Università degli Studi di Salerno
Dipartimento di Informatica

Tesina Del Progetto Di
Sistemi Operativi Avanzati

Implementazione Di Blast Su Un Cluster Spark Attraverso L'Algoritmo Aho-Corasick

Relatore
Prof. Giuseppe Cattaneo

Candidati
Alfonso Ingenito
Carmine Cristian Cruoglio
Luca Giaffreda

Anno Accademico 2020-2021

Introduzione

La genomica è una branca della biologia molecolare che si occupa dello studio del genoma degli organismi viventi. Il termine genoma si riferisce al DNA contenuto in una singola cellula di un organismo. La genomica si basa sulla bioinformatica per cercare di fotografare l'immagine complessiva del genoma, ovvero l'insieme dei geni e delle loro caratteristiche.Le origini e l'evoluzione di questa scienza coincidono quindi con lo sviluppo di tecnologie per l'analisi parallela di un elevato numero di geni in grado di produrre grandi quantità di dati successivamente all'esecuzione di un singolo esperimento.

Il lavoro seguente dimostra lo sviluppo di un'applicazione Spark chiamata ACBlastn che ha lo scopo di ricercare stringhe di lunghezza fissa all'interno di un database testuale attraverso l'utilizzo dell'algoritmo di Aho-Corasick. L'obiettivo successivo è l'utilizzo dell'applicazione realizzata per effettuare dei benchmark dettagliati al fine di valutare lo speedup e dimostrare quanto l'ambiente distribuito sia indispensabile nella manipolazione delle sequenze genomiche.

Il seguente elaborato è suddiviso in cinque sezioni. Nella prima parte si fornirà al lettore un piccolo background teorico in cui verranno discussi temi riguardanti l'analisi delle sequenze genomiche, l'allineamento locale e globale delle sequenze, l'algoritmo Smith-Waterman e l'algoritmo BLAST. Nella seconda parte dell'elaborato si parlerà dell'ambiente operativo descrivendo prima brevemente l'evoluzione del calcolo per la gestione dei Big Data e successivamente il paradigma di programmazione MapReduce e i middleware che lo supportano, quali Apache Hadoop e Spark. La terza parte tratterà dell'implementazione realizzata, introducendo dapprima il linguaggio di programmazione Scala, successivamente descrivendo l'algoritmo Aho-Corasick, su cui si basa l'implementazione, ed infine descrivendo l'architettura di ACBlastn. Nella quarta parte verrà presentata l'analisi dei risultati ottenuti con ACBlastn, inizialmente descrivendo il DAG che mostra il flusso dell'esecuzione attraverso le varie componenti e stage e successivamente mostrando i benchmark utilizzati per dimostrare la scalabilità di ACBlastn ed infine ulteriori test sulle performance di ACBlastn. Seguirà la quinta ed ultima sezione dell'elaborato con le conclusioni finali.

Indice

Capitolo 1

Background Teorico

1.1 Analisi delle Sequenze Genomiche

Il *genoma* è l'insieme di tutte le informazioni biologiche necessarie alla costruzione e al mantenimento di ogni organismo vivente. Nella sequenza del genoma sono contenute tutte le istruzioni per lo sviluppo e il funzionamento dell'organismo e per questo acquista notevole importanza conoscerne la sequenza completa. Il confronto delle sequenze, ovvero la valutazione di quanto due sequenze biologiche siano simili tra loro, è un compito fondamentale e di routine in Biologia Computazionale e Bioinformatica. Tradizionalmente, i metodi *alignment-based* sono lo standard de facto per tale valutazione. Esistono tre tipi di allineamento; *globale*, *locale* e *glocale* (globale + locale). L'allineamento globale tenta di allineare ogni residuo in ogni sequenza. Ciò è utile quando il set di dati ha più somiglianze ed è più o meno uguale in lunghezza. Quando le sequenze hanno meno regioni di somiglianze, viene spesso utilizzato l'allineamento locale. Glocal è un metodo ibrido, serve per cercare il miglior allineamento parziale possibile di due sequenze.

1.2 Allineamento Locale

Il problema dell'allineamento locale tratta il procedimento di trovare ed estrarre, all'interno di due stringhe, delle sottostringhe che presentano una somiglianza elevata. Questo perchè anche se due stringhe non sono molto simili prese completamente, esse possono avere comunque un'area che presenta un'alta "similarità".

Definiamo il problema dell'allineamento locale come:
*Date due stringhe **S1** e **S2**, trovare sottostringhe α e β , rispettivamente di S1 ed S2, la cui somiglianza (valore di allineamento locale ottimo) è massima su tutte le coppie di sottostringhe di S1 e S2. Usiamo v* per denotare il valore di una soluzione ottimale al problema di allineamento locale.*

L'allineamento locale è definito in termini di somiglianza, che massimizza una funzione obiettivo, invece una *edit distance* ha come scopo quello di minimizzare la funzione obiettivo.

Un possibile problema potrebbe verificarsi quando le sottostringhe individuate sono lunghe un singolo carattere e non identificano una regione di elevata somiglianza. Una formulazione come l'allineamento locale, in cui i match contribuiscono positivamente e i gap e gli spazi contribuiscono negativamente, ha una probabilità maggiore di trovare regioni più significative ad alta somiglianza.

L'allineamento locale è considerato il tipo di allineamento più appropriato per confrontare proteine di diverse famiglie. Le coppie di stringhe proteiche con una forte somiglianza locale spesso mostrano un'ampia somiglianza globale. [3]

Un confronto con l'allineamento globale
Perché non cercare regioni di alta somiglianza in due stringhe allineandole prima globalmente? Un allineamento globale tra due stringhe lunghe sarà certamente influenzato da regioni di alta somiglianza e un allineamento globale ottimale potrebbe allineare quelle regioni corrispondenti tra loro. Molto spesso, le regioni in cui sussiste un'alta somiglianza locale andrebbero perse nell'allineamento globale ottimale. Quindi, per identificare un'alta somiglianza locale è più efficace cercare esplicitamente la somiglianza locale.

Mostreremo che se le lunghezze delle stringhe S1 e S2 sono pari ad n e m, allora il problema dell'allineamento locale può essere risolto in tempo O(nm), ovvero lo stesso tempo impiegato dall'allineamento globale. Questa efficienza è sorprendente perché se ci fossero $O(n^2m^2)$ coppie di sottostringhe, ovvero se un allineamento globale potesse essere calcolato in tempo costante per ogni coppia, il tempo limite sarebbe comunque (n^2m^2). Se, ad esempio, usassimo O(kl) per indicare il limite di tempo impiegato per allineare stringhe di lunghezza k e l, allora il problema di allineamento locale avrebbe come limite temporale $O(n^3m^3)$. Il limite di tempo O(nm) è stato ottenuto da Temple Smith e Michael Waterman attraverso il loro algoritmo.

Nella definizione di allineamento locale data in precedenza, qualsiasi schema di punteggio era permesso per l'allineamento globale di due sottostringhe scelte. Una piccola restrizione aiuterà nel calcolo dell'allineamento locale, ovvero, assumiamo che l'allineamento globale di due stringhe vuote abbia valore zero. Questa assunzione viene usata per permettere all'algoritmo di allineamento locale di scegliere due sottostringhe vuote per a e B.

1.3 Smith-Waterman

Terminologia per l'allineamento locale e globale
Nella letteratura biologica, l'allineamento globale (similarity) è spesso indicato con il nome Needleman-Wunsch dagli autori che per primi hanno discusso di questa similarità. In maniera analoga, l'allineamento locale viene spesso indicato come Smith-Waterman. D'altronde, la soluzione data da Needleman-Wunsch funziona in tempo cubico e viene usata raramente, mentre quella data da Smith-Waterman in tempo quadratico ed è comunemente usata.

Uso di Smith-Waterman per trovare diverse regioni ad alta somiglianza
Molto spesso nelle applicazioni biologiche non è sufficiente, a partire dalle stringhe di input di Sl e S2, trovare una sola coppia di sottostringhe con l'allineamento locale ottimale. Piuttosto, bisogna trovare tutte o molte coppie di sottostringhe che hanno somiglianza al di sotto della soglia. Spesso, per trovare coppie aggiuntive di sottostringhe con similarità alta e risolvere il problema di allineamento del suffisso locale si fa uso della tabella di programmazione. L'osservazione chiave è che per qualsiasi cella (i, j) nella tabella, è possibile trovare una coppia di sottostringhe di Sl e S2 con similarità (valore di allineamento globale) pari a v(i, j). Pertanto, un modo semplice per cercare un insieme di sottostringhe altamente simili è trovare un insieme di celle nella tabella con un valore superiore ad una soglia impostata. Questo metodo non garantisce che tutte le stringhe simili vengano identificate ma viene comunque utilizzato in pratica.

1.4 Blast

Rilasciato nel 1990, Blast divenne da subito il motore di ricerca principale per database di sequenze biologiche. Le ragioni del successo sono state la velocità, la gamma di soluzioni date e le statistiche fornite sulle soluzioni. Il focus principale sullo sviluppo di BLAST ha riguardato la ricerca degli hot spots di allineamento e gli strumenti che forniscono le probabilità delle corrispondenze riportate.

BLAST si concentra sulla ricerca di regioni ad alta somiglianza locale con allineamenti senza spazi vuoti, valutati da una matrice di punteggio di peso alfabetico. È possibile, inoltre, creare allineamenti con alcuni gap concatenando diverse regioni localmente simili che BLAST trova.

Definiamo ora alcuni oggetti trattati da BLAST. Date due stringhe Sl e S2:

- Una coppia di segmenti è una coppia di sottostringhe di lunghezza uguale di Sl e S2, allineate senza spazi.

- Una coppia di segmenti massima localmente è una coppia di segmenti il cui punteggio di allineamento (senza spazi) cadrebbe espandendo o accorciando i segmenti su entrambi i lati

- Una coppia di segmenti massimi (MSP) in Sl, S2 è una coppia di segmenti con il punteggio massimo su tutte le coppie di segmenti in SI, S2.

L'algoritmo BLAST cerca di trovare tutte le sequenze che, insieme ad una sequenza di query di dimensione fissa P, contengono un MSP con punteggio superiore a quello di C impostato. La scelta di C è guidata dalle caratteristiche di P e dalle sequenze del database. BLAST segnala anche sequenze che non hanno un MSP superiore a C ma che hanno diverse coppie di segmenti che in combinazione sono statisticamente significative.

La strategia hit (hot-spot) di BLAST
BLAST ha due applicazioni differenti inglobate: BLASTP per i database proteici e BLAST per il DNA. Per trovare sequenze con un MSP superiore a C in un database di proteine, BLASTP usa la seguente strategia: Con una lunghezza fissa w e una soglia fissa t, BLAST trova tutte le sottostringhe di lunghezza-w di S (chiamate "parole" in BLAST) che si allineano a qualche sottostringa di lunghezza-w di P con un punteggio di allineamento superiore a t. Ogni tale hot-spot (chiamato "hit" in BLAST) viene poi esteso per vedere se è contenuto in una coppia di segmenti con punteggio superiore a C.

Una volta che si è verificata una hit, le due sequenze vengono estese per vedere se la hit appartiene ad una coppia di segmenti localmente massima con punteggio di allineamento almeno C. Queste estensioni vengono troncate in anticipo se il punteggio di allineamento in esecuzione è troppo al di sotto del punteggio migliore trovato in precedenza per un'estensione più breve. Ciò significa che BLAST non è ideale per trovare tutte le coppie di segmenti con punteggio almeno C (a livello globale). BLAST inoltre non fa uso della programmazione dinamica per esaminare le estensioni in quanto non consente spazi negli allineamenti.

L'efficacia di BLAST
Le scelte riguardanti la matrice di punteggio, la dimensione di w e di C sono fondamentali per l'efficienza e l'efficacia di BLAST. L'abbassamento di t riduce la possibilità che una sequenza con un punteggio MSP superiore a C venga persa, ma aumenta la quantità di calcolo richiesta. Queste scelte sono studiate empiricamente e il valore consigliato di w è tra tre e cinque per gli amminoacidi e di circa dodici per il DNA.

Capitolo 2

Ambiente Operativo

Qui verrà riassunto prima una breve introduzione ai sistemi centralizzati e successivamente si parlerà dell'evoluzione del calcolo distribuito. Seguiranno alcuni dettagli sul paradigma di programmazione MapReduce, i framework Apache Hadoop, Apache Spark ed infine sarà descritta la struttura dati utilizzata da quest'ultimo.

2.1 L'Evoluzione del Calcolo per la Gestione dei Big Data

Sistemi centralizzati
I sistemi centralizzati utilizzano un'architettura client/server in cui uno o più nodi client sono collegati direttamente a un server centrale. Gli aspetti positivi di questa tipologia di sistema sono: la facilità di gestione e risorse dedicate e convenienza economica per lo sviluppo di sistemi di piccole dimensioni. A tutti questi benefici corrispondono però anche degli svantaggi. È altamente dipendente dalla connettività di rete. Vi è meno possibilità di backup dei dati. La manutenzione del server si rivela difficile in quanto renderebbe inaccessibile il servizio per tutta la sua durata. Inoltre, vi è solo la possibilità di aumentare verticalmente le prestazioni entro un certo limite.

Sistemi distribuiti
Un sistema distribuito consiste di un insieme di macchine, ognuna gestita in maniera autonoma, connesse attraverso una rete. Ogni nodo del sistema distribuito esegue un insieme di componenti che comunicano e coordinano il proprio lavoro attraverso uno strato software detto *middleware*, in maniera che l'utente (utente del sistema ma anche, in maniera limitata, il programmatore e progettista) percepisca il sistema come un'unica entità integrata. I sistemi distribuiti vengono impiegati per permettere di utilizzarne appieno le risorse di calcolo, per assicurare l'alta disponibilità dei servizi offerti, le prestazioni, l'affidabilità e l'ottimizzazione dei costi.

Il middleware Il middleware è un software che fornisce alle applicazioni servizi e capacità frequentemente utilizzati, tra cui gestione dei dati e delle API, servizi per le applicazioni, messaggistica e autenticazione. Inoltre, aiuta gli sviluppatori a creare le applicazioni in modo più efficiente e agisce come un tessuto connettivo tra applicazioni, dati e utenti. Rende conveniente lo sviluppo, l'esecuzione e la scalabilità di applicazioni alle organizzazioni con ambienti multicloud e containerizzati. L'obiettivo del middleware è quello di assicurare, innanzitutto, che il programmatore di applicazioni concentri i propri sforzi sullo sviluppo della logica di business dell'applicazione e che non si debba interessare direttamente dei dettagli di comunicazione. Infine, proprio perché il middleware astrae significativamente la parte di distribuzione e di servizio delle applicazioni distribuite, lo sviluppo avviene ad alto livello, permettendo di poter utilizzare e riutilizzare framework e soluzioni.

2.2 Il paradigma di programmazione MapReduce e i middleware che lo supportano

2.2.1 Il paradigma

MapReduce è un paradigma per l'elaborazione di grandi quantità di dati su un'infrastruttura informatica distribuita. Supponendo che i dati di input siano organizzati come un insieme di coppie $< chiave, valore >$, si basa sulla definizione di due funzioni. La funzione *map* elabora una coppia di input $< chiave, valore >$ e restituisce un insieme intermedio (possibilmente vuoto) di coppie $< chiave, valore >$. La funzione di *reduce* unisce tutti i valori intermedi che condividono la stessa chiave per formare un insieme (possibilmente più piccolo) di valori. Queste funzioni vengono eseguite, come attività, sui nodi di un cluster di elaborazione distribuito. Tutte le attività relative alla gestione del ciclo di vita di questi compiti, così come la raccolta dei risultati della funzione map e la loro trasmissione alle funzioni di reduce, sono gestite in modo trasparente dal quadro sottostante (parallelismo implicito), senza oneri per il programmatore.

2.2.2 Apache Hadoop

Apache Hadoop è il framework più popolare che supporta il paradigma MapReduce. Permette l'esecuzione di calcoli distribuiti grazie all'interazione di due componenti architetturali: *YARN* (Yet Another Resource Negotiator) e *HDFS* (Hadoop Distributed File System). YARN gestisce il ciclo di vita di un'applicazione distribuita tenendo traccia delle risorse disponibili su un cluster di elaborazione e allocandole per l'esecuzione di attività applicative. HDFS è un file system distribuito e strutturato a blocchi progettato per essere eseguito su hardware di base e in grado di fornire tolleranza di errore attraverso la replica dei dati. Un cluster Hadoop di base è composto da un singolo nodo master e più nodi slave. Il nodo master arbitra l'assegnazione delle risorse di calcolo alle applicazioni da eseguire sul cluster e

mantiene un indice di tutte le directory e dei file archiviati nel file system distribuito HDFS. Inoltre, traccia fisicamente i nodi slave, memorizzando i blocchi di dati che compongono questi file. I nodi slave, a loro volta, ospitano una serie di lavoratori (chiamati anche container), incaricati di eseguire attività map e reduce, oltre ad utilizzare l'archiviazione locale per mantenere un sottoinsieme dei blocchi di dati HDFS. Una delle caratteristiche principali di Hadoop è la sua capacità di sfruttare il calcolo locale dei dati. Questo termine si riferisce alla possibilità di avvicinare le attività ai dati su cui devono operare (piuttosto che il contrario). Ciò consente di ridurre notevolmente la congestione della rete e aumentare la velocità complessiva del sistema durante l'elaborazione di grandi quantità di dati. Inoltre, al fine di mantenere i file in modo affidabile e bilanciare correttamente il carico tra diversi nodi di un cluster, i file di grandi dimensioni vengono automaticamente suddivisi in blocchi più piccoli, replicati e distribuiti su nodi diversi.

2.2.3 Apache Spark

Spark è un sistema distribuito veloce e generale per l'elaborazione di big data su un cluster. Fornisce API di alto livello che supportano più lingue come Scala, Java, Python, SQL ed R. È costituito da due blocchi principali: un modello di programmazione che crea un grafico delle dipendenze (DAG) e un sistema di runtime ottimizzato che utilizza questo grafico per pianificare le unità di lavoro in un cluster e trasporta anche codice e dati nei nodi di lavoro in cui verranno elaborati dai processi degli esecutori.

2.2.4 Resilient Distributed Datasets

Il fulcro del modello di programmazione Spark è l'astrazione Resilient Distributed Dataset (RDD), una struttura di dati parallela tollerante ai guasti che può essere creata e manipolata usando un ricco set di operatori. I programmatori iniziano definendo uno o più RDD attraverso trasformazioni di dati che risiedono originariamente su storage stabile o altri RDD. Esempio di trasformazioni sono operazioni come map, filter o reduce e sono lazy, nel senso che restituiscono un nuovo RDD che dipende dal vecchio ma non vengono eseguite fino a quando nel grafico della pipeline non viene specificata un'azione che coinvolge tale RDD. Le azioni sono operazioni che restituiscono un valore all'applicazione ed esportano i dati in un sistema di archiviazione. Oltre al gestore cluster interno, le applicazioni Spark possono essere eseguite anche su gestori cluster esterni come Hadoop YARN o Mesos. Inoltre, un'applicazione Spark può essere eseguita su un file system distribuito, ad esempio HDFS. Ciò consente a ciascun nodo di lavoro di un cluster di leggere i dati di input e di scrivere i dati di output utilizzando un disco locale anziché un file server remoto.

2.2.5 Partitions, parallelism, and shuffling

Per impostazione predefinita, Spark tenta di leggere i dati in un RDD dai nodi vicini. Poiché Spark di solito accede ai dati distribuiti, per ottimizzare le operazioni di trasformazione crea partizioni per contenere i blocchi di dati. Il numero di partizioni di un RDD riflette il grado di parallelismo (numero di compiti) impiegato da Spark durante l'elaborazione. Quando viene creato un RDD caricando file da HDFS, il suo numero di partizioni è uguale al numero di suddivisioni di input del file originale su HDFS. Il meccanismo Spark per la ridistribuzione dei dati tra le partizioni è chiamato shuffle. Si verifica quando determinate trasformazioni, come groupByKey o reduceByKey, vengono emesse su un RDD e causano lo spostamento di dati attraverso processi diversi o via cavo (tra gli esecutori su macchine separate). Un RDD ottenuto tramite una trasformazione shuffle di un altro RDD erediterà il suo numero di partizioni. Tuttavia, per quanto riguarda la scelta di un "buon" numero di partizioni, ciò che si desidera in genere è avere almeno tante partizioni quanti sono i core. Un modo per influenzare questa scelta è specificare un valore personalizzato per la proprietà spark.default.parallelism. Un'altra opzione è l'introduzione di un partizionatore personalizzato. I partizionatori sono oggetti che definiscono il modo in cui gli elementi di un RDD chiave-valore sono partizionati per chiave. Il partizionatore predefinito Spark (ovvero HashPartitioner) sceglie la partizione dove mappare un elemento come valore Object.hashCode di Java della sua chiave (modulo numero di partizioni) o 0 per hash negativi. È anche possibile implementare una classe di partizionatore personalizzata che definisce un metodo personalizzato per assegnare le chiavi alle partizioni. Questa funzione è utile quando per qualche motivo i dati RDD devono essere distribuiti in modo non uniforme tra le partizioni.

Capitolo 3

Implementazione

Il terzo capitolo tratterà dell'ambiente di sviluppo, descrivendo nel dettaglio il linguaggio di programmazione scelto, l'algoritmo Aho-Corasick e la soluzione adottata per l'implementazione dell'algoritmo per la ricerca di regioni ad alta somiglianza locale.

3.1 Il linguaggio di programmazione Scala

Scala è un linguaggio di programmazione che ha caratteristiche sia orientate agli oggetti che funzionali. È staticamente tipato, con un sistema di tipi avanzato più potente di quello di altri linguaggi come Java e C#. Il suo nome deriva da scalable language: scalabile, nel senso che il core dei costrutti del linguaggio è così modulare che nuove funzionalità possono essere aggiunte tramite librerie invece che modificando il linguaggio stesso. Scala è adatto per la costruzione di sistemi modulari di grandi dimensioni così come per compiti di scripting. Le caratteristiche funzionali ne facilitano l'uso per la scrittura di programmi concorrenti che fanno uso di più core o processori. Scala compila in bytecode Java che gira sulla JVM. Ha un interoperabilità ottima con Java in quanto le librerie di quest'ultimo possono essere utilizzate senza problemi da Scala.

3.2 Algoritmo Aho–Corasick

L'algoritmo *Aho-Corasick* ricerca gli elementi di un insieme di stringhe all'interno di un testo input costruendo una macchina a stati finiti che ricorda un Trie (o albero dei prefissi) con collegamenti aggiuntivi tra i vari nodi interni. I collegamenti aggiuntivi consentono delle transizioni veloci tra le corrispondenze di stringhe non riuscite, ad altri rami del Trie che hanno però un prefisso comune.
La complessità dell'algoritmo è pari alla lunghezza delle stringhe **n** più la lunghezza del testo cercato **m** più il numero di corrispondenze di output **z**.

$$O(n + m + z)$$

NB. Poiché vengono trovate tutte le corrispondenze, può esserci un numero quadratico di corrispondenze se ogni sottostringa corrisponde.

Il grafo utilizzato dall'algoritmo è la struttura dati Aho-Corasick costruita a partire dal dizionario specificato. Viene utilizzata una tabella in cui ogni riga rappresenta un nodo nel Trie, ed ha una colonna che indica il percorso, ovvero la sequenza (unica) di caratteri dalla radice al nodo.

Dictionary {a, ab, bab, bc, bca, c, caa}

Path	In dictionary	Suffix link	Dict suffix link
()	–		
(a)	+	()	
(ab)	+	(b)	
(b)	–	()	
(ba)	–	(a)	(a)
(bab)	+	(ab)	(ab)
(bc)	+	(c)	(c)
(bca)	+	(ca)	(a)
(c)	+	()	
(ca)	–	(a)	(a)
(caa)	+	(a)	(a)

Figura 3.1: Tabella algoritmo Aho-Corasick

La struttura dati ha un nodo per ogni prefisso di ogni stringa nel dizionario. All'interno del grafo, per ogni nodo c'è un arco diretto "figlio" da ogni nodo a un nodo il cui nome si trova aggiungendo un carattere. C'è inoltre un arco diretto "suffisso" da ciascun nodo al nodo che è il suffisso rigorosamente più lungo possibile nel grafico. Gli archi suffisso possono essere calcolati in tempo lineare eseguendo una ricerca in ampiezza a partire dalla radice.

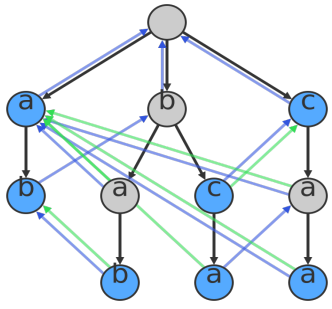

Figura 3.2: I collegamenti dei suffissi sono in blu; quelli del dizionario in verde. I nodi corrispondenti alle voci del dizionario sono evidenziati in blu.

L'obiettivo per l'arco suffisso di un nodo visitato può essere trovato seguendo l'arco suffisso del suo genitore fino al suo nodo suffisso più lungo e cercando un figlio del nodo suffisso il cui carattere corrisponda a quello del nodo visitato.
Se il carattere non esiste come figlio, possiamo trovare il successivo suffisso più lungo (seguendo nuovamente l'arco suffisso) e quindi cercare il carattere. Possiamo farlo finché non troviamo il carattere (come figlio di un nodo) o raggiungiamo la radice.

C'è un arco "suffisso dizionario" da ogni nodo al nodo successivo nel dizionario che può essere raggiunto seguendo gli archi suffisso.
Gli archi suffisso dizionario possono essere calcolati in tempo lineare attraversando ripetutamente gli archi suffisso fino a trovare un nodo che è presente nel dizionario e memorizzando queste informazioni.

Ad ogni passaggio, il nodo corrente viene esteso trovando il suo figlio, e se questo non esiste, trovando il figlio del suo suffisso, e se questo non funziona, trovando il figlio del suffisso del suo suffisso, e così via, finendo infine nella radice nodo se non si è visto nulla prima.
Quando l'algoritmo raggiunge un nodo, emette tutte le voci del dizionario che terminano nella posizione del carattere corrente nel testo di input. Questo viene fatto stampando ogni nodo raggiunto seguendo i collegamenti del suffisso del dizionario, a partire da quel nodo e continuando fino a raggiungere un nodo senza collegamento del suffisso del dizionario. Inoltre, viene stampato il nodo stesso, se è una voce del dizionario.

3.3 Algoritmo ACBlastn

L'applicazione ACBlastn, costituita da due funzioni ed il main, ha lo scopo di ricercare stringhe di lunghezza fissa all'interno di un database testuale attraverso l'utilizzo dell'algoritmo di Aho-Corasick. Di seguito verrà spiegato come è stato costruito il file utilizzato per effettuare le query sul database di nucleotidi. Sucessivamente le funzioni saranno esposte nel dettaglio spiegandone il funzionamento.

Costruzione del file query.txt
La creazione del file **query.txt**, composto da stringhe di lunghezza k su ogni record, avviene tramite l'applicazione multithread *KMC* sul file multisequenza *Picea_glauca.fasta*. KMC è una utility progettata per contare k-mers da insiemi di sequenze genomiche dette reads, esso è uno dei migliori tool in circolazione in quanto è sia molto veloce che frugale nell'utilizzo delle risorse. Un k-mer non è altro che la suddivisione di una sequenza in sottosequenze di lunghezza k. Più in generale, una sequenza di lunghezza L avrà $L - k + 1$ k-mers e n^k k-mers totali possibili, dove n è il numero di monomeri possibili. Quando tutte le sequenze del genoma saranno state analizzate, il programma restituirà un file contenente tutti i k-mer trovati in maniera univoca. Siccome l'output di KMC contiene anche il numero di occorren-

ze del k-mer, esso è stato rielaborato attraverso l'utilizzo della *spark-shell* ed in particolare andando a selezionare l'elemento della tupla (in Spark) corrispondente al k-mer.

deleteCR()

La seguente funzione permette la rimozione del carattere di *newline* dal file del database *Picea_abies.fasta.long*. Viene applicata all'RDD del database ed utilizzata all'interno del main(). Dichiariamo un StringBuffer che ci servirà come contenitore; Per ogni blocco di partizione effettuiamo una *substring* da 0 a $read.length - 1$ (per togliere il carattere corrispondente a newline) e concateniamo il risultato nella variabile str. Successivamente salviamo il risultato in un Array[String] e lo restituiamo sotto forma di Iterator.

Listing 3.1: deleteCR()

```
def deleteCR(block:Iterator[String]):Iterator[String] = {
  val str:StringBuffer = new StringBuffer("")
  while (block.hasNext) {
    val read = block.next
    str.append(read.substring(0, read.length -1))
  }
  val array:Array[String] = Array[String] (str.toString)
  array.toIterator
}
```

searchPartitions()

La funzione *searchPartitions()* è utilizzata per ricercare in ogni partizione del database, una partizione del file delle query. Essa prende in input tre variabili: l'indice della partizione **index**, il blocco della partizione **block** e la partizione del file delle query **queryPart** sotto forma di Array[String].

Listing 3.2: searchPartitions()

```
def searchPartitions (index:Int, block:Iterator[String], queryPart:
   Array[String]):Iterator[(String, Int)] = {
  val array:ArrayBuffer[(String, String)] = ArrayBuffer()
  val ac = new Automaton
  ac.setFailTransitions()
  queryPart.foreach(ac.addWord("query", _))
  while (block.hasNext)
    array ++= ac.search(block.next)
  array.map(x => (x._1, 1)).toIterator
}
```

Inizialmente viene dichiarato un ArrayBuffer di String che conterrà le query ed il rispettivo

numero di occorrenze trovate all'interno del database e viene inizializzata la classe Automaton per poter utilizzare i metodi di aggiunta *addWord()* e ricerca *search()* delle query. Per creare la struttura Aho-Corasick sul blocco delle query andiamo a scorrere l'Array *queryPart* e applichiamo ad ogni elemento la funzione addWord(). L'operazione successiva è quella di accedere e analizzare il blocco di dati contenuto nella struttura Iterator. Un Iterator è un modo per accedere singolarmente agli elementi di un insieme. Esso non carica l'intera raccolta nella memoria ma i singoli elementi uno dopo l'altro. Pertanto, gli iterator, sono utili quando i dati sono troppo grandi per la memoria. Per accedere agli elementi e/o controllare se ci sono elementi disponibili possiamo usare hasNext, mentre next per stampare l'elemento successivo. Ogni elemento sarà passato come input alla funzione *search()* la quale restituisce come output una tupla (String, String) composta dalla query individuata e dal nome del blocco di query: in questo caso "query". Gli elementi dell'array risultante saranno poi opportunamente modificati per restituire la tupla (query, n. occorrenza).

main()

All'interno della funzione Main memorizziamo i parametri passati dall'utente su linea di comando al momento dell'esecuzione dell'applicazione. Tali parametri sono: il file del *db* dove avviene la ricerca delle query e il file delle *query*.

Listing 3.3: main()

```
def main(args:Array[String]):Unit = {
  val spark = SparkSession.builder.
    appName("ACBlastn").
    getOrCreate
  val sc = spark.sparkContext

  // load query file
  val query = sc.textFile(args(2)).collect
  // load db file
  val db = sc.textFile(args(1), sc.defaultParallelism)

  val dbFile = db.mapPartitions(deleteCR)

  val dbCache = dbFile.cache()

  var count = 0
  var tmpSize = 0
  val blockSize = 1024 * 1024 * 10
  while (count < query.size && tmpSize <= blockSize) {
    tmpSize += (query(count).getBytes.length + 1)
    count = count + 1
  }
```

```
23
24    var i = 0
25    val parts = query.grouped(count)
26    while (parts.hasNext) {
27      val part = parts.next
28      val tmp = dbCache.
29        mapPartitionsWithIndex(searchPartitions(_, _, part))
30      tmp.reduceByKey(_+_).
31        saveAsTextFile(s"resultQueryPart-${i}")
32      i = i + 1
33    }
34
35    spark.stop
36  }
```

È possibile controllare l'applicazione Spark tramite un processo del driver chiamato SparkSession. Una volta istanziato l'oggetto verrà assegnato il nome dell'applicazione tramite il metodo appName. L'istanza SparkSession è il modo in cui Spark esegue le manipolazioni definite dall'utente nel cluster. Esiste una corrispondenza uno a uno tra SparkSession e un'applicazione Spark. Tramite SparkSession dichiariamo una variabile SparkContext la quale ci servirà come punto di ingresso per le funzionalità API di basso livello.

Attraverso SparkContext è possibile utilizzare il metodo *textFile* per leggere gli input da HDFS. All'RDD corrispondente al file delle query viene applicata la funzione collect per trasformarlo in un Array[String] così da poterlo partizionare ed utilizzare indipendentemente dalla schedulazione dei task effettuata dal DAG. Al file del database viene assegnato un numero minimo di partizioni uguali al numero di core disponibili. Successivamente viene applicata la funzione *deleteCR()* descritta in precedenza e l'RDD risultante viene salvato in cache poiché non dovrà più subire modifiche durante tutto il corso dell'esecuzione.

L'operazione di partizionamento del file delle query è strutturata in modo da conteggiare il numero di record fino ad un massimo di 10MB: valore medio per cui la creazione della struttura Aho-Corasick associata non causa OutOfMemory. Questo valore *count* sarà passato come input alla funzione *grouped* la quale effettivamente andrà a selezionare esattamente il numero di record indicato.

Attraverso il ciclo while andiamo a scorrere le partizioni create che saranno passate come input alla funzione *searchPartitions* descritta in precedenza insieme al numero di partizione e al relativo blocco del database. Il ciclo si conclude salvando gli output intermedi su HDFS. Essenzialmente *ogni partizione delle query* viene ricercata all'interno di *tutte le partizioni del database*.

Capitolo 4

Analisi delle risultati

In primo luogo, verrà data una breve nozione del Directed Acyclic Graph, illustrando poi il DAG che l'applicazione realizza. Verrà poi effettuata un'analisi sui risultati prodotti, in particolare sarà mostrata la scalabilità dell'applicazione presentando gli esiti delle esecuzioni, in funzione del numero di nodi utilizzati. Successivamente, verranno mostrati i risultati in funzione della taglia dell'input, in questo modo sarà possibile comparare le prestazioni dell'applicazione Spark con l'applicazione multithread BLASTN.

4.1 Directed Acyclic Graph (DAG)

Il DAG in Apache Spark è un insieme finito di vertici e archi, dove i vertici rappresentano gli RDD e gli archi rappresentano l'operazione da applicare su un RDD. Contiene una sequenza di vertici tale che ogni arco è diretto dal primo al successivo nella sequenza. Apache Spark DAG consente all'utente di immergersi nello stage ed espandere i dettagli su qualsiasi di esso. Nella visualizzazione dello stage, i dettagli di tutti gli RDD appartenenti a quello stage vengono espansi. Lo Scheduler divide Spark RDD in fasi basate su varie trasformazioni applicate. Ogni fase è composta da attività, basate sulle partizioni dell'RDD, che eseguiranno lo stesso calcolo in parallelo. Il primo livello è l'interprete, Spark utilizza un interprete Scala, con alcune modifiche.
I seguenti punti spiegheranno il suo funzionamento:

1. Quando si immette il codice in Spark Console (creando RDD e applicando operatori), Spark crea un grafico operatore.

2. Quando l'utente esegue un'azione, il grafico viene inviato a un *DAG Scheduler*. Lo scheduler del DAG divide il grafico dell'operatore in fasi (map e reduce).

3. Una fase è composta da attività basate su partizioni dei dati di input. Lo scheduler del DAG riunisce gli operatori per ottimizzare il grafico. Ad esempio, molti operatori di map possono essere programmati in un'unica fase. Questa ottimizzazione è la chiave per le prestazioni di Spark. Il risultato finale di uno scheduler del DAG è un insieme di fasi.

4. Le fasi vengono passate all'utilità di pianificazione, essa non conosce le dipendenze tra le fasi, avviando semplicemente le attività tramite il gestore cluster.

5. Viene avviata una nuova JVM per executor, e il lavoratore eseguirà soltanto i compiti che gli vengono passati.

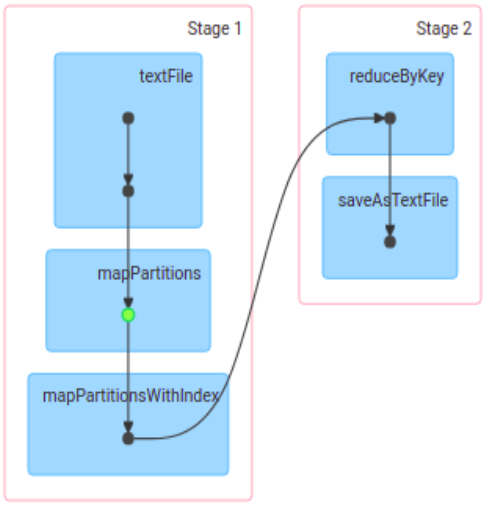

Figura 4.1: DAG dell'applicazione.

4.2 Scalabilità

La *scalabilità* è la capacità di un software o di un hardware di adattarsi a un aumento di domanda o di carico di lavoro. Dunque indica se un sistema è portato a crescere o meno. Distinguiamo due tipi di scalabilità:

Weak scalability. Utilizzata per calcolare come cala l'efficienza della soluzione all'aumentare del numero dei processori. Per fare ciò il carico del lavoro aumenta con l'aumentare dei processori. Tale aumento viene fatto in modo proporzionale, così facendo il lavoro affidato al singolo processore sarà costante.

Strong scalability. Utilizzata per calcolare come cala l'efficienza del problema quando aumenta il numero dei processori, però con il carico di lavoro che non cresce. Quindi lo stesso carico di lavoro viene diviso sui vari processori.

Diamo adesso dimostrazione della scalabilità di questa applicazione, mostrando l'esecuzione in funzione dei nodi del cluster (1, 2, 4, 8). La taglia dell'input del file del *database* sarà di 2 GB, mentre 85 MB (5 milioni di record da 16 caratteri ciascuno) per la dimensione del file delle *query*. Saranno illustrati due immagini per esecuzione: *overview* per mostrare il tempo impiegato e le risorse allocate e la vista *Ganglia* uno strumento di monitoraggio distribuito e scalabile, che illustrerà le prestazioni del cluster. Infine seguirà una spiegazione sui risultati ottenuti.

Esecuzione con un nodo

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Oct 05 18:56:01 +0200 2021
Elapsed:	16mins, 20sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	25922890 MB-seconds, 1954 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.2: Un nodo: Application Overview.

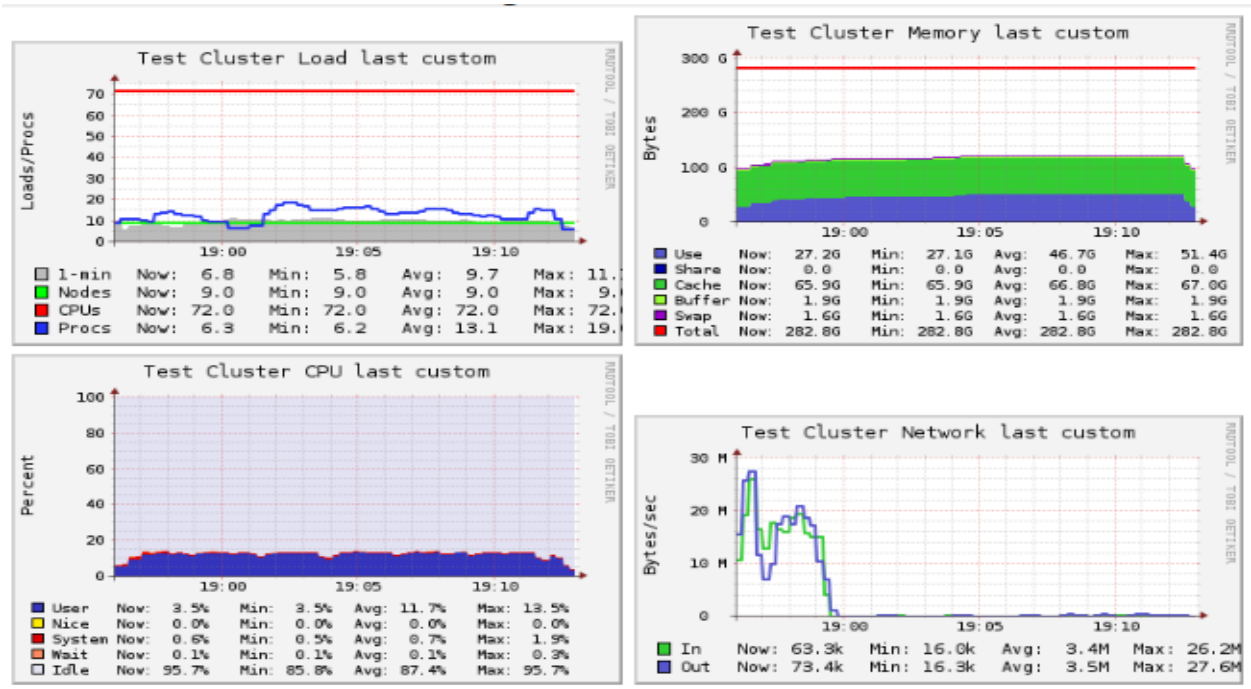

Figura 4.3: Un nodo: Ganglia.

Esecuzione con due nodi

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Oct 05 18:44:26 +0200 2021
Elapsed:	9mins, 47sec
Tracking URL:	History
Log Aggregation Status:	NOT_START
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	30277173 MB-seconds, 1746 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.4: Due nodi: Application Overview.

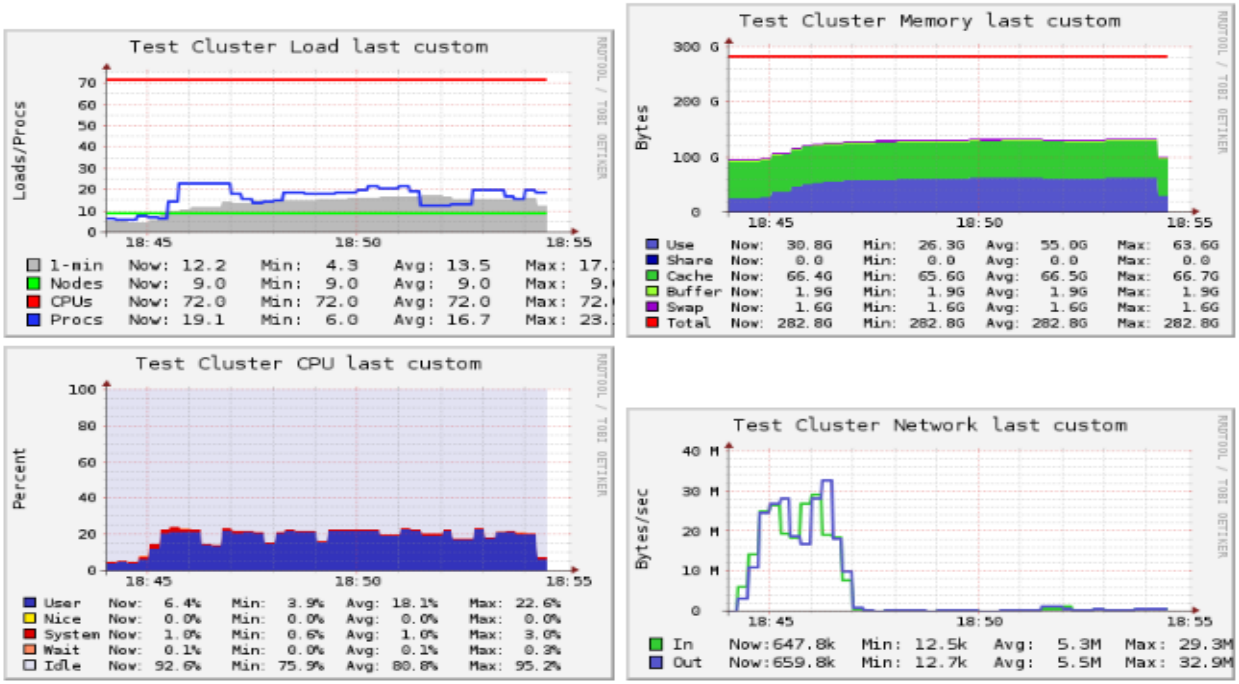

Figura 4.5: Due nodi: Ganglia.

Esecuzione con quattro nodi

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Oct 05 18:18:48 +0200 2021
Elapsed:	6mins, 3sec
Tracking URL:	History
Log Aggregation Status:	NOT_START
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	36796396 MB-seconds, 1783 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.6: Quattro nodi: Application Overview.

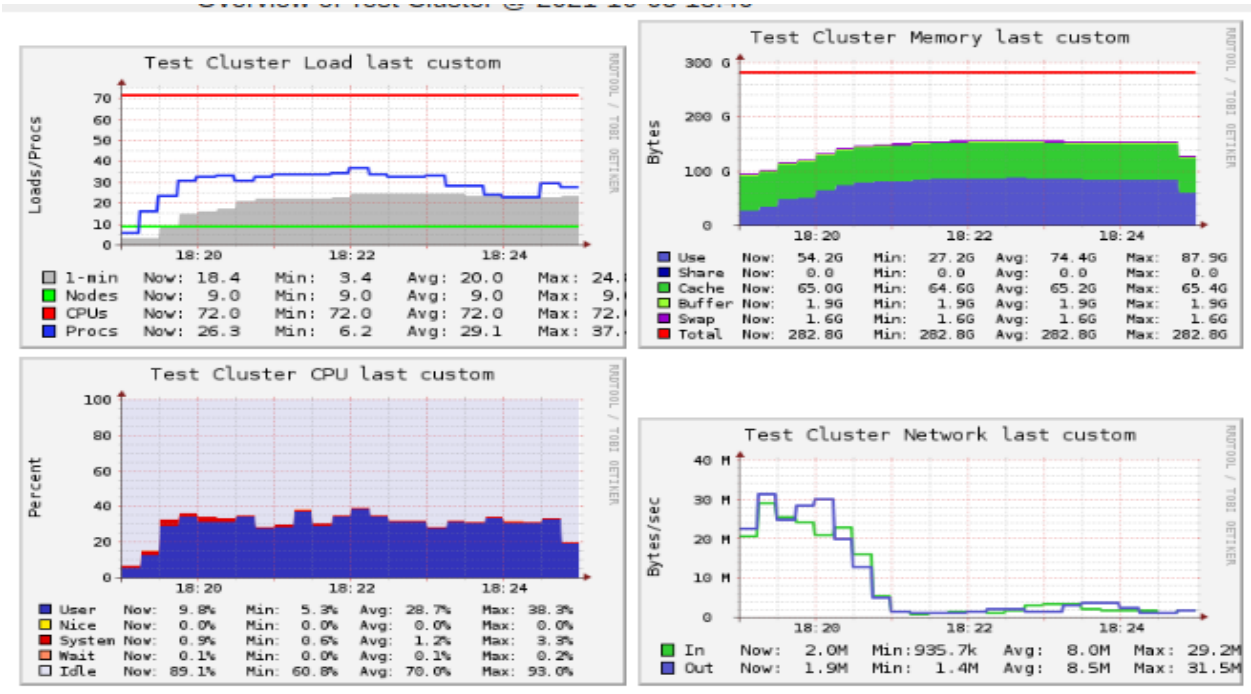

Figura 4.7: Quattro nodi: Ganglia.

Esecuzione con otto nodi

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Tue Oct 05 17:59:20 +0200 2021
Elapsed:	4mins, 17sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	51551368 MB-seconds, 2258 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.8: Otto nodi: Application Overview.

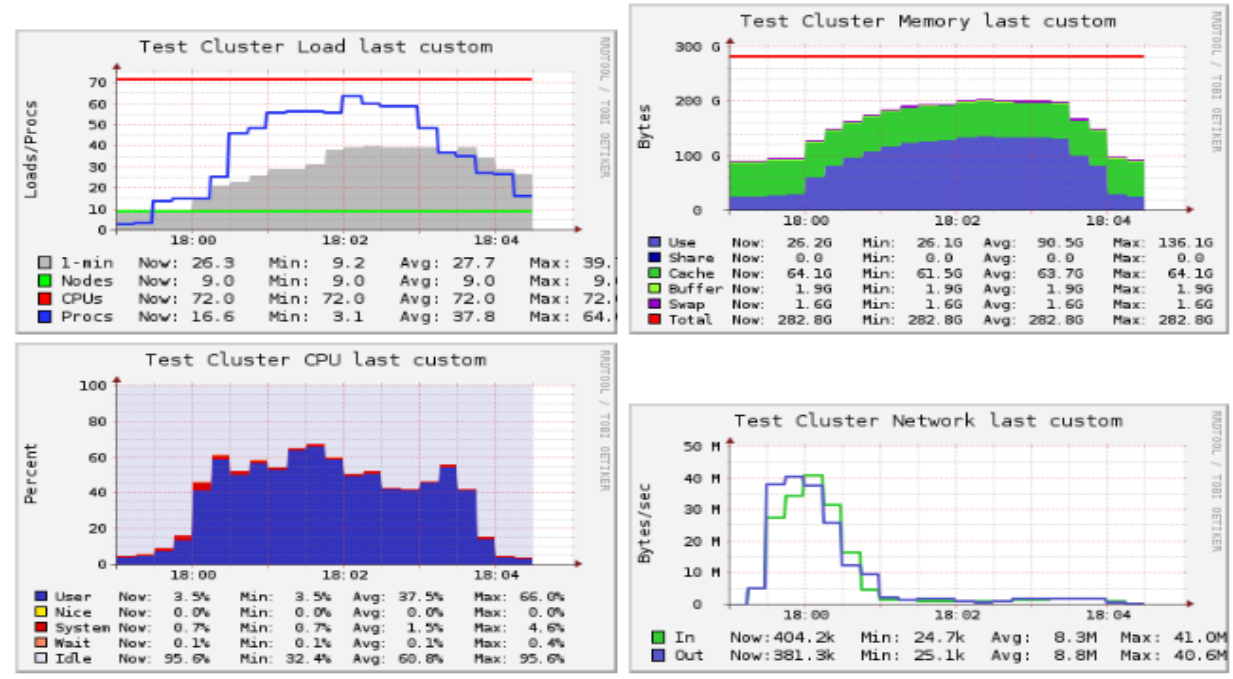

Figura 4.9: Otto nodi: Ganglia.

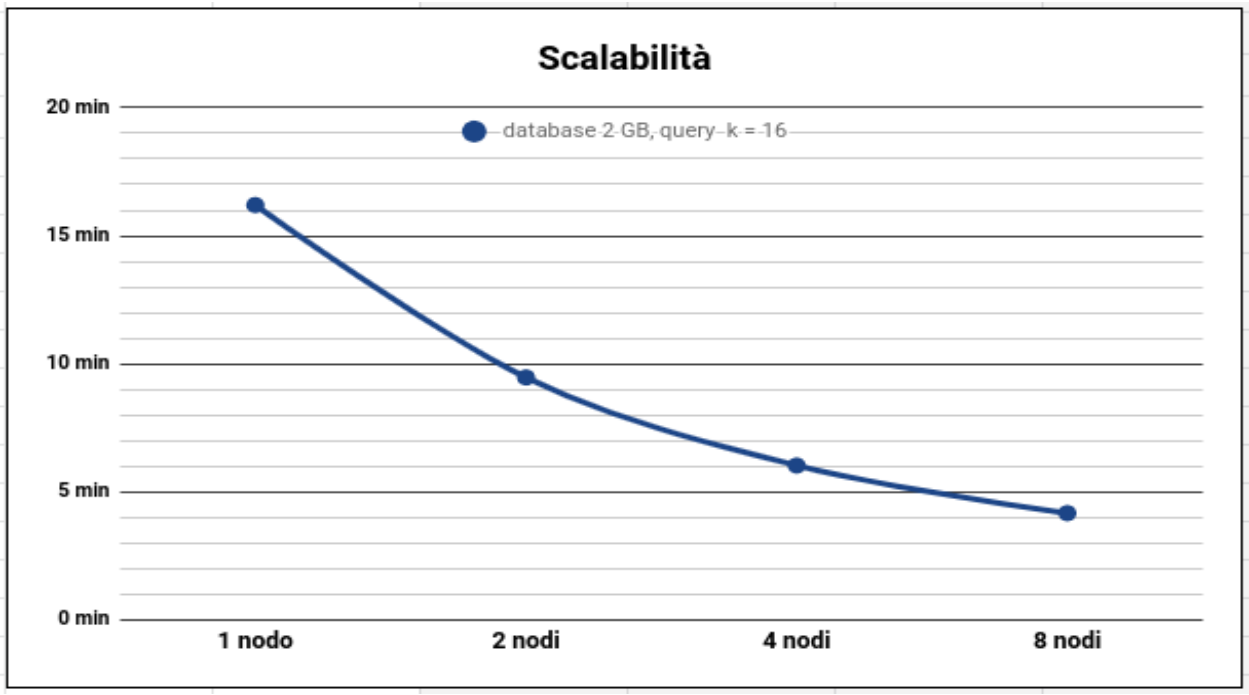

Figura 4.10: In funzione del numero di nodi aggiunti, si può subito notare una notevole diminuzione dei tempi di esecuzione di almeno il 50% introducendo il parallelismo.

4.3 Risultati prodotti: ACBlastn.scala

Di seguito saranno mostrati ulteriori benchmark dove a differenza di quelli visti in precedenza il numero dei nodi rimane invariato mentre il file delle query e il database variano. In particolare, per ogni esecuzione dei diversi input (2GB, 4GB e 8GB), query.txt sarà rispettivamente di dimensioni pari a 85 MB, 160 MB e 310 MB (cioè 5 milioni di record di lunghezza 16, 32, 64 caratteri). Saranno illustrati due immagini per esecuzione: *overview* per mostrare il tempo impiegato e le risorse allocate e la vista *Ganglia* uno strumento di monitoraggio distribuito e scalabile, che illustrerà le prestazioni del cluster. Infine saranno mostrati due istogrammi per chiarire i risultati ottenuti in funzione del tempo e della memoria utilizzata.

4.3.1 Database 2GB

5 milioni di record da 16 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 20:43:36 +0200 2021
Elapsed:	4mins, 13sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	50858878 MB-seconds, 2230 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.11: Application Overview: db 2GB, query 81 MB, k = 16.

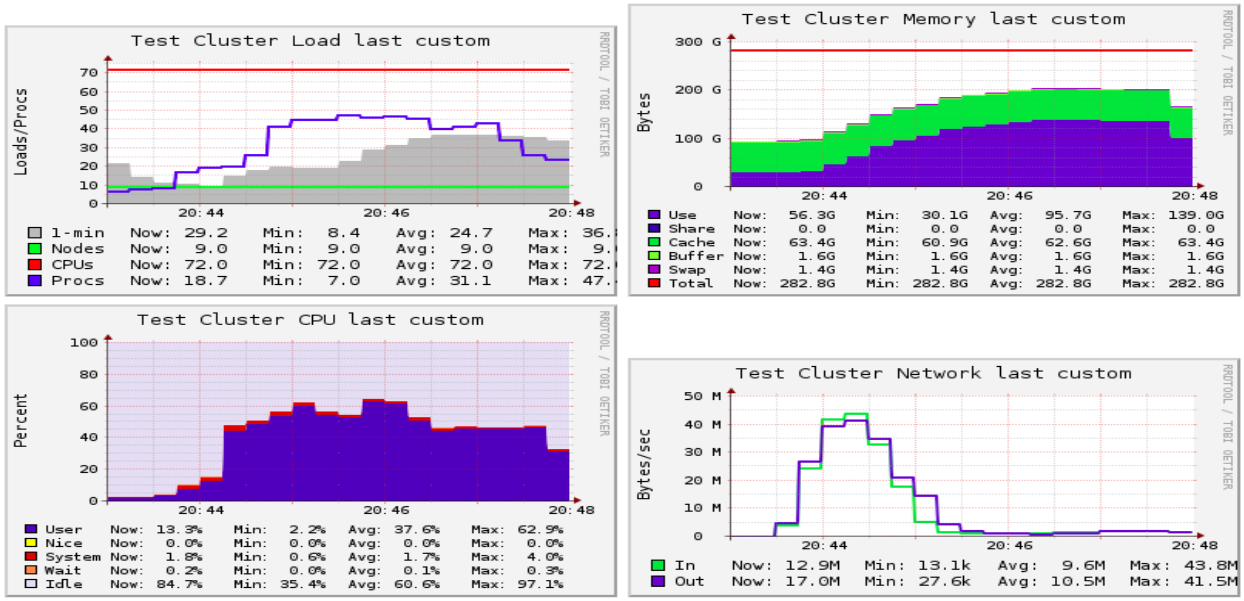

Figura 4.12: Ganglia: db 2GB, query 81 MB, k = 16.

5 milioni di record da 32 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 20:48:04 +0200 2021
Elapsed:	12mins, 37sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	154523602 MB-seconds, 6758 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.13: Application Overview: db 2GB, query 157 MB, k = 32.

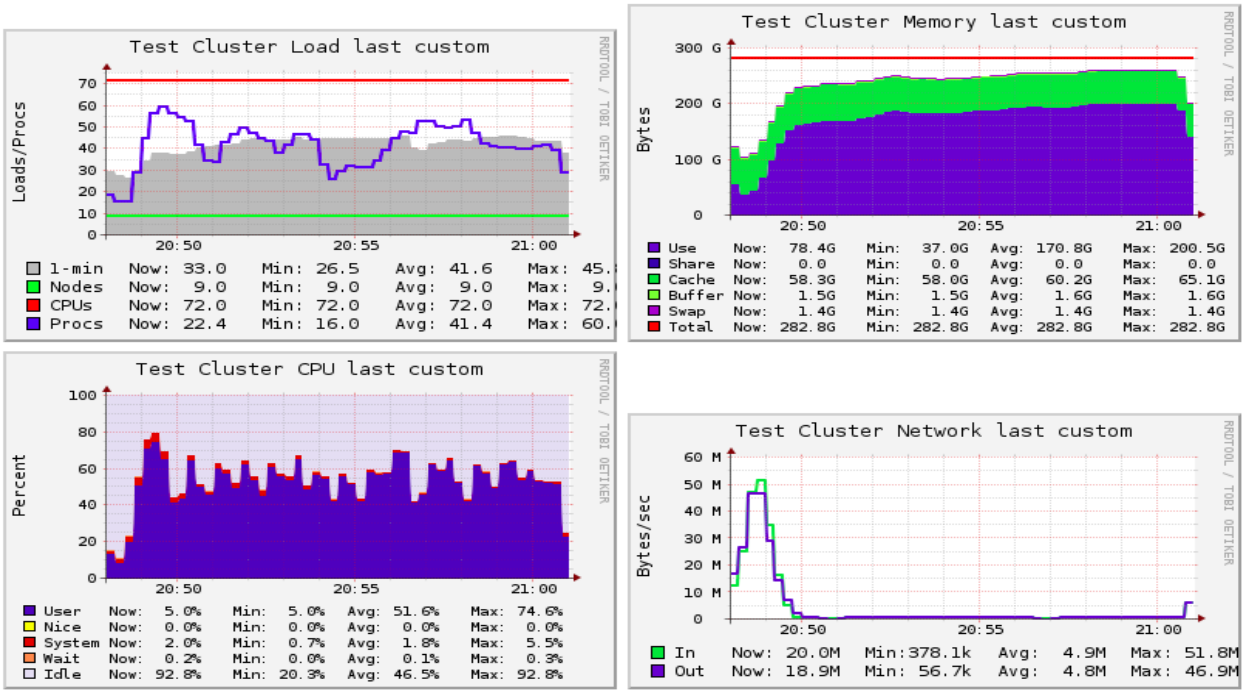

Figura 4.14: Ganglia: db 2GB, query 157 MB, k = 32.

5 milioni di record da 64 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 21:00:57 +0200 2021
Elapsed:	47mins, 30sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	585145203 MB-seconds, 25586 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.15: Application Overview: db 2GB, query 310 MB, K = 64.

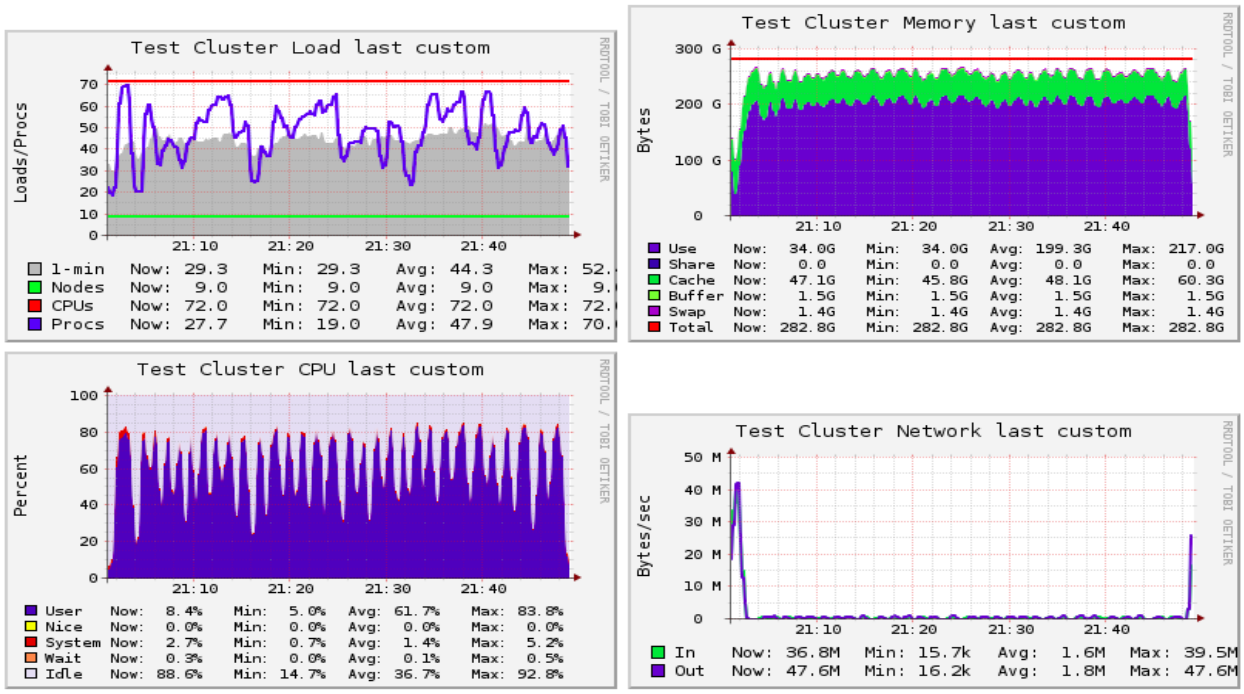

Figura 4.16: Ganglia: db 2GB, query 310 MB, K = 64.

4.3.2 Database 4GB

5 milioni di record da 16 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 21:48:42 +0200 2021
Elapsed:	6mins, 35sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	79859149 MB-seconds, 3491 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.17: Application Overview: db 4GB, query 81 MB, k = 16.

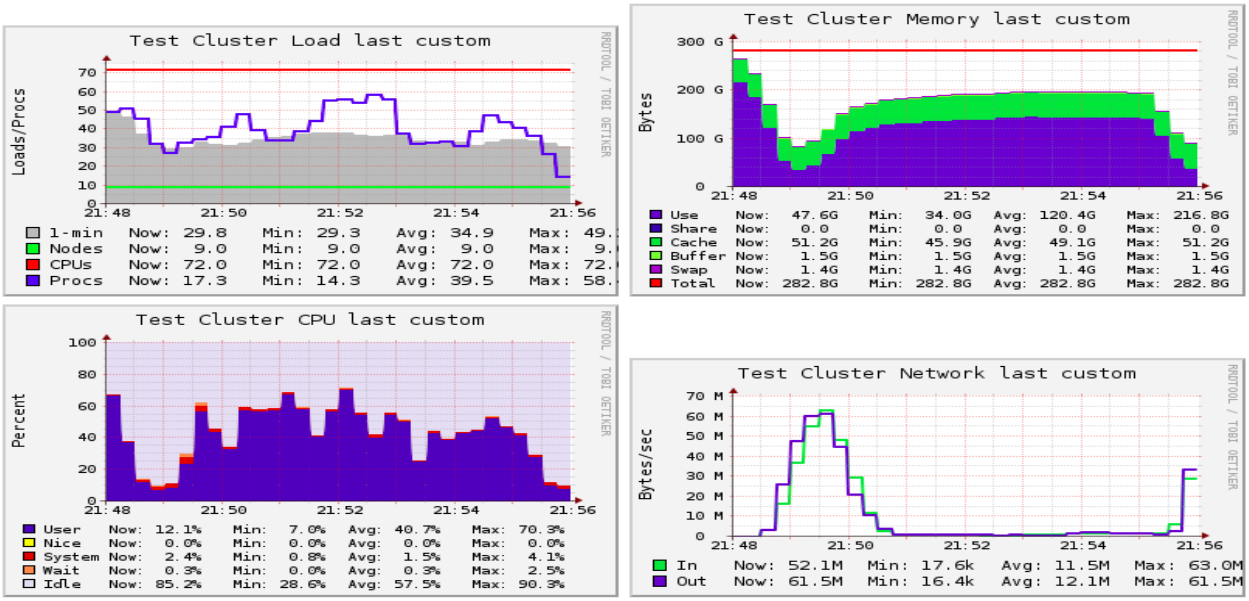

Figura 4.18: Ganglia: db 4GB, query 81 MB, k = 16.

5 milioni di record da 32 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 21:55:32 +0200 2021
Elapsed:	23mins, 16sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	285483471 MB-seconds, 12485 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.19: Application Overview: db 4GB, query 157 MB, k = 32.

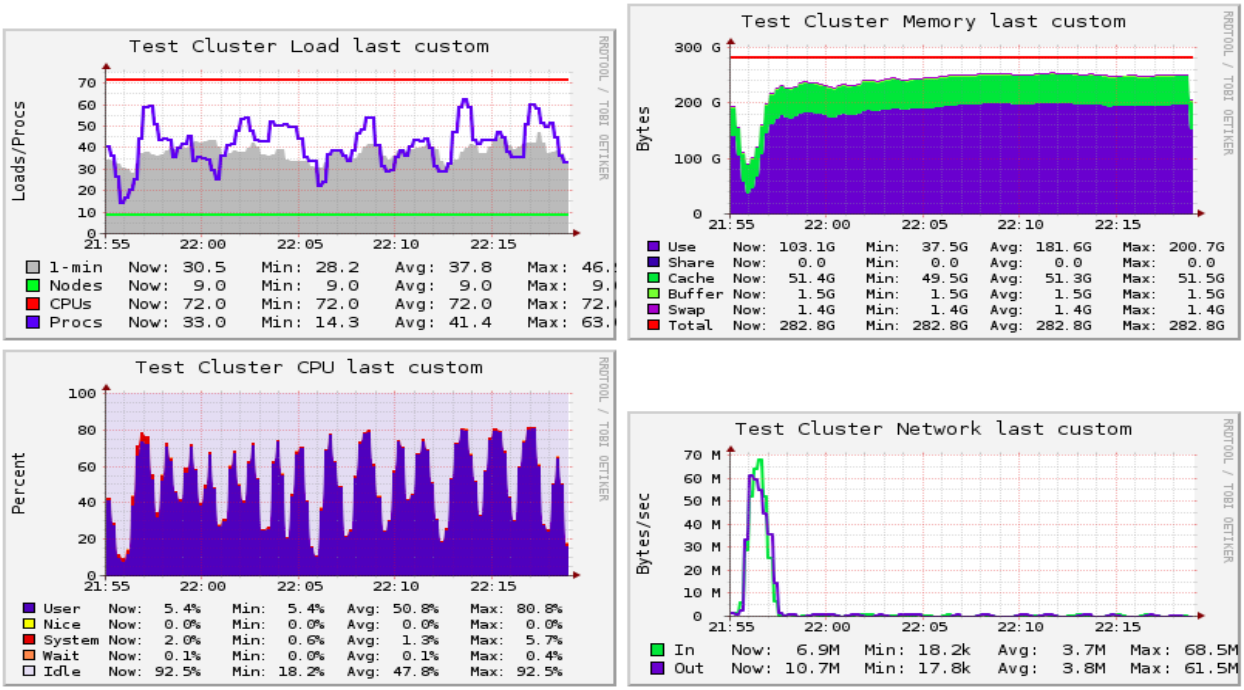

Figura 4.20: Ganglia: db 4GB, query 157 MB, k = 32.

5 milioni di record da 64 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 22:19:04 +0200 2021
Elapsed:	1hrs, 20mins, 51sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	996872962 MB-seconds, 43595 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.21: Application Overview: db 4GB, query 310 MB, K = 64.

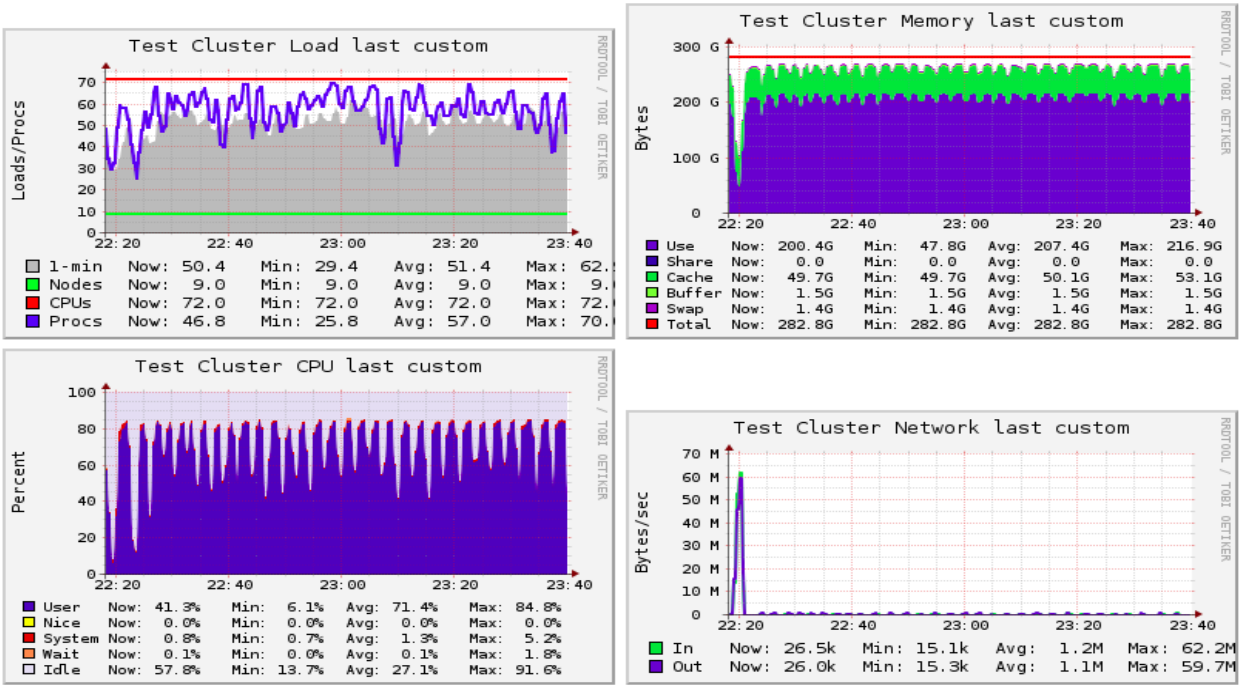

Figura 4.22: Ganglia: db 4GB, query 310 MB, K = 64.

4.3.3 Database 8GB

5 milioni di record da 16 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 23:40:12 +0200 2021
Elapsed:	10mins, 31sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	128442629 MB-seconds, 5622 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.23: Application Overview: 8gb, query 81 MB, k = 16.

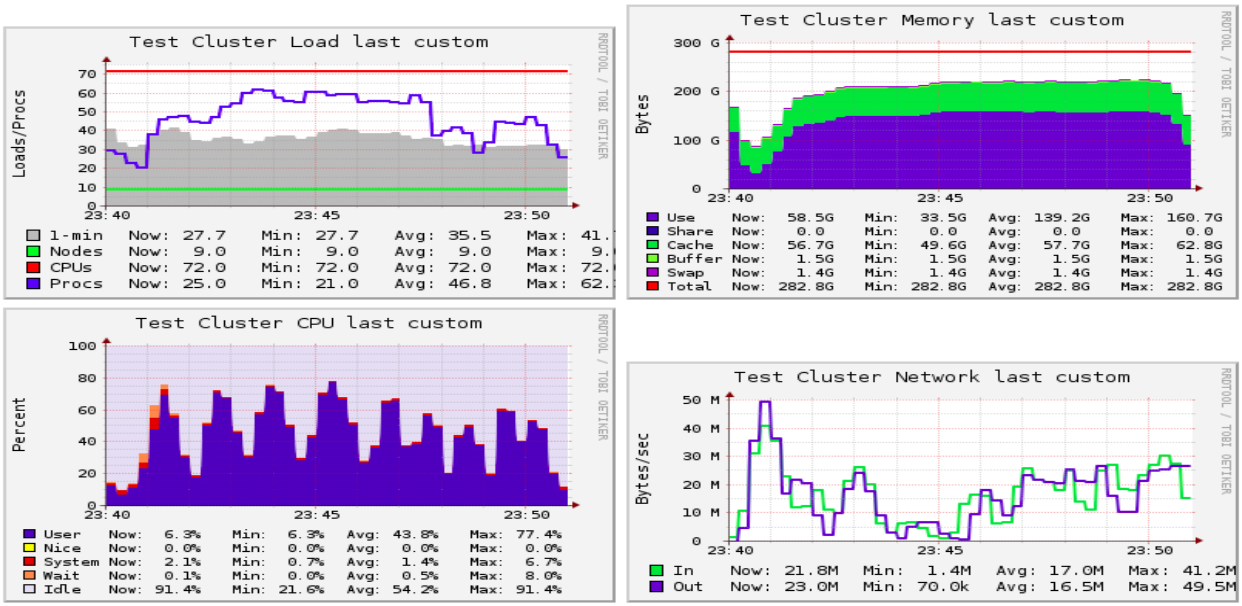

Figura 4.24: Ganglia: 8gb, query 81 MB, k = 16.

5 milioni di record da 32 caratteri

Application Overview

User:	cruoglio
Name:	ACBlastn
Application Type:	SPARK
Application Tags:	
Application Priority:	0 (Higher Integer value indicates higher priority)
YarnApplicationState:	FINISHED
Queue:	default
FinalStatus Reported by AM:	SUCCEEDED
Started:	Wed Oct 27 23:50:57 +0200 2021
Elapsed:	37mins, 37sec
Tracking URL:	History
Log Aggregation Status:	SUCCEEDED
Application Timeout (Remaining Time):	Unlimited
Diagnostics:	
Unmanaged Application:	false
Application Node Label expression:	<Not set>
AM container Node Label expression:	<DEFAULT_PARTITION>

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	463248377 MB-seconds, 20258 vcore-seconds
Aggregate Preempted Resource Allocation:	0 MB-seconds, 0 vcore-seconds

Figura 4.25: Application Overview: 8gb, query 157 MB, k = 32.

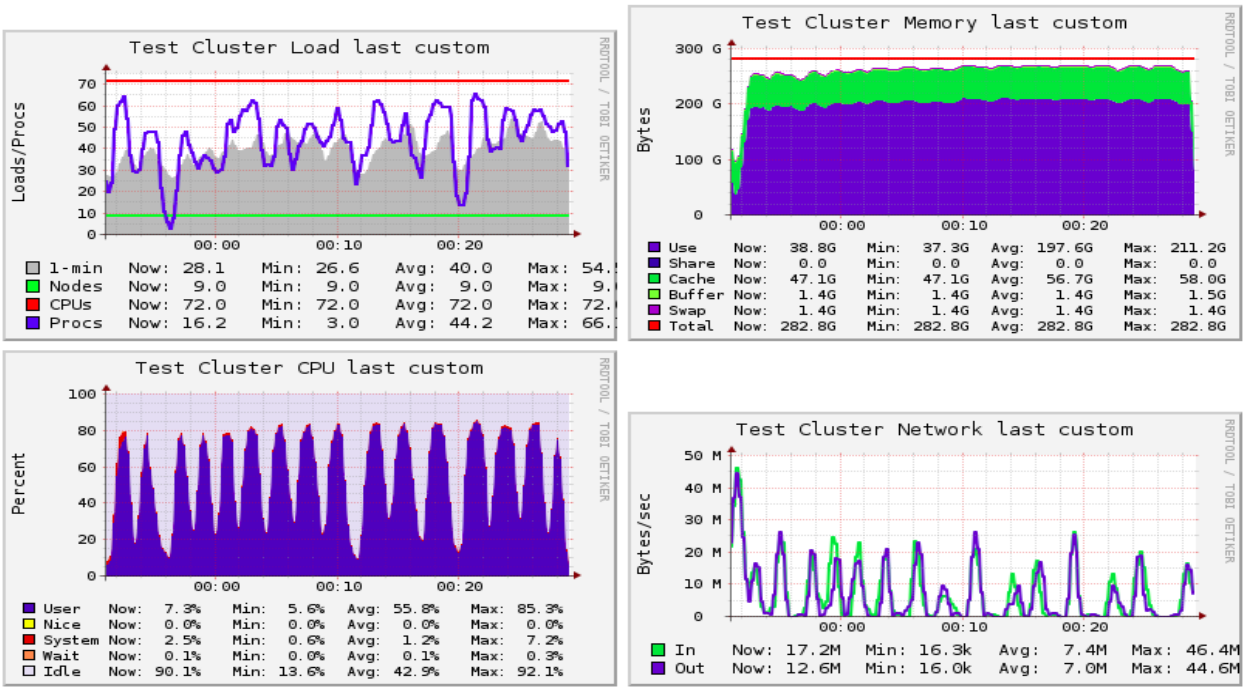

Figura 4.26: Ganglia: 8gb, query 157 MB, k = 32.

Questi due istogrammi rappresentano le esecuzioni dei 3 input per i 3 valori di grandezza di un singolo recod, cioè 16, 32 e 64. Il primo istogramma, indica il tempo di esecuzione; Il secondo, la quantità di MB allocati dall'applicazione durante tutta la durata della esecuzione. È subito evidente che al crescere della taglia dell'input, il tempo e la memoria utilizzata aumentano. Particolare attenzione va posta alla differenza tra tutte le esecuzioni degli input, con k = 64: Questo è l'effetto della crescita della struttura Aho-Corasick dove i nodi del Trie rappresentante la stringa aumenta dratiscamente e di conseguenza gestire questa quantità di informazioni risulta essere più oneroso. Per quanto riguarda l'esecuzione con input 8gb e record da 64 caratteri l'applicazione è andata in OutOfMemory.

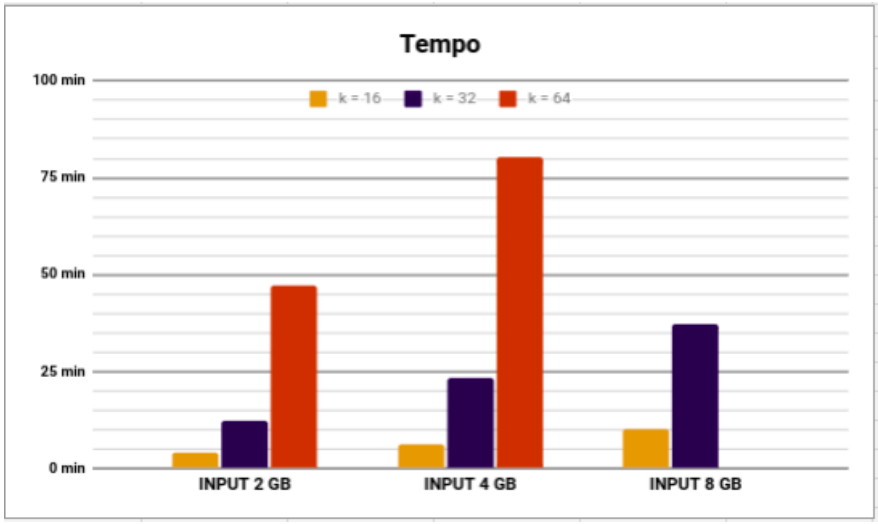

Figura 4.27: Istogramma rappresentante le esecuzioni dei 3 input per i 3 valori di k.

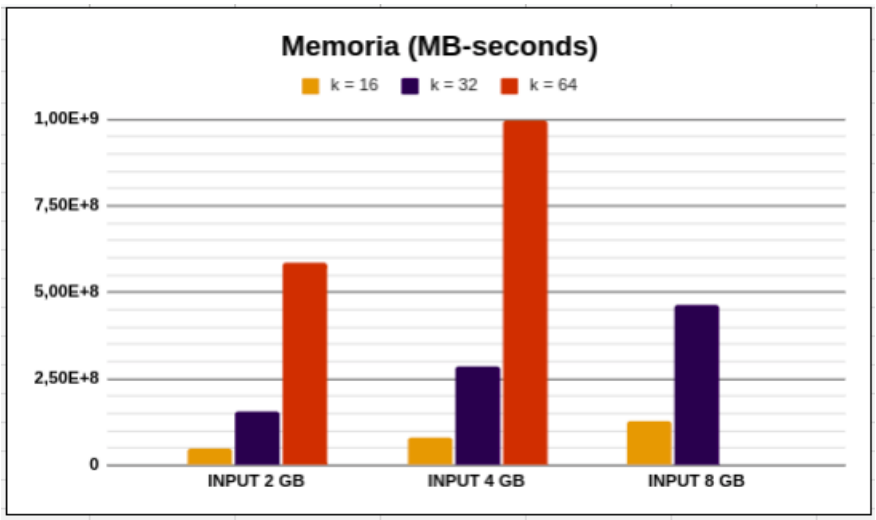

Figura 4.28: Istogramma rappresentante le esecuzioni dei 3 input per i 3 valori di k.

Capitolo 5

Conclusioni

In questo lavoro è stata affrontata la progettazione e l'implementazione di un'applicazione distribuita che ricerca da file FASTA contenenti sequenze genomiche. L'obiettivo è stato quello di computare in parallelo, nel modo più veloce possibile e soprattutto senza incorrere in errori frequenti dovuti all'ambiente di sviluppo distribuito. Per l'implementazione di questo programma è stato scelto il linguaggio Scala progettato per esprimere i modelli di programmazione generali in modo raffinato, conciso e indipendente dai tipi. Inoltre Scala è adatto a questo tipo di operazioni grazie alla sua natura funzionale che permette la modellazione dei dati attraverso le funzioni. Il file delle query, composto da stringhe di lunghezza k, è stato costruito utilizzando l'applicazione multithread KMC su un file multisequenza di tipo FASTA. KMC è un tool per l'estrazione di k-mer da una sequenza genomica il cui output è stato rielaborato allo scopo di fornire solo le query, identificate in maniera univoca. Ai fini di test la dimensione delle query è stata fissata ad un valore k costante per ogni gruppo di prove. Grazie alla prima fase di test, utilizzando un database da 2GB e una dimensione delle query k pari a 16, è stato dimostrato che l'applicazione realizzata scala in maniera lineare, riducendo i tempi di esecuzione all'aumentare del numero dei nodi. Si pensi che i tempi di esecuzione si riducono di almeno il 50% con l'introduzione del parallelismo. Con il secondo insieme di test è stato dimostrato che la dimensione delle query influisce negativamente sul carico del sistema. In particolare, si è riscontrato un aumento del tempo di esecuzione e un aumento della memoria totale utilizzata dai vari container al crescere di k. I possibili sviluppi futuri del progetto riguardano la creazione di una struttura Aho-corasick unica per ogni container. Allo stato attuale, per via della natura della black-box di aho-corasick utilizzata, viene creata una struttura per ogni partizione del database, ovvero, ogni container genera al suo interno diverse strutture in quanto elabora più partizioni. La creazione di una struttura unica potrebbe tradursi in un miglioramento generale delle prestazioni del sistema con un minor impatto sulla memoria occupata dai vari container. Un'altra possibile miglioria futura per facilitare la leggibilità dei risultati e meno influente della precedente con i tempi di esecuzione, potrebbe essere quella di realizzare una funzionalità che in maniera automatica unisca i risultati intermedi prodotti dall'applicazione in un unico risultato finale.

Bibliografia

[1] BlastReduce: High Performance Short Read Mapping with MapReduce, Michael C. Schatz.

[2] CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications, Matsunaga A., Tsugawa M., Fortes J. (2008)

[3] Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Gusfield D. (1997).

[4] Aho–Corasick algorithm: https://en.wikipedia.org/wiki/Aho-Corasick_algorithm

[5] Spark: The Definitive Guide: Bill Chambers, Matei Zaharia, 2018

[6] Documentazione di Scala: https://docs.scala-lang.org/

[7] Scala Wikipedia: https://it.wikipedia.org/wiki/Scala_(linguaggio_di_programmazione)

[8] k-mer - Wikipedia: https://en.wikipedia.org/wiki/K-mer