1
          matchesdict += ((key,value))
        }

      }

      return matchesdict.toArray


}

.reduceByKey((a,b) =>{
          if(a(0) > b(0)){
            a(0)=b(0)
            a(1)=b(1)
          }
          if(a(2) < b(2)){
            a(2)=b(2)
```

```
        a(3)=b(3)
      }

      if (a(4) < min_shared_kmers) {a(4) = a(4)+b(4)}

      a

    })
```

```
1       (18281,18305)[340,560,340,560]
        1 (18226,19682)[29,982,412,1379]
        2 (18181,18946)[86,488,86,488]
        3 (18233,19843)[3,451,3,451]
        4 (18176,19108)[1339,1650,1339,1650]
        5 (18232,19879)[1043,245,1046,248]
        6 (18176,19114)[1343,166,1343,166]
        7 (18207,18762)[789,1431,793,1435]
        8 (18239,19787)[38,282,336,601]
        9 (18206,18797)[293,2373,1360,150]
       10 (18190,19310)[220,60,220,60]
       11 (18193,18575)[152,484,153,485]
       12 (18207,18769)[501,1984,503,1986]
       13 (18189,19221)[173,9,470,322]
       14 (18207,18774)[1443,1198,1443,1198]
       15 (18199,18519)[98,1304,98,1304]
       16 (18180,19002)[1170,124,1170,124]
       17 (18281,18331)[757,78,1083,366]
       18 (18187,19419)[80,484,82,486]
       19 (18181,18972)[115,875,115,875]
       20 (18206,18813)[293,1905,293,1905]
       21 (18186,19454)[25,205,25,205]
       22 (18280,18304)[797,766,797,766]
       23 (18232,19841)[1043,113,1046,116]
       24 (18282,18370)[201,950,202,951]
       25 (18227,19683)[997,3,1417,428]
       26 (18187,19428)[79,678,81,680]
       27 (18235,19941)[782,2,782,2]
       28 (18207,18789)[1439,931,1439,931]
N…  dict_match.log                                    0% ln :1 %:1
"dict_match.log" 5797834L, 175722081B
```

Figura 3.2: Esempio risultato del **Dizionario dei match**

3.5 Esecuzione

Per effettuare la lettura delle fingerprint si acquisisce una stringa alla volta. Se è una stringa del tipo id_bool, allora sarà instanziata una fingerprint con valore id pari all'id letto e il parametro isreverse assumerà il valore bool. In seguito verrà letta la lista di interi che compongono la fingerprint.

Dopo questa fase, si passa al riempimento del dizionario delle occorrenze dei k-mer temporaneo. Per ogni k-mer che viene letto, viene prima controllato che soddisfi alcuni requisiti,

come la grandezza totale del k-mer maggiore o uguale di una grandezza fissata. In seguito viene considerata la read e l'indice in cui tale k-mer appare, e nel dizionario la chiave k-mer, avrà come valore la coppia read e start. Quando deve essere inserita una coppia, prima si cerca all'interno del dizionario l'entry con chiave ottenuta applicando la funzione hash sul k-mer, ottenendo così un dizionario (la lista associata a tale k-mer). Sul dizionario così ottenuto, viene ricercata la presenza o meno della entry che ha per chiave il k-mer. Se non è presente alloco una nuova entry che ha per chiave il k-mer. Dopo aver effettuato tale controllo, alla lista di coppie associata al k-mer viene aggiunta la nuova coppia appena letta.

Una volta riempito il dizionario temporaneo, si fa il filtering delle entry che hanno al massimo un numero di coppie associate ad esso pari al parametro `max_kmer_occurence` stabilito all'inizio.

Successivamente viene computato il dizionario dei match utilizzando il dizionario delle occorrenze. Per ogni entry del dizionario delle occorrenze, vengono analizzate le coppie associate ad ogni k-mer e viene calcolata la lista di interi che rappresenta la posizione leftmost e quella rightmost del k-mer. Le coppie vengono analizzate 2 alla volta. Così facendo le due read rispettive diventano la chiave della entry del dizionario dei match e il valore associato a tale chiave sarà la lista di interi che stabilisce il leftmost ed il rightmost. Contemporanemente viene salvato anche il numero di volte in cui una coppia di read viene aggiornata (non cè più quindi il dizionario min_sharing_dict).

I controlli per l'aggiornamento o meno dei valori leftmost e rightmost viene fatta direttamente all'interno dell'update per alleggerire il codice nel main.

Infine viene calcolato il dizionario degli overlap associando ad ogni coppia di read una lista parametri (descritti precedentemente) se e solo se rispettano determinate condizioni descritte all'interno dell'algoritmo. Quindi, gli overlap riconciliati saranno prodotti come record dei campi seguenti:

- id del primo read (senza il terminatore di strand)
- lunghezza del primo read
- posizione 0-based di inizio dell'overlap sul primo read
- posizione 1-based di fine dell'overlap sul primo read
- id del secondo read (senza il terminatore di strand)
- lunghezza del secondo read
- posizione 0-based di inizio dell'overlap sul secondo read
- posizione 1-based di fine dell'overlap sul secondo read
- strand del secondo read rispetto al primo (0: se uguale; 1: se opposto)

3.6 Algoritmo di 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 (r_1, r_2) nel dizionario si ottiene il leftmost k-mer L e il rightmost k-mer R.

Vengono poi fatte le due seguenti verifiche:

(1) r_1 e r_2 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 r_2 (per costruzione L viene prima di R in r_1).

Se tali verifiche hanno esito positivo, allora si determina la coppia di sottostringhe s_1 e s_2 (cioè la regione comune) indotte dai k-mers L e R sui corrispondenti reads. Cioè, s_1 è la sottostringa della read che corrisponde a r_1 che inizia nella stessa posizione di inizio della sottostringa corrispondente a L su r_1 e finisce nella stessa posizione di fine della sottostringa corrispondente a R su r_1. Analogamente, s_2 è la sottostringa di r_2 che inizia nella stessa posizione di inizio della sottostringa corrispondente a L su r_2 e finisce nella stessa posizione di fine della sottostringa corrispondente a R su r_2. Viene calcolato poi un valore "atteso" minimo di k-mers condivisi tra le due reads sulla base dei parametri in input `min_region_kmer_coverage`, `min_total_length` e della lunghezza della regione comune (minimo tra le due lunghezze di s_1 e s_2). Se il numero di k-mers condivisi da r_1 e r_2 supera questo minimo atteso, e inoltre si ha che (1) la differenza di lunghezza tra s_1 e s_2 non supera una certa percentuale `max_diff_region_percentage` del massimo tra le due lunghezze di s_1 e s_2 e (2) il massimo delle lunghezze di s_1 e s_2 é oltre la soglia `min_region_length`, allora si procede a calcolare l'overlap tra r_1 e r_2. Questo é praticamente ottenuto estendendo la regione comune in modo da ottenere un overlap prefisso-suffisso oppure un overlap in cui una delle due reads corrisponde a una sottostringa (anche non propria) dell'altro. A questo punto, se il minimo tra le due lunghezze di s_1 e s_2 è almeno una percentuale `min_overlap_coverage` della lunghezza di tale overlap e la lunghezza dell'overlap é almeno il parametro `min_overlap_length`, allora l'overlap viene tenuto (altrimenti viene scartato).

Si considerino ora, dati due reads in input n_1 e n_2 tutti gli overlap trovati, che saranno al massimo 4 dal momento che ogni read viene duplicato. Per ogni coppia n_1, n_2 viene tenuto l'overlap (tra quelli trovati) di lunghezza massima. La riconciliazione rimappa eventualmente l'inizio e la fine dell'overlap selezionato rispetto alle reads originali. Ad esempio, se si ha che la versione originale di n_1 ha un suffisso che coincide con un prefisso della versione r&c di n_2, allora devo rimappare l'overlap in un prefisso della versione originale di n_2.

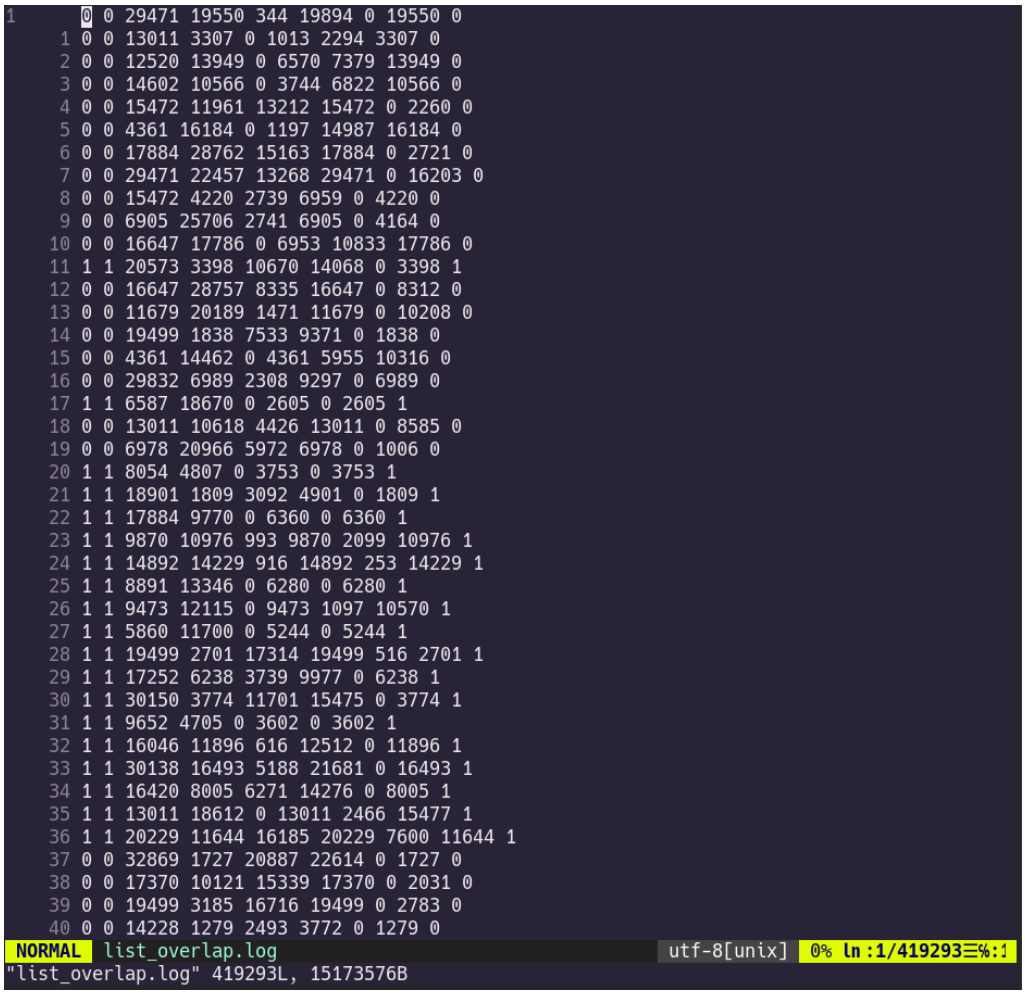


Figura 3.3: Lista di **Overlap**

Capitolo 4

Implementazione in Scala/Spark

L'implementazione in Scala/Spark è stata realizzata seguendo l'algoritmo originale. Facendo ciò ci siamo accorti che l'algoritmo rispecchia pienamente il modello di programmazione **map/reduce**. L'algoritmo comincia importando *org.apache.spark* che è il core delle funzionalità di Spark. Infatti ci da accesso ad oggetti come org.apache.spark.**SparkContext** che è l'entry point di Spark e org.apache.spark.rdd.**RDD** che è il tipo di dato che rappresenta l'oggetto distribuito e ci permette di utilizzare molte delle operazioni con parallelismo implicito.

```
import org.apache.spark.input.PortableDataStream
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.streaming.{Seconds, StreamingContext}
```

Intoltre andremo anche ad importare dei tipi di dati mutabili come ArrayBuffer che ci permettono di avere flessibilità all'interno degli array (Gli array e le tuple in Scala sono trattati come **frozen set** in Python).

```
import scala.collection.mutable.ArrayBuffer
```

Una volta importati tutti gli oggetti che ci servono per poter usufruire dei vantaggi Spark ed aver capito di quali strutture dati necessitiamo, possiamo procedere a dichiarare come **val** (Quindi costanti) le variabili che caratterizzeranno la precisione e i paramentri dell'esecuzione del nostro algoritmo. Tali parametri sono stati descritti già in precedenza, in particolare viene impostato:

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

Dopo aver impostato tali valori possiamo procedere a dichiarare lo **SparkContext** e le directory di input/output che ssaranno segnate come **var** (Ovvero mutabili)

```
var sc:  SparkContext = null
```

```
var outputPath: String = "output"
var inputPath: String = "input"
```

Successivamente si procede con la stesura del vero e proprio **main** che conterrà in primis la dichiarazione dello SparkContext e poi l'implementazione dell'algoritmo.

```
def main(args: Array[String]) {

      if (args.length < 2) {
        System.err.println
            ("Errore nei parametri sulla command line")
        System.err.println
            ("Usage:\nit.unisa.di.soa.WordCount
           inputDir outputDir [local|masterName]")
        throw new IllegalArgumentException
            (s"illegal number of argouments.
           ${args.length} should be at least 2")
        return
      }

      outputPath = args(1)
      inputPath = args(0)

      if (args.length > 2) {
        local = (args(2).compareTo("local") == 0)
      }
      val sparkConf = new SparkConf().
            setAppName("Scala WordCount").
            setMaster(if (local) "local" else "yarn")
      // Create the SparkContext
      val sc = new SparkContext(sparkConf)

      inputPath =
        if (local) "data/"
        else s"hdfs://${args(2)}:9000/${inputPath}"

      println(s"Opening Input Dataset: ${inputPath}")
```

4.1 Stage 0

L'algoritmo comincia con la lettura delle **Fingerprint**. Tali fingerprint possono essere lette dallo SprkContext, 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.

```
val result = sc.textFile(inputPath)
```

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

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

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 "|" tramite il comando:

```
val l_line = line.split("␣").filter(_ != "|")
```

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 ottenendolo in modo deterministico e con complessità lineare tramite un'equazione che usa come variabili il valore **ID** ed il valore di **BOOL** ovvero:

$ID.toInt * 2 + temp$

temp è una variabile che ha valore complementare rispetto al valore **BOOL**

```
var temp = 1
if(read_id(1)=="1") temp = 0
```

Dopo il completamento delle operazioni questa fase può ritenersi conclusa.

$$1\_0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \rightarrow (1,1,1)(1,1,2)(1,1,3)(1,1,4)(1,1,5)(1,1,6)(1,1,7)(1,1,8)(1,1,9)$$

```
def getreads(line:String): Array[(Int,Int,Int)] ={

      val l_line = line.split("␣").filter(_ != "|")
      val read_id=l_line.head.split("_")
      val fingerprint_list=l_line.slice(2,l_line.size)


      val res_fin_list = new ArrayBuffer[(Int,Int,Int)]()

      var temp = 1
      if(read_id(1)=="1") temp = 0

      for(i <- 0 until fingerprint_list.size by 1){
        res_fin_list +=
            ((fingerprint_list(i).toInt,read_id(0).toInt*2+temp,i))
      }
      return res_fin_list.toArray
```

```
}
```

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**.

```
.flatMap(triple => getkmers(triple))
```

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 *"_"*.

Una volta lette le k triple si fa la somma del loro campo **INTERO** e si verica che il valore così ottenuto sia $\geq min\_total\_length$. 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 siamo leggendo (Il primo vlore del k-mer).

(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)),(5_6_7_8,(1,1))

```
def getkmers(triple: Array[(Int,Int,Int)]):
    Array[(String,ArrayBuffer[(Int,Int)])]={

    var res = ArrayBuffer[(String,ArrayBuffer[(Int,Int)])]()

    for (i <- 0 until triple.size-k by 1){
```

```
        var somma=0
        var kmer = ""
        for(j <- 0 until k by 1){
          var temp_v =triple(i+j)._1
          somma=somma+temp_v
          kmer+=temp_v.toString+"_"
        }
        if (somma >= min_total_length)
          res += ((kmer,ArrayBuffer((triple(i)._2,triple(i)._3))))
    }

    return res.toArray

}
```

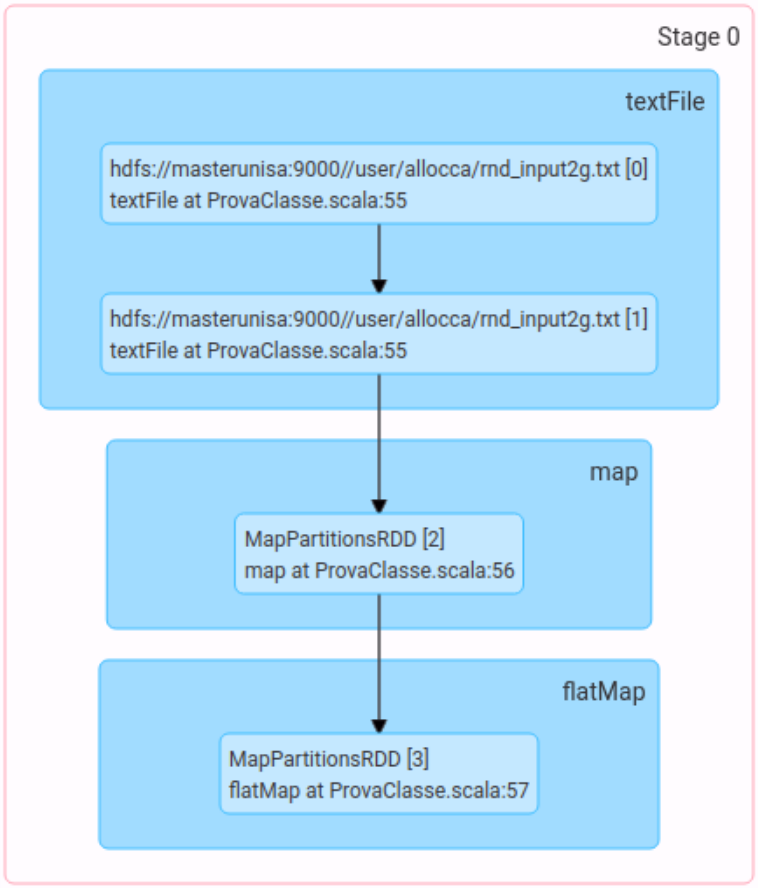


Figura 4.1: Stage **0**

4.2 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 precendetemente

ottenuti. Per fare ciò utilizziamo proprio la struttura dell' **RDD**.

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:

i.) **reduceByKey**

ii.) 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 list di coppie ogni volta.

La concatenazione di liste è espressa con l'operatore ++ di Scala.

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

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 dei 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.

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 trutturra 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** risultate.

```
def getmatchesdict(t_entry : (String,ArrayBuffer[(Int,Int)]))
    :Array[(String,Array[Int])] ={

    val matchesdict = new ArrayBuffer[(String, Array[Int])]()

    for (i <- 0 until t_entry._2.size by 1){
        var first_occ = t_entry._2(i)._1
```

```
        for (j <- i+1 until t_entry._2.size by 1){
          var second_occ = t_entry._2(j)._1
          var key = first_occ+"_"+second_occ
          var value = new Array[Int](5)

          value(0)=t_entry._2(i)._2
          value(1)=t_entry._2(j)._2
          value(2)=t_entry._2(i)._2
          value(3)=t_entry._2(j)._2

          value(4) = 1
          matchesdict += ((key,value))
        }

      }

    return matchesdict.toArray
}
```

Avendo eseguito una *flatmap* avremo che l'RDD risultate sarà un insieme di coppie (**READ1_READ2**,(**READ1**,**START1**,**READ2**,**START2**,1))

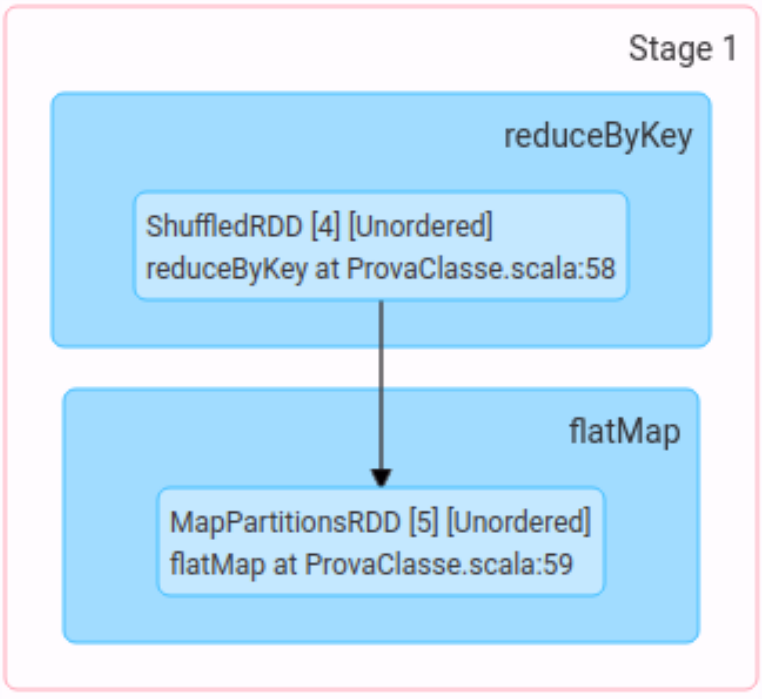


Figura 4.2: Stage **1**

4.3 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**.

```
reduceByKey((a,b) =>{
    if(a(0) > b(0)){
        a(0)=b(0)
        a(1)=b(1)
    }
    if(a(2) < b(2)){
        a(2)=b(2)
        a(3)=b(3)
    }

    if (a(4) < min_shared_kmers) {a(4) = a(4)+b(4)}

    a

})
```

Il numero di volte in cui queste coppie compaiono all'interno dell'**RDD** deve essere $\leq$ *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**).

Infine viene effettuato un filtraggio di tutte le coppie che hanno un numero di k-mer condivisi $\leq$ *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.

```
    .filter((tupla) => tupla._2(4) >= min_shared_kmers)
    .map(y => (y._1,(y._2(0),y._2(1),y._2(2),y._2(3),y._2(4))))

result.saveAsTextFile(outputPath)
sc.stop()
```

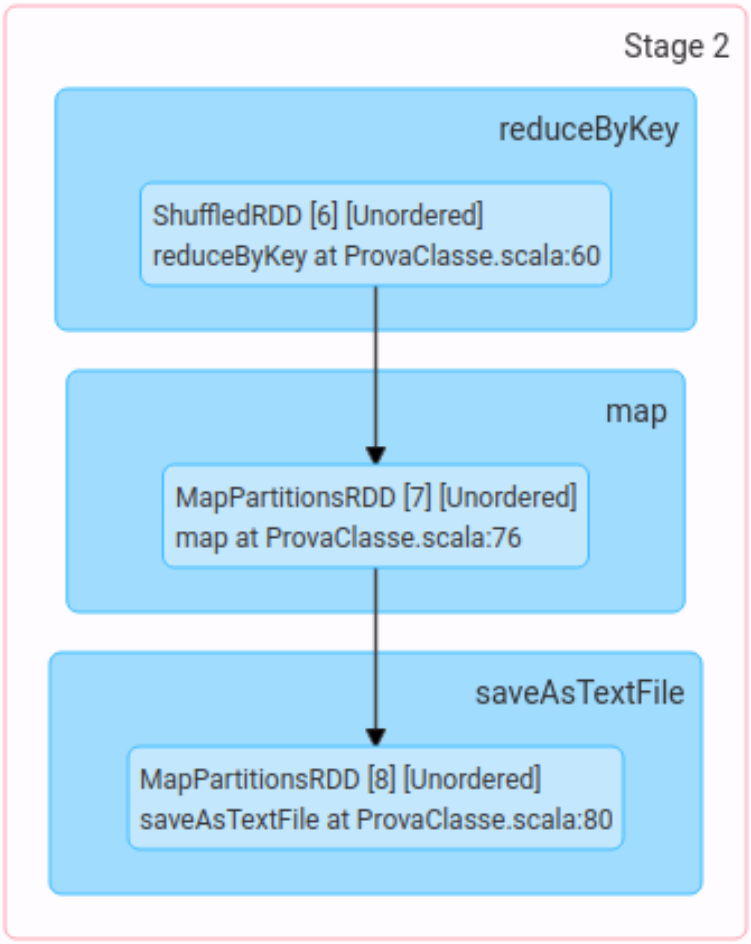


Figura 4.3: Stage **2**

4.4 Implementazione con mapPartitions

Un ulteriore miglioramento è dato dall'impiego della funzione **mapPartitions** di Scala. Tale funzione ritorna un nuovo RDD applicando una funzione ad ogni partizione dell'**RDD** al fine di non utilizzare una **map** che invece elabora ogni riga dell'RDD in modo "*seriale*"

```
val result = sc.textFile(inputPath)
        .mapPartitions(iterator =>
            iterator.map(line => getreads(line)))
        .mapPartitions(iterator =>
            iterator.flatMap(triple => getkmers(triple)))
        .reduceByKey(_++_)
        .mapPartitions(iterator =>
            iterator.flatMap(x => getmatchesdict(x)))
       .reduceByKey((a,b) =>{
          if(a(0) > b(0)){
            a(0)=b(0)
            a(1)=b(1)
          }
          if(a(2) < b(2)){
            a(2)=b(2)
            a(3)=b(3)
          }

          if (a(4) < min_shared_kmers) {a(4) = a(4)+b(4)}

          a

        })
        //.filter((tupla) => tupla._2(4) >= min_shared_kmers)
        .map(y => (y._1,(y._2(0),y._2(1),y._2(2),y._2(3),y._2(4))))
```

Tale implementazione differisce di poco dalla precedente ma ci da una sostanziale **miglioramento delle prestazioni**.

4.4.1 Stages con mapPratitions

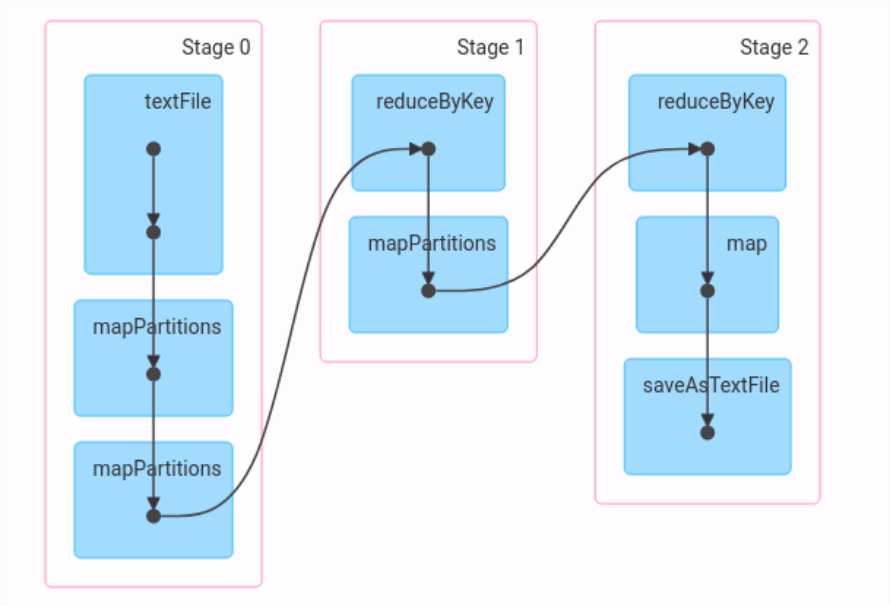


Figura 4.4: Stages con **mapPartitions**

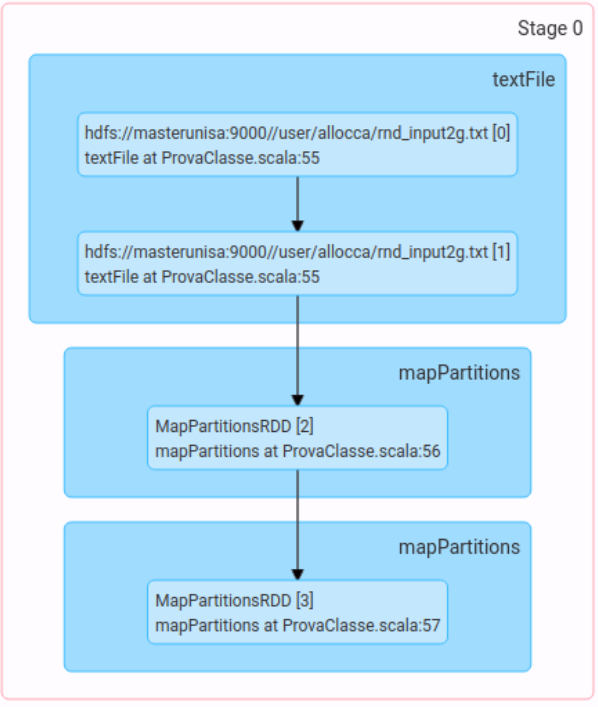


Figura 4.5: Stage 0 con **mapPartitions**

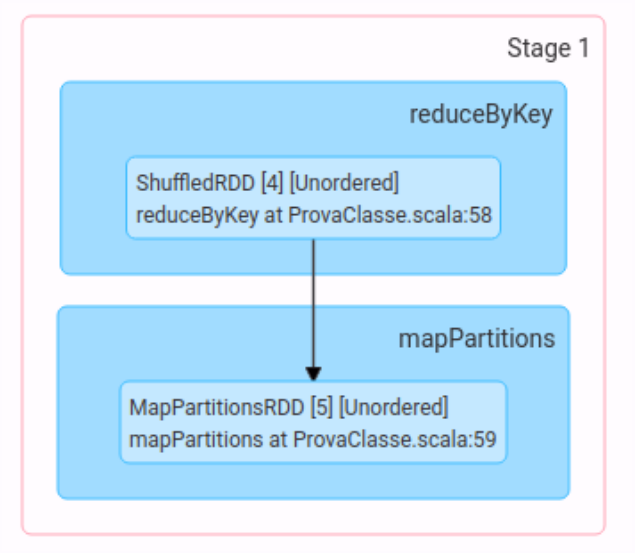


Figura 4.6: Stage 1 con **mapPartitions**

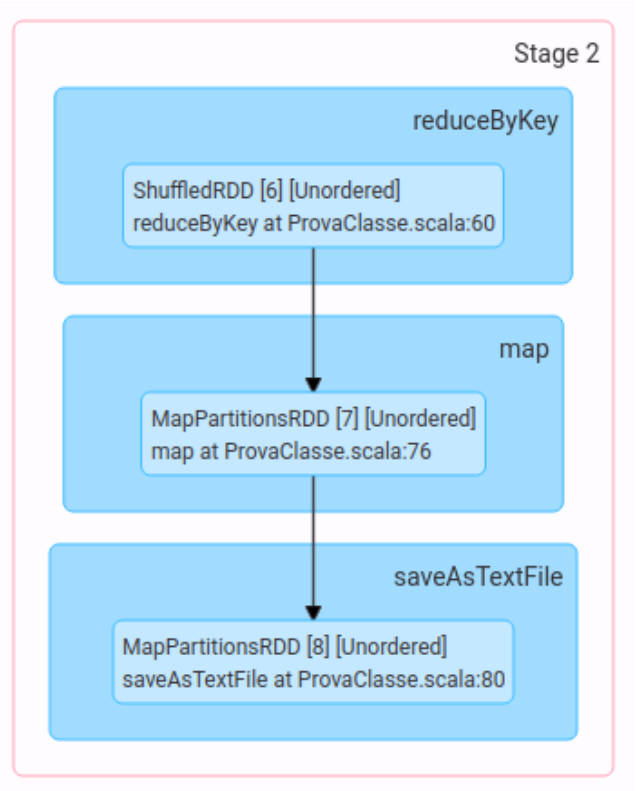


Figura 4.7: Stage 2 con **mapPartitions**

Capitolo 5

Esecuzione algoritmo

In questo capitolo si parlerà dell' esecuzione seriale e parallela e delle configurazioni utilizzate.

L'esecuzione seriale fa riferimento all'esecuzione in locale di un'implementazione in linguaggio C dell'algoritmo disponibile su *https://github.com/strumenti-formali-per-la-bioinformatica/kfinger-c-implementation*

Gli input utilizzati per il confronto sono liste di fingerprint stipati in file di 2 dimensioni diverse:

Size	-h
45894053	43.8 M
1723645784	1.6 G

Tabella 5.1: Taglie file in input.

5.1 Esecuzione seriale

L'esecuzione dell'algoritmo seriale è stata fatta in locale con un'implementazione in C ed una in Python.

L'algoritmo in locale è stato necessariamente eseguito con un l'input di taglia 43.8 M a fronte dell'enorme memoria che richiedeva l'esecuzione con il file di taglia 1.6 G. Ovviamente la soluzione proposta non è per niente scalabile e non c'è parallelismo.

I risultati ottenuti da tale esecuzione devono essere visti come una **baseline** di confronto per i risultati ottenuti con l'esecuzione parallela di tale algoritmo.

Di seguito in 5.5 verranno riportate anche le caratteristiche tecniche della macchina su cui è stato eseguito l'algoritmo ed i risultati ottenuti effettuando già un primo confronto tra **Python** e **C**.

Il programma verrà avviato con il comando

./final-program -l

L'esecuzione di tale programma con quella flag permetterà la generazione di file di *log* dei vari risultati intermedi. Tali risultati sono riportati in 3.1 3.2 3.3 e sono stati fondamentali per la **verifica** della **correttezza** della versione parallela dell'algoritmo.

Caratteristiche sistema

Nome dispositivo LAPTOP-26FE8SRB
Processore Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
RAM installata 16,0 GB (15,8 GB utilizzabile)
Tipo sistema Sistema operativo a 64 bit, processore basato su x64

Python		**C**	
Operazione	**Tempo(s)**	**Operazione**	**Tempo(s)**
Secondi lettura linee	10.52	Secondi lettura linee	2
Secondi creazione dizionario	100.20	Secondi creazione dizionario	25
Secondi creazione msdict e mdict	185.98	Secondi creazione msdict e mdict	67
Secondi calcolo overlaps	87.86	Secondi calcolo overlaps	5

Tabella 5.2: Dettagli esecuzione in locale.

5.2 Esecuzione parallela

L'esecuzione parallela è un'approccio utilizzato per accelerare il calcolo degli overlap tra sequenze genomiche e migliorare le prestazioni degli algoritmi impiegati. Invece di eseguire il calcolo in modo sequenziale, l'esecuzione parallela suddivide il lavoro tra più processori o nodi di calcolo, che lavorano contemporaneamente per ridurre il tempo di esecuzione complessivo. Un framework ampiamente utilizzato per l'esecuzione parallela è Apache Spark, che fornisce un'infrastruttura distribuita per l'elaborazione di grandi volumi di dati. Spark permette di sfruttare la potenza di calcolo di un cluster di macchine, coordinando le attività di calcolo in modo efficiente.

Nel contesto degli algoritmi di calcolo degli overlap, l'esecuzione parallela con Spark può essere implementata in diversi modi. Ad esempio, è possibile suddividere il carico di lavoro tra i nodi di calcolo assegnando a ciascun nodo un subset delle sequenze da analizzare. Questo approccio consente di elaborare parallelamente diverse porzioni delle sequenze e combinare i risultati ottenuti.

Inoltre, Spark offre anche operazioni di trasformazione e azioni che consentono di manipolare e aggregare i dati in modo efficiente. Queste operazioni possono essere sfruttate per eseguire passaggi intermedi dell'algoritmo di calcolo degli overlap, come l'ordinamento delle sequenze, la generazione delle sottostringhe o il calcolo delle fingerprints, in modo parallelo su più nodi di calcolo. L'esecuzione parallela con Spark può portare a notevoli miglioramenti delle prestazioni rispetto all'esecuzione sequenziale, consentendo di gestire grandi volumi di dati genomici in tempi ridotti. Tuttavia, è importante progettare in modo adeguato l'architettura del sistema e l'algoritmo di calcolo per garantire un'effettiva parallelizzazione del lavoro e sfruttare al meglio le risorse di calcolo disponibili.

Il programma sul cluster è stato lanciato sia con l'input di taglia 43.8 M sia su quello di taglia 1.6 G. L'esecuzione su file di piccole dimensioni ha fornito risultati simili alla baseline

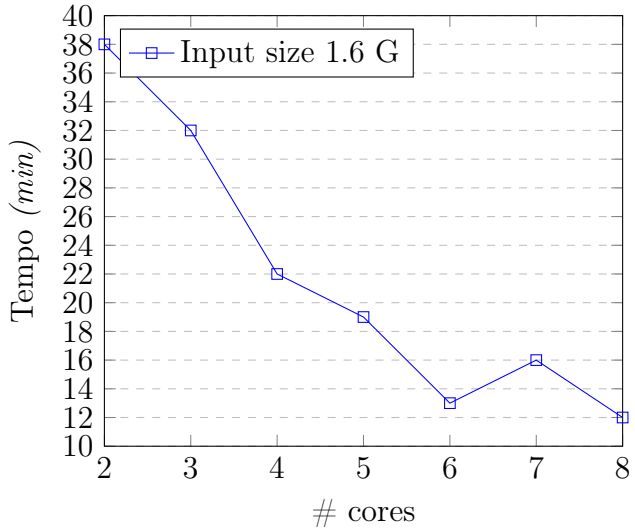
tracciata dall'esecuzione in parallelo, mentre l'esecuzione dello stesso sul file più grande non è stato possibile confrontarlo.

Il programma è stato lanciato con il comando:

org.apache.spark.deploy.SparkSubmit –master yarn –deploy-mode client –conf spark.driver.memory=4g –class it.unisa.di.soa.ProvaClasse –num-executors 8 –executor-memory 27g –executor-cores 7 target/ScalaWordCount-1.0-SNAPSHOT.jar /user/allocca/rnd_input2g.txt out2gb7execsenzafiltro masterunisa

Per le esecuzione sono stati fatti test su un numero diverso di executor cores portando ovviamente risultati diversi.

Esecuzione su cluster con input di 1.6 G



Esecuzione su cluster con input di 43.8 M

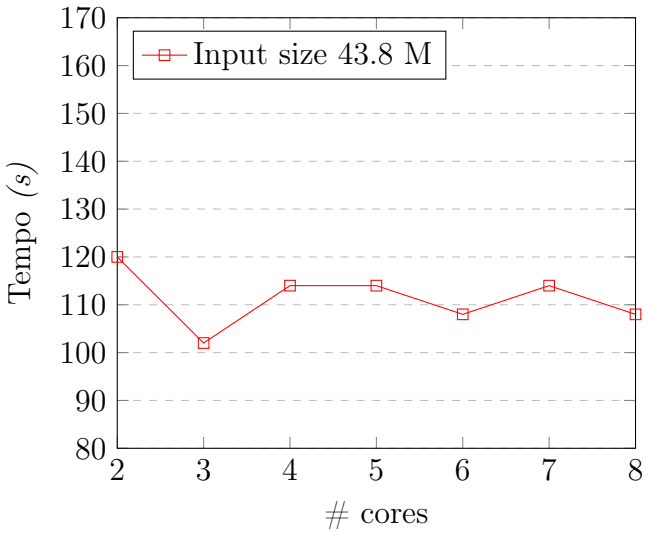


Figura 5.1: Esecuzione con input di **1.6 G**

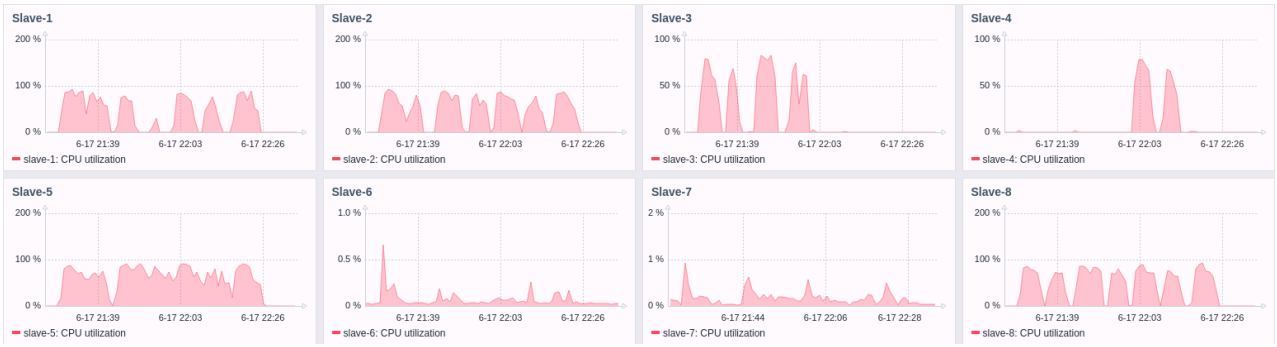


Figura 5.2: Slave con input di **1.6 G**

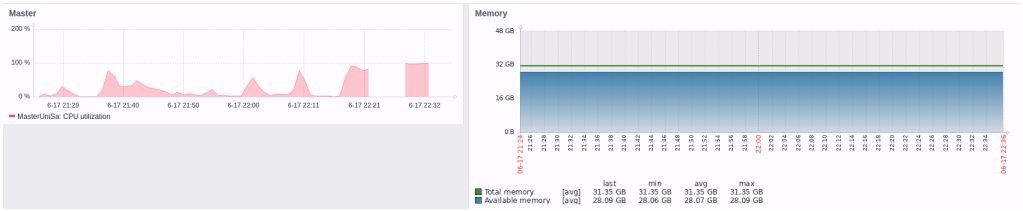


Figura 5.3: Master e memoria con input di **1.6 G**

Link per analisi approfondite:

- *http://90.147.185.21:18080/history/application_1682379418624_0280/jobs/* - 8 core
- *http://90.147.185.21:18080/history/application_1682379418624_0281/jobs/* - 7 core
- *http://90.147.185.21:18080/history/application_1682379418624_0282/jobs/* - 6 core
- *http://90.147.185.21:18080/history/application_1682379418624_0283/jobs/* - 5 core

Figura 5.4: Esecuzione con input di **43.8 M**

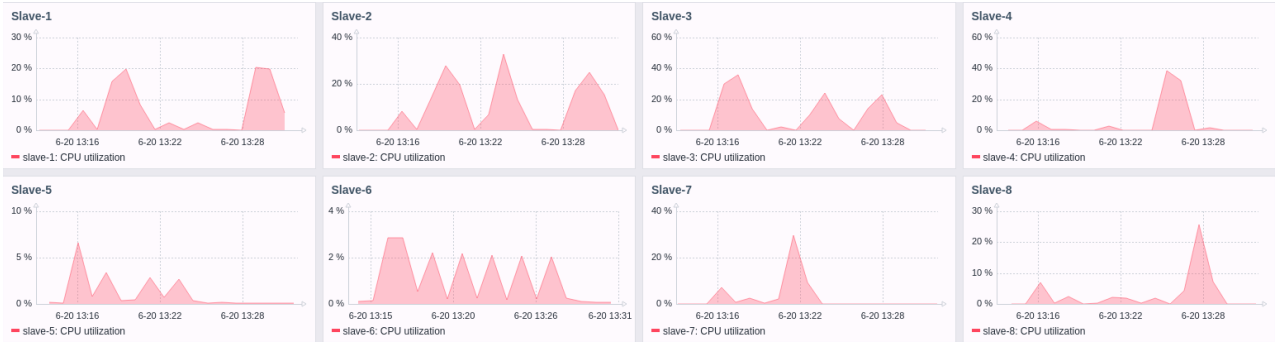


Figura 5.5: Slave con input di **43.8 M**

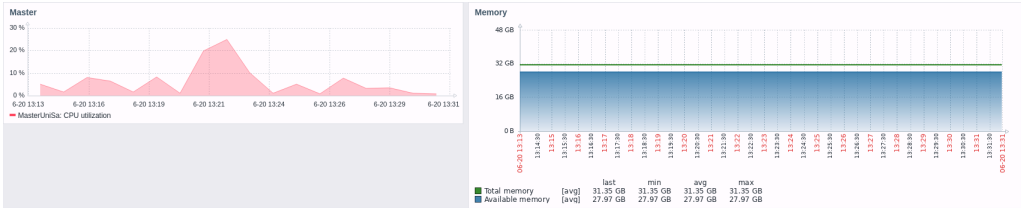


Figura 5.6: Master e memoria con input di **43.8 M**

Link per analisi approfondite:

- *http://90.147.185.21:18080/history/application_1687198779129_0160/jobs/* - 8 core
- *http://90.147.185.21:18080/history/application_1687198779129_0164/jobs/* - 7 core
- *http://90.147.185.21:18080/history/application_1687198779129_0003/jobs/* - 6 core
- *http://90.147.185.21:18080/history/application_1687198779129_0004/jobs/* - 5 core
- *http://90.147.185.21:18080/history/application_1687198779129_0005/jobs/* - 4 core
- *http://90.147.185.21:18080/history/application_1687198779129_0006/jobs/* - 3 core
- *http://90.147.185.21:18080/history/application_1687198779129_0007/jobs/* - 2 core

5.2.1 Configurazione cluster

Proprietà Spark	
Nome	**Valore**
spark.driver.memory	4g
spark.executor.cores	7
spark.executor.instances	8
spark.executor.memory	27g
spark.master	yarn

Tabella 5.3: Proprietà Spark.

Proprietà Hadoop	
Nome	**Valore**
dfs.blocksize	64m
dfs.replication	1
file.blocksize	8
spark.executor.memory	67108864
file.stream-buffer-size	4096
io.file.buffer.size	65536
yarn.nodemanager.resource.cpu-vcores	7
yarn.nodemanager.resource.memory-mb	30720
yarn.resourcemanager.webapp.address	masterunisa:8088
yarn.nodemanager.resource.memory-mb	30720

Tabella 5.4: Proprietà Hadoop.

5.2.2 Diverse configurazioni Spark

Per far partire più istanze dell'algoritmo con configurazioni spark divrese è stato eseguito il seguente script bash:

```
#! /bin/bash

if [[ $# -lt 1 || $# -gt 2 ]]; then
    echo "$0: arguments error: \nUsage:
        $0 inputDataSet output [local|masterName]"
    exit -1
fi

inputDS=$1
if [[ $# -eq 3 ]]; then
```

```
    mode=$3
else
    mode=masterunisa
fi

spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 8 --executor-memory 27g --executor-cores 7 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb7exec $mode



spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 7 --executor-memory 27g --executor-cores 6 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb6exec $mode



spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 6 --executor-memory 27g --executor-cores 5 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb5exec $mode



spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 5 --executor-memory 27g --executor-cores 4 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb4exec $mode



spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 4 --executor-memory 27g --executor-cores 3 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb3exec $mode


spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 3 --executor-memory 27g --executor-cores 2 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar   $inputDS out2gb2exec $mode
```

```
spark-submit --class it.unisa.di.soa.ProvaClasse \
  --master yarn --deploy-mode client --driver-memory 4g \
  --num-executors 2 --executor-memory 27g --executor-cores 1 \
  target/ScalaWordCount-1.0-SNAPSHOT.jar    $inputDS out2gb1exec $mode
```

5.3 Task

Sono stati analizzati i task di cui il programma si componeva per ogni Stage.

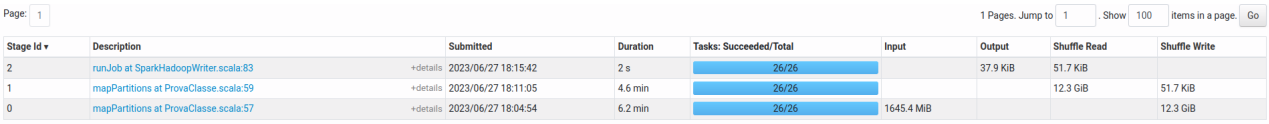


Figura 5.7: Starge e Task rispettivi

Ricaviamo così i tempi medi che per ogni stage saranno rispettivamente:

Stage	Tempo(s)
0	14.307692307692308
1	10.615384615384615
2	0.07692307692307693

Tabella 5.5: Tempi medi per task.

Ne risulta che i tempi medi per l'ultimo task sono molto bassi, mentre per i primi due il discorso è diverso. Questi tempi sono stati presi dopo l'impiego della mapPartitions.

Capitolo 6

Risultati ottenuti

6.1 Sequenziale e Parallelo

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 quando 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' agloritmo (Pienamente implementata) in quanto l'algoritmo presenta una complesstià 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**.

```
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 6.1: Taglia dizionari con input **2 GB** senza **filtro**

```
allocca@masterunisa:~$ hdfs dfs -ls -h outputinputbio2
Found 3 items
-rw-r--r--   1 allocca supergroup          0 2023-06-05 20:47 outputinputbio2/_SUCCESS
-rw-r--r--   1 allocca supergroup     22.4 M 2023-06-05 20:47 outputinputbio2/part-00000
-rw-r--r--   1 allocca supergroup     22.3 M 2023-06-05 20:47 outputinputbio2/part-00001
```

Figura 6.2: Taglia dizionari con input **40 MB** senza **filtro**

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 aggiungessimo 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** si **riduce** di molto.

$$filter((tupla) \Rightarrow tupla._2(4) \geq min\_shared\_kmers)$$

```
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 6.3: 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.

Capitolo 7

Conclusioni

In conclusione, l'algoritmo Kfinger rappresenta un potente strumento per il calcolo degli Overlap, fornendo informazioni preziose per la caratterizzazione delle sequenze genomiche e l'analisi delle relazioni tra di esse.

Gli overlap fra le sequenze genomiche possono rivelare relazioni funzionali o evolutive fra le sequenze, l'utilizzo di questo algoritmo ci permette di individuare sequenze simili o sequenze che condividono una particolare regione, fornendoci informazioni utili sulle relazioni evolutive o su sequenze correlate, in poco tempo grazie alla parallelizzazione con Spark.

Il genoma può contenere regioni complesse, come ripetizioni o strutture genomiche ripiegamenti, che possono rendere difficile l'analisi delle sequenze. L'algoritmo Kfinger aiuta a identificare e caratterizzare tali regioni complesse, consentendo una migliore comprensione della struttura e della funzione genomica.

L'algoritmo Kfinger offre un approccio efficiente e scalabile per il calcolo degli overlap tra sequenze genomiche. Grazie alla sua implementazione ottimizzata, Kfinger consente di ridurre notevolmente i tempi di calcolo rispetto ad altri metodi, consentendo l'analisi rapida di grandi volumi di dati genomici.

La sua efficacia e la sua efficienza lo rendono uno strumento di rilevanza significativa per la genomica computazionale e la ricerca biologica.

7.1 Limiti implementazione seriale

La computazione seriale, o esecuzione sequenziale di un programma, presenta alcuni limiti che possono influire sulle prestazioni e sull'efficienza del calcolo. Ecco alcuni dei principali limiti della computazione seriale:

i.) **Tempo di esecuzione**: La computazione seriale richiede che le istruzioni vengano eseguite una dopo l'altra, senza sfruttare il parallelismo o la capacità di eseguire più operazioni contemporaneamente. Ciò può comportare tempi di esecuzione più lunghi, specialmente quando si lavora con compiti complessi o con grandi volumi di dati.

ii.) **Utilizzo limitato delle risorse**: In una computazione seriale, solo una risorsa (ad esempio, un processore) viene utilizzata per eseguire il calcolo. Questo può limitare l'utilizzo delle risorse disponibili, specialmente in sistemi con molte risorse di calcolo a disposizione come i cluster di macchine.

iii.) **Scalabilità limitata**: La computazione seriale può avere difficoltà a scalare per gestire grandi volumi di dati o un aumento del carico di lavoro. Poiché le operazioni vengono eseguite una dopo l'altra, l'esecuzione seriale può diventare un collo di bottiglia quando si cerca di elaborare un numero crescente di dati o di eseguire compiti complessi.

iv.) **Mancanza di parallelismo**: La computazione seriale non sfrutta il parallelismo, che è la capacità di eseguire più operazioni contemporaneamente. Questo può limitare la velocità e l'efficienza del calcolo, in particolare quando si lavora con algoritmi o task che potrebbero essere parallelizzati per ottenere prestazioni migliori.

Per superare questi limiti, sono stati sviluppati approcci di calcolo parallelo e distribuito, come l'utilizzo di framework come Apache Spark Questi approccio ci consente di sfruttare al meglio le risorse di calcolo disponibili, ridurre i tempi di esecuzione e migliorare l'efficienza complessiva del calcolo.

7.2 Vantaggi dell'implementazione in Scala/Spark

L'esecuzione di Kfinger con Spark offre numerosi vantaggi, tra cui:

i.) **Scalabilità**: Il calcolo distribuito su un cluster di macchine permette di gestire dataset di grandi dimensioni, consentendo l'analisi di un numero maggiore di sequenze genomiche.

ii.) **Prestazioni migliorate**: L'esecuzione parallela su più nodi di calcolo riduce il tempo di elaborazione complessivo, accelerando il calcolo degli overlap tra le sequenze genomiche.

iii.) **Gestione efficiente delle risorse**: Spark ottimizza l'utilizzo delle risorse di calcolo disponibili nel cluster attraverso diverse tecniche e meccanismi che consentono di massimizzare l'efficienza e l'efficacia del calcolo distribuito (come il partizionamento dei dati, elaborazione in memoria, caching e persistenza dei dati, pianificazione delle attività, tolleranza ai guasti, scheduling dinamico delle risorse, ottimizzazione del trasferimento dei dati) consentendo di sfruttare al meglio la potenza di calcolo a disposizione.

iv.) **Tolleranza ai guasti**: Spark fornisce meccanismi di gestione dei guasti, garantendo che il calcolo possa essere ripreso in caso di malfunzionamenti dei nodi di calcolo.

7.3 Sviluppi futuri

Gli sviluppi futuri nel campo del calcolo degli overlap tra sequenze genomiche utilizzando l'algoritmo Kfinger con Spark potrebbero includere diverse direzioni di ricerca e miglioramenti tecnologici. Ecco alcuni possibili sviluppi futuri:

i.) **Ottimizzazione degli algoritmi**: Gli algoritmi di calcolo degli overlap pssono essere ottimizzati per ridurre ulteriormente il tempo di esecuzione e migliorare l'efficienza computazionale

ii.) **Gestione del volume di dati genomici sempre più grandi**: Gli sviluppi futuri potrebbero concentrarsi sulla gestione efficiente di dataset sempre più grandi, utilizzando il calcolo distribuito su cluster più potenti

iii.) **Miglioramento delle prestazioni di Spark**:

iv.) **Il framework Spark è in continua evoluzione** e gli sviluppi futuri potrebbero portare a miglioramenti delle prestazioni, ottimizzazioni dell'architettura e nuove funzionalità che potrebbero essere sfruttate per ulteriori miglioramenti nell'analisi degli overlap genomici

In conclusione possiamo dire che con il continuo progresso della ricerca in bioinformatica e informatica, ci si aspetta che nuove tecniche e strumenti emergano per affrontare le sfide sempre più complesse e sfruttare appieno il potenziale delle analisi genomiche.

Bibliografia

[1] C. Allocchio, C. Battista, M. Carboni, and L. dell'Agnello. The italian academic network garr: evolution in the gigabit era. *Computer Communications*, 26(5):477–480, 2003. 9

[2] A. Apostolico. Llexicographically minimal strings. pages 283–293, 1993. 6

[3] A. Barysheva. Sparkbwa: Speeding up the alignment of high-throughput dna sequencing data. 2017. 3

[4] M. Bhargavikrishna and S. Jyothi. Distributed computing in big data frame work: A review. *INFORMATION TECHNOLOGY IN INDUSTRY*, 9(3):1049–1077, 2021. 10

[5] P.-A. Binz, J. Shofstahl, J. A. Vizcaíno, H. Barsnes, R. J. Chalkley, G. Menschaert, E. Alpi, K. Clauser, J. K. Eng, L. Lane, et al. Proteomics standards initiative extended fasta format. *Journal of proteome research*, 18(6):2686–2692, 2019. 17

[6] P. Bonizzoni, A. Petescia, Y. Pirola, R. Rizzi, R. Zaccagnino, and R. Zizza. Kfinger: capturing overlaps between long reads by using lyndon fingerprints. In *Bioinformatics and Biomedical Engineering: 9th International Work-Conference, IWBBIO 2022, Maspalomas, Gran Canaria, Spain, June 27–30, 2022, Proceedings, Part II*, pages 436–449. Springer, 2022. 17

[7] D. Brown. Large-scale sequence analysis with kfinger on apache spark. journal of parallel and distributed computing. pages 83–94, 2018. 5

[8] W. H. Chan. Lyndon tree: A space-efficient data structure for representing and indexing huge collections of strings. pages 299–312, 2011. 6

[9] L. Chen. Lyndon-array based compression of genomic data. 2014. 6

[10] M. Crochemore. Lyndon words. pages 71–94, 1985. 6

[11] R. Guo, Y. Zhao, Q. Zou, X. Fang, and S. Peng. Bioinformatics applications on apache spark. *GigaScience*, 7(8):giy098, 2018. 10

[12] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989. 14

[13] A. Jones. K-mer-based sequence analysis toolkit. bioinformatics. pages 403–404, 2012. 5

[14] A. Kala Karun and K. Chitharanjan. A review on hadoop — hdfs infrastructure extensions. In *2013 IEEE Conference on Information & Communication Technologies*, pages 132–137, 2013. 13

[15] H. Karau and R. Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O'Reilly Media, Inc.", 2017. 10

[16] A. Konagaya. Trends in life science grid: from computing grid to knowledge grid. *BMC bioinformatics*, 7(5):1–7, 2006. 9, 10

[17] H. Li. Kmerhash: An efficient approach to support k-mer based sequence analysis. pages 2987–2994, 2016. 5

[18] M. Lothaire. Combinatorics on words. 1997. 6

[19] B. J. Morse, N. L. Gullekson, S. A. Morris, and P. M. Popovich. The development of a general internet attitudes scale. *Computers in Human Behavior*, 27(1):480–489, 2011. Current Research Topics in Cognitive Load Theory. 9

[20] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004. 14

[21] T. Rentsch. Object oriented programming. *ACM Sigplan Notices*, 17(9):51–57, 1982. 14

[22] H. Shah. Sparkalign: A fast and scalable algorithm for distributed sequence alignment. in proceedings of the international conference on parallel processing. pages 1–10, 2016. 5

[23] J. G. Shanahan and L. Dai. Large scale distributed data science using apache spark. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 2323–2324, New York, NY, USA, 2015. Association for Computing Machinery. 10

[24] J. Smith. Kfinger: An algorithm for computing sequence overlaps. pages 553–570, 2010. 5

[25] P. Tader. Server monitoring with zabbix. *Linux Journal*, 2010(195):7, 2010. 15

[26] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery. 12

[27] Y. Wang. Ktree: An efficient approach for k-mer based sequence analysis. pages 2369–2375, 2014. 5

[28] Wikipedia contributors. Inter-process communication — Wikipedia, the free encyclopedia, 2023. [Online; accessed 11-June-2023]. 12

[29] Wikipedia contributors. Lazy evaluation — Wikipedia, the free encyclopedia, 2023. [Online; accessed 11-June-2023]. 11

[30] J. Yang. Sparkblast: Scalable blast with spark. pages 1388–1390, 2018. 3

[31] M. Zaharia. Spark: Cluster computing with working sets. in proceedings of the 2nd usenix conference on hot topics in cloud computing. 2010. 5