Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Stima del diametro di un grafo con il framework Apache Spark™

Laureando:
Boscolo Anzoletti Marco
Matricola 1057654

Relatori:
Prof. Andrea Alberto Pietracaprina
Dott. Matteo Ceccarello

Anno accademico 2014–2015

Sommario

Il tema centrale di questo lavoro di tesi è l'analisi e l'implementazione dell'algoritmo HADI per la stima del diametro e del diametro effettivo dei grafi. HADI si basa sul paradigma di programmazione MapReduce ed è pensato per lavorare con grafi di grandi dimensioni in Apache Hadoop. Per migliorare le prestazioni di HADI faremo uso di Spark, un nuovo framework per il calcolo distribuito che si presenta come molto più efficiente rispetto ad Hadoop. L'uso di Apache Spark rende il calcolo veloce e sicuro, grazie ai metodi innovativi di gestione dei dati e della computazione *in-memory*. Presentiamo anche alcune tecniche per misurare la cardinalità di un insieme di elementi in modo veloce, tra cui i contatori HyperLogLog, che saranno usati nella nostra implementazione. Tutti i programmi creati saranno valutati con diversi esperimenti che svolgeremo su un cluster di 16 macchine e avendo come input grafi che provengono da strutture dati reali. L'obiettivo finale è duplice: migliorare l'algoritmo HADI con una nuova implementazione e poi gettare le basi per la creazione di un modello di calcolo per Spark. I risultati ottenuti mostrano un incremento delle prestazioni fino a un ordine di grandezza rispetto all'algoritmo HADI originale, pensato per Hadoop.

a Francesca

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

ELENCO DEI CODICI

1 INTRODUZIONE

Elaborare grandi quantità di dati si è rivelato un bisogno di primaria importanza per le aziende, grandi e non, che nell'ultima decade hanno visto crescere esponenzialmente i dati loro disponibili, provenienti dai clienti dei loro servizi e in generale dalla rete internet. Prendendo il caso di Google, le operazioni di ricerca, catalogazione, filtraggio dei dati sono di assoluta importanza per il corretto funzionamento del motore di ricerca e dell'elaborazione dei banner pubblicitari, due settori che rappresentano il *core business* dell'azienda. Prima dell'avvento di MapReduce, gli sviluppatori dovevano creare dei programmi, indipendenti tra loro, ognuno dei quali doveva eseguire un particolare tipo di calcolo. Perciò esistevano centinaia di piccoli programmi che operavano su documenti, pagine web, file di log, query e ogni altro tipo di dato, con conseguenti difficoltà nella gestione degli errori, parallelizzazione del calcolo e gestione della memoria. Una conseguenza era che i programmatori spendevano più tempo a risolvere questi problemi che a occuparsi di cosa il programma doveva effettivamente svolgere, e il codice che si creava era molto pesante e difficile da leggere. Così, nel 2004, i progettisti di Google crearono MapReduce, un paradigma di calcolo per gestire l'elaborazione su cluster di grandi quantità di dati.

MapReduce [5], il cui nome deriva dalle primitive map e reduce dei linguaggi funzionali, fornisce al programmatore un'interfaccia semplice e potente per creare algoritmi di elaborazione dei dati e nasconde a quest'ultimo i dettagli della parallelizzazione, distribuzione dei dati e gestione dei guasti. Secondo il modello di computazione MapReduce, un programma prende in input un insieme di coppie chiave/valore, esegue alcune operazioni su di essi e produce in output un nuovo insieme di coppie chiave/-valore. MapReduce consente di gestire la computazione su dati di diverso genere in modo semplice grazie all'uso delle funzioni di tipo *map* e *reduce* e fornisce al programmatore uno strumento che astrae i dettagli relativi alla macchina su cui si esegue il programma.

Senza scendere in ulteriori dettagli, MapReduce (di cui Apache Hadoop ne è la realizzazione open source più famosa) è stato per anni lo strumento più utilizzato per eseguire questo genere di calcoli sui dati. Tuttavia, con l'introduzione di elaborazioni sempre più complesse, il paradigma MapReduce ha mostrato ai progettisti due punti deboli: la sua intrinseca "lentezza" nell'eseguire più cicli di operazioni sugli stessi dati e un utilizzo della memoria molto dispendioso. Per questi e altri motivi, si stanno cercando nuovi modi di approccio al calcolo distribuito che possano soddisfare le esigenze odierne. Google ha già innovato la sua struttura di calcolo con il suo Google Cloud Dataflow, destinato a rimpiazzare quasi del tutto MapReduce. Sul fronte open source invece, nel giugno 2013 è stato rilasciato un nuovo framework dall'Apache Software Foundation che promette di migliorare enormemente le prestazioni di Hadoop: Apache Spark [22].

Questo lavoro di tesi sarà fortemente incentrato sull'utilizzo di Spark per risolvere problemi di calcolo distribuito focalizzati sull'elaborazione di dataset di gradi dimensioni. In particolare, lavoreremo sui *grafi*. Un *grafo* è una struttura discreta formata

da due insiemi: un insieme di vertici e uno di archi, i quali collegano tra loro i vertici. Questo modo di rappresentazione le informazioni è molto comune: mappe stradali, infrastrutture telefoniche, reti elettriche possono essere descritte con i grafi; senza scordare che i *social network*, uno dei successi dell'informatica moderna, altro non sono che grafi, in cui i vertici sono le persone e gli archi i collegamenti di amicizia tra di esse. Perciò, qui studieremo che cosa sono questi *grafi* e in che modo poter ricavare delle informazioni interessanti che li descrivono. Misure di interesse sono il *raggio*, il *raggio effettivo*, il *diametro* e il *diametro effettivo*.

Vedremo in seguito la definizione matematica di questi termini, però accenniamo già che questi valori sono importanti nell'analisi dei grafi di grandi dimensioni. Per fare un esempio, nelle reti sociali il *raggio* rappresenta il valore della massima distanza tra una persona e una qualsiasi altra sempre iscritta al servizio. Un gruppo di ricerca italiano [1] ha dimostrato che il *raggio* medio di Facebook non è quello che si potrebbe in apparenza pensare, cioè un valore elevato, bensì 4,74. Questo dato può essere rilevante quando si studiano i fenomeni legati alle interazioni sociali o al comportamento di gruppo. Gli esperimenti sulle misure del grafo hanno ambiti applicativi, oltre che in ingegneria, anche in chimica, biologia molecolare, in ricerca operativa, in linguistica e in tante altre discipline.

Tra i tanti algoritmi che sono stati realizzati per elaborare i grafi, ci concentriamo su HADI [12], un algoritmo basato sul paradigma MapReduce e che si occupa di calcolare le misure di *raggio effettivo* e *diametro effettivo* dei grafi. Studieremo a fondo le componenti di HADI, la sua idea generale per il calcolo del *diametro effettivo* e come intende parallelizzare l'elaborazione tramite Hadoop. La stima del *diametro* si basa su una particolare tecnica di "conteggio", approssimata, che permette di ridurre lo spazio richiesto per memorizzare le informazioni necessarie al calcolo: si tratta dei contatori di Flajolet-Martin [8]. Questi contatori permettono di stimare in modo approssimato il numero di elementi di un insieme e saranno uno degli elementi più importanti dell'intero algoritmo.

Ci occupiamo, poi, di descrivere lo stato dell'arte di questo tipo di elaborazione nelle architetture *tightly coupled* (multiprocessore), basandoci sul lavoro di Boldi e Vigna per l'algoritmo HyperANF [3]. Qui otterremo alcune idee interessanti che saranno utilizzate nella costruzione delle nostre implementazioni.

Tornando ad HADI, spiegheremo perché questo algoritmo possiede alcune lacune quando viene messo in pratica. Innanzitutto i contatori sopracitati, benché accurati, non sono molto efficienti (occupano molto spazio) e allo stato attuale esistono tecniche migliori per fare la stessa stima con prestazioni molto più elevate. In secondo luogo, tutti i difetti di Hadoop emergono anche in HADI: lentezza nella computazione interattiva e accessi frequenti al disco appesantiscono tutta l'elaborazione.

Perciò, una volta capito l'algoritmo HADI, il nostro obiettivo è quello di aggiornare e migliorare questa procedura, servendosi del nuovo framework Spark e degli strumenti che mette a disposizione. Quello che faremo è portare HADI su Spark, testando:

- Diverse possibilità di implementazione a livello di API fornite da Spark;
- Modificando il tipo di contatori, partendo da quelli di Flajolet-Martin usati in HADI originale, fino ai moderni HyperLogLog utilizzati in HyperAnf;
- Vari tipi di grafo provenienti da dati reali e diversi per peso, diametro e complessità di elaborazione.

Un obiettivo trasversale a questo è il seguente. Da tutto questo studio di Spark, vogliamo ricavare delle informazioni interessanti sul suo comportamento, al fine di creare una bozza di un *modello di computazione* per esso. Si tratta di un argomento molto vasto e che qui introduciamo con qualche utile osservazione, data dagli esperimenti che abbiamo svolto sui grafi.

Dopo tutto questo lavoro, abbiamo ottenuto alcuni risultati significativi. Innanzitutto abbiamo abbassato la complessità computazionale dell'algoritmo HADI di un fattore logaritmico. Questo grazie all'uso di Spark. Inoltre, da un punto di vista pratico, gli esperimenti hanno dimostrato che la nuova implementazione riduce il tempo di calcolo di quasi un ordine di grandezza rispetto ad HADI, e affinando ancor di più la stima del diametro. Questo soprattutto per mezzo dei nuovi contatori HyperLogLog.

Procederemo descrivendo un argomento alla volta così da avere tutti gli strumenti per "attaccare" il problema e cercare una soluzione. Perciò la tesi è così suddivisa:

Nel SECONDO CAPITOLO descriviamo Spark ad alto livello; quali caratteristiche possiede, quali sono i punti di forza e di debolezza e in che modo possiamo servircene per i nostri calcoli.

Nel TERZO CAPITOLO presentiamo la struttura discreta *grafo*, con le sue definizioni e caratteristiche principali. Dopodiché mostreremo che cos'è HADI, qual è l'idea generale che usa per calcolare il diametro effettivo e come svolge queste operazioni in parallelo. Infine introduciamo *HyperLogLog*, una tecnica per stimare il numero di elementi di un insieme in modo efficiente e che ci servirà per migliorare la computazione. In questo capitolo faremo anche l'analisi teorica delle prestazioni degli algoritmi e dell'accuratezza statistica delle nostre stime.

Nel QUARTO CAPITOLO prendiamo Spark e lo usiamo per creare una nuova implementazione dell'algoritmo HADI. Descriviamo tre implementazioni, ognuna con un'idea di base differente, ne presentiamo il codice, e poi facciamo l'analisi teorica di quella più rilevante.

Nel QUINTO CAPITOLO presentiamo gli esperimenti che sono stati eseguiti con le implementazioni di HADI sui grafi, in modo da provare la correttezza e l'efficienza. Useremo grafi con diverse caratteristiche e che derivano da strutture reali; quindi ci serviremo di reti *mesh-2D* per analizzare il comportamento di Spark e capire quali sono le misure che più influiscono sulla sua computazione.

Nel SESTO CAPITOLO esponiamo le conclusioni del lavoro, provando ad abbozzare un modello di calcolo per Spark e valutando la bontà della nostra implementazione di HADI. Infine faremo un cenno agli sviluppi futuri che potrà avere questo studio.

Buona lettura!

2 COS'È SPARK

Apache Spark™ [22] è un nuovo framework per la computazione di grandi moli di dati su cluster. Viene scherzosamente descritto come "quello più veloce di Hadoop" perché nel confronto con il suo predecessore ha prestazioni 10-100 volte maggiori. Spark è implementato in Scala [15], un linguaggio di programmazione ad alto livello che integra caratteristiche sia dei linguaggi funzionali sia della programmazione orientata agli oggetti. La compilazione di un programma in Scala produce bytecode per la JVM. Spark fornisce le API per i linguaggi Scala, Java e Python, e offre dei preziosi tool di sviluppo come Spark SQL per creare delle query con Spark, e MLlib, una libreria ricca di algoritmi e strumenti classici per il machine learning.

Il vantaggio principale dell'utilizzo di Spark è la sua estrema velocità nell'eseguire programmi di elaborazione dati. Il motivo di queste prestazioni risiedono in una miglior gestione della memoria; aspetto che lo diversifica da Hadoop. Prendendo come esempio una generica elaborazione fatta con il paradigma MapReduce, in Hadoop questa produce il seguente schema di lavoro, che può essere iterato più volte (semplificando):

1. Load dei dati da disco locale verso i nodi *worker* del cluster;
2. Esecuzione della funzione assegnata;
3. Store dei dati su disco locale.

Le ripetute fasi di load/store rendono il sistema complessivamente lento. Spark, invece, cerca di mantenere in memoria i dati, esegue le operazioni di trasformazione e solo alla fine memorizzare i dati sul disco. Vedremo in seguito come si realizza questa procedura.

Ad alto livello, un'applicazione Spark è formata da un *driver program*, che contiene la funzione *main* scritta dall'utente, e di una serie di *parallel operation* definite nel programma che verranno eseguite sui vari nodi *worker* che compongono il cluster. Fin qui nulla di nuovo, è il modello del calcolo distribuito master/slave. La vera innovazione è stata introdotta nel modo di definire i dati da elaborare. Infatti Spark mette a disposizione un'astrazione molto potente, il *resilient distribuited dataset* (RDD), che rappresenta una collezione di dati "immutabili "a cui il programmatore si può riferire direttamente tramite l'oggetto associato. L'RDD realizza la *fault tolerance* grazie all'informazione di *lineage*: quando una partizione di un RDD si perde a causa di un guasto o di un altro errore, l'RDD ha tutte le informazioni riguardo la storia di quella partizione, in termini di operazioni effettuate su di esso — come una linea di discendenza —, che gli consentono di ricostruirla.

2.1 RESILIENT DISTRIBUITED DATASET

Spark si basa sul concetto di *resilient distribuited dataset* (RDD), che è una collezione di elementi che possono essere elaborati in parallelo e che possiede le proprietà di *fault*

tolerance. Dal punto di vista del programmatore, un RDD è semplicemente oggetto su cui invocare dei metodi e applicare delle funzioni, e come conseguenza si ottiene un codice pulito. Per chiarezza, quando parliamo di RDD dobbiamo distinguere "l'oggetto" RDD dalla sua concreta persistenza in memoria. Per esempio, con il codice **val** lines = SparkContext.textFile("dati.txt"), creo un oggetto RDD di nome lines che rappresenta il mio insieme di dati contenuti nel file di testo dati.txt. Tuttavia, finché questo RDD non sarà utilizzato in un'operazione di calcolo, non ci sarà nessuna realizzazione concreta in memoria di un file RDD, ma sarà memorizzato un puntatore con l'informazione: "lines è l'RDD creato a partire dal file di testo dati.txt". Perciò ogni volta che si parla di RDD si intende "la collezione di dati" facendo attenzione al fatto che questa non è subito creata e memorizzata nel cluster.

Ci sono diversi modi per trasformare dei dati in ingresso a Spark in un RDD:

- Da un file in un *shared file system*, come HDFS (Hadoop), Cassandra, Amazon S3, ecc.
- Partire da una collezione di Scala (come un array, un vettore o una linked list) chiamando la funzione di Spark adeguata direttamente nel *driver program*.
- Applicando delle trasformazioni a RDD esistenti. Un RDD con elementi di tipo A può essere trasformato in un altro RDD con elementi di tipo B con un'operazione di *map*, che applica agli elementi una funzione $A \Rightarrow B$. Altri metodi disponibili sono *flatMap*, *filter* e altri ancora messi a disposizione da Spark.
- Cambiando la persistenza di un RDD esistente. Normalmente, gli RDD sono volatili, cioè vengono creati su richiesta quando sono utilizzati nelle operazioni in parallelo (per esempio far passare un blocco di un file attraverso una funzione di *map*) e poi sono eliminati dalla memoria dopo l'utilizzo. L'utente può alterare la persistenza in due modi: 1) funzione *cache*: l'oggetto RDD appena elaborato rimane in memoria, in previsione di un successivo utilizzo 2) funzione *save*: esporta il dataset in un file di tipo *distribuited filesystem*. Il codice sottostante ci da un esempio:

```
val lines = SparkContext.textFile("dati.txt")
val lineLengths = lines.map(e => e.length)
val cachedLineLengths = lineLengths.cache()
```

  L'oggetto cachedLineLengths è creato applicando il metodo cache() a lineLengths. In fase di calcolo, verranno creati prima l'RDD lineLengths e poi un nuovo RDD identico a questo, con la differenza che lineLengths è volatile e sarà eliminato dalla memoria, mentre cachedLineLengths rimarrà a disposizione della macchina fino al termine del programma.

Internamente, ogni oggetto RDD ha diverse caratteristiche per descrivere le informazioni che rappresenta. La più evidente è la suddivisione in partizioni (Figura 1). Un RDD è diviso in partizioni così da essere distribuito e poi elaborato in parallelo, con un conseguente aumento della velocità di esecuzione. Ogni *partizione* è un sottoinsieme dei dati contenuti nell'RDD, può variare di dimensione e può essere specificato dal programmatore il criterio con cui operare la divisione. Perciò nell'oggetto RDD vi sono contenute queste cinque informazioni principali:

- Una lista delle partizioni;
- Una funzione per il calcolo delle partizioni;
- Un elenco delle dipendenze da altri RDD (come visto in precedenza, un RDD può essere creato partendo da un altro RDD. Coerentemente all'idea di *lineage* vengono salvati dei puntatori agli RDD padre);
- (opzionale) un oggetto Partitioner (su RDD di coppie chiave/valore, per creare delle partizioni in funzione delle chiavi);
- (opzionale) un elenco di posizioni preferite di un *filesystem* da cui partire per calcolare le partizione (ad esempio, posizioni di blocco in un file HDFS).

Tutta la schedulazione ed esecuzione in Spark si basa su questi (e altri) metodi, che consentono l'esecuzione di task su di essi con prestazioni ottimali. Per questo motivo, i progettisti possono implementare RDD personalizzati in cui sovrascrivere le funzioni di base, per adeguare il comportamento degli RDD agli scopi specifici di ogni applicazione.

2.1.1 Operazioni su RDD

Gli oggetti RDD supportano due tipi di operazioni: le *trasformazioni*, che creano un nuovo dataset RDD partendo da quello esistente, e le *azioni*, che producono un valore da restituire al *driver program* dopo aver fatto i calcoli sul dataset. Per esempio, *map* è una *trasformazione* che fa passare ogni elemento di un dataset attraverso una funzione e come risultato restituisce un nuovo RDD. Invece *reduce* è una *azione* che aggrega tutti gli elementi di un RDD (sempre applicando una funzione scritta dall'utente) e ritorna il risultato al *driver program* (per il caso di coppie chiave/valore esiste una *trasformazione*, reduceByKey, che restituisce il risultato della *reduce* applicata, per ogni chiave, ai valori con quella chiave, e poi ritorna un RDD).

In Spark tutte le *trasformazioni* sono "lazy", nel senso che i risultati non vengono calcolati subito, ma si crea un nuovo oggetto che al suo interno contiene l'informazione dell'operazione che si desidera effettuare sul dataset. Questo è il motivo per cui, di base, la creazione di un oggetto RDD non coincide con la sua realizzazione in memoria. Nel momento in cui si applica un'*azione* a un RDD, e quindi si chiede un risultato da restituire al *driver*, allora in quel momento le *trasformazioni* sono eseguite sugli RDD: vengono generati in memoria, usati per le elaborazioni, e poi eliminati se non ce n'è più bisogno.

Questo approccio è uno dei fattori che rende Spark efficiente. Per esempio, supponendo di eseguire una serie di task MapReduce a un dataset, se quello che interessa è solo il risultato prodotta dall'ultima istruzione di reduce (come di solito accade), allora il programma restituirà solamente quel risultato, e non tutti quei dataset creati e modificati durante l'esecuzione del programma.

Come impostazione predefinita, gli RDD sono realizzati solamente quando, nel codice del programma, si è in presenza di un'*azione*. Questo è il trigger che avvia il processo di creazione dei dataset RDD. Per questo motivo, viene data la possibilità di cambiare la persistenza dell'RDD con i metodi *persist* o *cache* per mantenere gli

RDD in memoria nel cluster. In questo modo l'accesso ai dati avverrà più velocemente. Vedremo poi le possibilità che fornisce Spark sulla persistenza dei dati.

Per cominciare

Un semplice esempio di elaborazione su RDD è il seguente. Consideriamo il codice sottostante:

```
val lines = SparkContext.textFile("dati.txt")
val lineLengths = lines.map(e => e.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Nella prima riga si definisce una nuova variabile lines partendo dal file di testo dati.txt per ottenere un RDD di stringhe. Per fare ciò si utilizza il metodo textFile della classe SparkContext. Questa classe è quella principale di Spark e rappresenta la connessione con il cluster: quando in un programma si crea un oggetto SparkContext, allora si attivano tutte le funzioni relative al calcolo su cluster, come la connessione con gli *slave*, la creazione dell'interfaccia di monitoraggio e così via. Sempre partendo da essa è possibile richiamare i metodi per creare oggetti RDD, accumulatori, variabili e ogni altro strumento di Spark. Tornando all'RDD, c'è da sottolineare che esso, all'invocazione di textLine, non è caricato in memoria e men che meno usato in qualche calcolo. Questo per la questione delle *azioni* spiegata prima. Molto semplicemente, quando viene creato lines, avviene che al suo interno è inserito un puntatore al file di testo.

Nella seconda riga associamo a lineLengths il risultato della funzione map applicata all'RDD. Questa funzione prende ogni singola stringa di lines e crea un altro RDD dove ne memorizza la lunghezza. Anche in questo caso nessun calcolo è eseguito (paradigma "lazy").

Nella terza riga abbiamo una reduce, che è un'*azione*. Quest'operazione prende due elementi del dataset e ne restituisce uno (in questo caso la somma dei due) e viene iterata su tutti gli elementi e ripetuta sui risultati parziali finché non si ottiene il risultato finale, cioè la somma di tutte le lunghezze delle stringhe. A questo punto del codice, Spark procede a ritroso per trovare qual è la partenza di tutto il processo, servendosi dell'informazione di *lineage* descritta in Figura 1. Solo ora gli RDD vengono realizzati in memoria, e, per ognuno di questi, ogni partizione è caricata in memoria tra le diverse macchine del cluster. Ogni macchina calcola le funzioni di map e reduce sulla partizione (o sulle partizioni se sono più di una) che gli è stata assegnata, ritornando al *driver* il risultato.

Lavorare con coppie chiave/valore

In generale, con Spark è possibile eseguire calcoli su qualsiasi tipo di RDD. Vista la loro importanza nell'elaborazione di grandi quantità di dati, esistono alcune operazioni "speciali" pensate per gli RDD formati da coppie chiave/valore. Le più comuni sono quelle relative allo "shuffle" distribuito, cioè i raggruppamenti e le aggregazioni degli elementi per chiave.

In Scala queste operazioni sono automaticamente disponibili sugli RDD che contengono oggetti di tipo Tuple2 (tipo di dato presente nel linguaggio; si crea semplicemente scrivendo (a,b)).

Per esempio, il codice sottostante usa reduceByKey su coppie chiave/valore per contare le occorrenze di ogni riga all'interno di un testo:

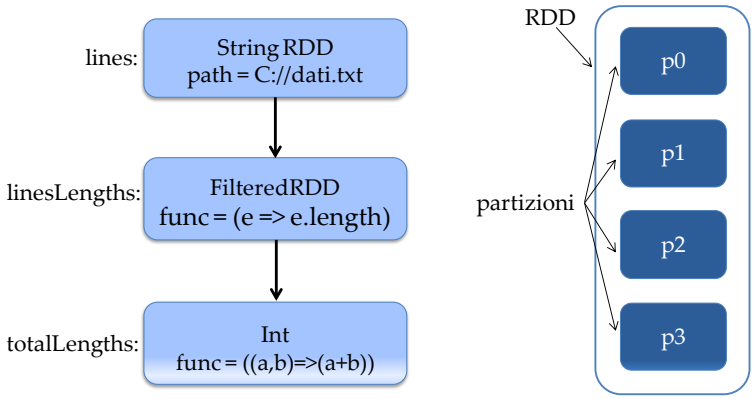


Figura 1: A sinistra, catena della *lineage* dei dataset RDD descritti nell'esempio. A fianco lo schema di un RDD con quattro partizioni.

```
val lines = SparkContext.textFile("dati.txt")
val pairs = lines.map(d => (d, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Come prima, creiamo l'RDD lines che è un dataset di stringhe. Poi creiamo pairs, un RDD di elementi di tipo Tuple2. Ogni elemento è formato da una stringa presa da lines e dalla sua lunghezza (per esempio ("questaStringa") => ("questaStringa", 13)). Infine con reduceByKey otteniamo un RDD che contiene, per ogni chiave presente in pairs, la somma dei valori associati a quella chiave.

Notiamo, in questo codice, che non sarà fatta alcuna reale creazione di RDD finché non sarà chiamata un'operazione del tipo *azione*, come count(), collect() o first().

Operazioni di base

Spark mette a disposizione del programmatore diverse operazioni che possono essere eseguite su RDD. Nella documentazione ufficiale [18] si trovano spiegate nel dettaglio, comprese quelle operazioni ancora in fase di sviluppo, messe a disposizione dagli utenti che contribuiscono al progetto. Di seguito, accenniamo solo alle operazioni più comuni, divise nei due tipi:

Tabella 1: Alcune *azioni* più comuni di Spark.

Azione	Significato
reduce(*func*)	Aggrega gli elementi di un RDD con la *func* (che prende due argomenti in ingresso e ne restituisce uno). La funzione deve godere delle proprietà associativa e commutativa per poter essere calcolata in parallelo.
count()	Restituisce il numero di elementi nell'RDD.
collect()	Restituisce gli elementi dell'RDD al *driver program* in un array.
countByKey()	Solo su RDD del tipo (K, V). Restituisce una hashmap di coppie (K, Int) con il conteggio del numero di elementi per ogni chiave.

Tabella 2: Alcune *trasformazioni* più comuni di Spark.

Trasformazione	Significato
map(*func*)	Restituisce un nuovo RDD applicando *func* a tutti gli elementi dell'RDD sorgente. La funzione associata a map deve essere del tipo func: T => (f(T) = U), dove T rappresenta un elemento dell'RDD di tipo T, mentre U è un elemento del (nuovo) tipo RDD creato. Per fare esempio banale, se parto da un RDD "dateCalendario" in cui ogni elemento è descritto come (giorno, mese, anno), dateCalendario.map(a => (a._2, a._1, a._3)) restituisce un RDD con le stesse date, ma con le posizioni di mese e giorno scambiate.
filter(*func*)	Restituisce un nuovo RDD selezionando gli elementi dell'RDD sorgente per cui la *func* ritorna *true*.
flatMap(*func*)	Come map, però ogni elemento in input può produrre un numero di output ≥ 0.
mapPartitions(*func*)	La funzione in ingresso deve essere del tipo func: Iterator(T) => Iterator(U). A differenza di map, che prende un solo elemento in input e ne restituisce un'altro in output, *func* accetta un iteratore su tutti gli elementi di una partizione. Questa è distribuita a ogni partizione dell'RDD e applicata ad argomenti diversi.
groupByKey([*numTask*])	Su dataset (K, V), restituisce un dataset di coppie (K, Iterable<V>). *nota*: per operare un'aggregazione su chiavi (come una somma), funzioni come reduceByKey sono più performanti.
reduceByKey(*func*, [*numTask*])	Questa trasformazione si applica a dataset di coppie chiave/valore (K, V), e restituisce un nuovo RDD(K, V) operando l'aggregazione per chiave usando la funzione *func*, che dev'essere del tipo (V, V) => V (nell'esempio del paragrafo precedente, per sommare tutte le occorrenze di una stessa parola all'interno di un testo avevamo func: ((a, b) => a + b)). In opzione, si può indicare il numero di partizioni RDD desiderate con *numTask* se si desidera che l'RDD abbia un diverso partizionamento rispetto all'RDD padre.

Continua nella prossima pagina

Continua dalla pagina precedente

Trasformazione	Significato
sortByKey([*ascending*], [*numTask*])	Si applica a RDD di tipo chiave/valore (K, V) dove alle chiavi è associato una qualche forma di ordinamento. Restituisce un nuovo RDD con le stesse coppie chiave/valore, però ordinate in base alla chiave. L'ordinamento può essere ascendente o discendente in base al valore **true** o **false** associato ad *ascending*.

2.2 LIVELLI DI PERSISTENZA

Una delle innovazioni più apprezzate in Spark è la possibilità di modificare la persistenza degli RDD. Come visto in precedenza, di base un RDD è calcolato solo all'invocazione di un'*azione* e poi sparisce dalla memoria. Con il metodo persist() ogni nodo mantiene la sua partizione di RDD in memoria, così da poter eseguire altri calcoli su di essa da parte di successive istruzioni. Ciò incrementa le prestazioni (sperimentalmente anche di un fattore 10) ed è fondamentale per realizzare algoritmi iterativi ed eseguire query di tipo interattivo con una certa velocità. Alcune modalità di persistenza che Spark mette a disposizione:

Tabella 3: Livelli di persistenza degli RDD.

Livelli di persistenza	Significato
MEMORY_ONLY	Mantiene l'RDD in memoria, cioè le varie partizioni vengono memorizzate nei nodi del cluster. Le partizioni sono salvate nella forma di oggetti Java, a cui non è applicata alcuna forma di serializzazione. Se la dimensione dell'RDD è troppo grande per la memoria del cluster, alcune partizioni non verranno memorizzate, ma si provvederà a calcolarle "al volo" quando ci sarà la necessità di utilizzarle (impostazione predefinita).
MEMORY_AND_DISK	Come MEMORY_ONLY, però, nel caso di RDD con dimensioni superiori alla memoria disponibile, le partizioni in eccesso sono memorizzate su disco.
MEMORY_ONLY_SER	La differenza con MEMORY_ONLY è che memorizzo gli oggetti Java in forma serializzata. Come impostazione predefinita Spark si serve della classe ObjectOutputStream di Java (è anche possibile usare implementazioni più efficienti). In generale, questa forma di persistenza è più efficiente per lo spazio occupato ma richiede più calcoli da parte della CPU.
DISK_ONLY	Salva l'RDD su disco.

2.3 VARIABILI CONDIVISE

Normalmente, quando Spark esegue un'operazione (come map o reduce) sul cluster, il codice della funzione è copiato in ogni nodo e con esso anche le variabili associate. Perciò in ogni macchina c'è una copia delle variabili del programma principale, e nessun aggiornamento o modifica di queste variabili è poi restituito al *driver program*. Questo perché un generico supporto alla lettura-scrittura di variabili condivise tra i task sarebbe estremamente inefficiente a causa del costo dovuto allo scambio di messaggi nella rete (il collo di bottiglia tipico di queste elaborazioni). Per questo, e per la necessità di disporre di variabili condivise, Spark fornisce due soli tipi di queste variabili: accumulatori e variabili broadcast.

2.3.1 Variabili broadcast

Le variabili broadcast consentono al programmatore di fornire ogni nodo del cluster di una stessa variabile (in sola lettura) in modo efficiente. Per esempio, se ogni nodo ha bisogno di un array di grandi dimensioni, invece di inviarlo attraverso il task, questo viene memorizzato nella cache di ogni nodo per mezzo della variabile broadcast. Spark si serve di algoritmi efficienti per la trasmissione di queste variabili al fine di ridurre i costi di comunicazione.

2.3.2 Accumulatori

Gli accumulatori sono variabili pensate appositamente per riceve frequenti aggiornamenti. Su di esse si può solo "aggiungere" attraverso operazioni di tipo associativo, perciò supportano efficacemente operazioni di aggiornamento svolte in parallelo. Contatori e sommatori sono fatti con questo tipo di variabile.

2.4 PANORAMICA DEL FUNZIONAMENTO DI SPARK SU CLUSTER

Esponiamo qui un'idea, sempre ad alto livello, di come un generico programma è eseguito su una macchina parallela, specificando i componenti della macchina e il loro uso.

2.4.1 Componenti

Un'applicazione Spark è eseguita come un insieme di processi indipendenti sul cluster, coordinati dall'oggetto SparkContext contenuto nel programma principale (che abbiamo sempre chiamato *driver program*). Nello specifico, lo SparkContext deve connettersi al *cluster manager* (che può essere sia quello rilasciato da Spark, sia di altre distribuzioni, come YARN, Mesos) che ha il compito di allocare le risorse. Una volta connesso, ha il compito di avviare gli *executor*, che sono i processi responsabili dello svolgimento delle operazioni. Ogni processo *executor* è in realtà una Java Virtual Machine, a cui viene inviato:

- Il codice del programma (contenuto in un file JAR)

- I task che deve eseguire. Possono essere più di uno e a ogni task corrisponde una partizione di RDD

In Figura 2 vediamo lo schema delle componenti di Spark. Alcune osservazioni:

- Ogni applicazione possiede i suoi processi *executor*, che rimangono attivi per tutto il tempo in cui il programma è in esecuzione e con il compito di realizzare i task associati dividendoli in vari thread. Questo tipo di implementazione ha il vantaggio di isolare le applicazioni tra loro, sia dal lato della schedulazione (ogni driver decide la schedulazione dei suoi task), sia dal lato dell'esecuzione, perché task provenienti da applicazioni differenti sono eseguiti in differenti JVM. Questo implica che le informazioni tra due applicazioni non possono essere condivise, a meno di salvare una parte di queste informazioni su un supporto di memorizzazione esterna.
- Spark non conosce nello specifico e non si preoccupa del funzionamento del *cluster manager*.
- Siccome il driver schedula i task sul cluster, dovrebbe essere "vicino" agli *worker*. Sarebbe preferibile che si trovassero tutti nella stessa LAN.

2.4.2 Esempio di applicazione in Spark

Dopo aver descritto le parti che compongono il framework e il suo funzionamento ad alto livello — in particolare il ruolo dell'RDD — siamo pronti per analizzare un'applicazione reale in Spark. Nell'esempio del Codice 1 presentiamo un semplice programma per calcolare la media dei voti attribuiti a una lista canzoni. In input ho un dataset fatto di stringhe così formattate: "userId <TAB> songId <TAB> rating". Quello che vogliamo ottenere è la media dei voti che una singola canzone ha ottenuto, indipendentemente da quanti utenti l'hanno votata o da quanti voti ha preso rispetto alle altre. La scala dei voti va 1 a 5.

Dalla definizione del problema si intuisce che il paradigma MapReduce può essere utilizzato per ottenere una soluzione efficiente e quindi usiamo Spark per scrivere la nostra applicazione. Anche se l'avvio della computazione avviene con l'istruzione saveAsTextFile (che è un'*azione*), guardiamo cosa avviene in Spark partendo dall'inizio del programma, immaginando che i calcoli avvengano sequenzialmente in ordine di chiamata.

Nel momento in cui viene creato lo SparkContext si definisce l'ambiente di lavoro relativo all'applicazione SongsCount: quanti e quali sono gli executor, quanta memoria associare a ogni executor, quanti core usare per ogni macchina e tutti gli eventuali parametri che si possono impostare con SparkConf. Creato il primo RDD, songsData, Spark lo divide in partizioni con una dimensione massima di 32MB (valore di default) e questi vengono distribuiti tra i nodi del cluster per essere elaborati. Il numero di partizioni non varia da un RDD all'altro, se non esplicitato come parametro nei metodi che lo consentono, e questo è un vantaggio perché permette di parallelizzare le operazioni che non hanno dipendenze da altri RDD. Infatti, le tre istruzioni successive sono dei map che applicano una funzione ad ogni elemento dell'insieme. Quest'operazione gode delle proprietà associativa e commutativa, per cui si può applicare a tutti gli elementi di una partizione senza preoccuparsi del risultato prodotto dagli altri. La

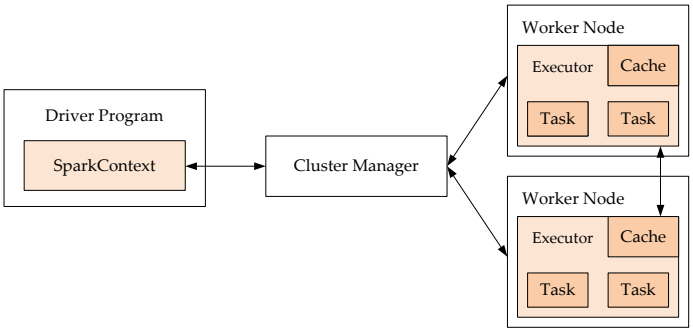


Figura 2: Schema dei componenti di Spark.

Codice 1: Esempio di applicazione MapReduce in Spark.

```
/* SongsCount.scala */

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object SongsCount {
  def main(args: Array[String]) {
    // Indirizzo a cui si trova il file
    val logFile = "C:\\spark\\test.txt"
    // Creo lo SparkConf per impostare il nome dell'applicazione e abilitare
    // il registro degli eventi
    val conf = new SparkConf().setAppName("SongsCount").set(
        "spark.eventLog.enabled", "true")
    // Creo lo SparkContext con la configurazione definita in "conf"
    val sc = new SparkContext(conf)

    /* Definisco un RDD di stringhe a partire dal file di testo. D'ora in poi
       ci sarà una sequenza di "trasformazioni" da RDD a RDD, ricordando che
       queste non saranno realizzate fino all'avvento di una "azione"
    */
    val songsData = sc.textFile(logFile)
    // Definisco un RDD del tipo Tuple3 (a,b,c) usando il TAB come separatore
    val tuple3RDD = songsData.map(a => a.split("\t"))
    // Uso map per trasformare gli elementi in valori interi
    val dataInt = tuple3RDD.map(a => (Integer.parseInt(a(0)),
        Integer.parseInt(a(1)), Integer.parseInt(a(2))))
    // Creo un RDD di coppie chiave/valore
    val realKV = dataInt.map(a => (a._2, (a._3, 1)))
    // RDD ottenuto sommando i voti dati alle singole canzoni
    val res = realKV.reduceByKey((a,b) => (a._1+b._1, a._2+b._2))
    // Uso map per associare ad ogni canzone la media dei voti: totale/#voti
    val songsRate = res.map(a=> (a._1, a._2._1.toFloat/a._2._2))

    /* Salvo i risultati su disco. Questa "azione" è il punto di avvio del
       processo di calcolo. Seguendo la lineage degli RDD Spark comincia il
       lavoro a partire dal primo RDD "songsData"
    */
    songsRate.saveAsTextFile("C:\\finalCount")
  }
}
```

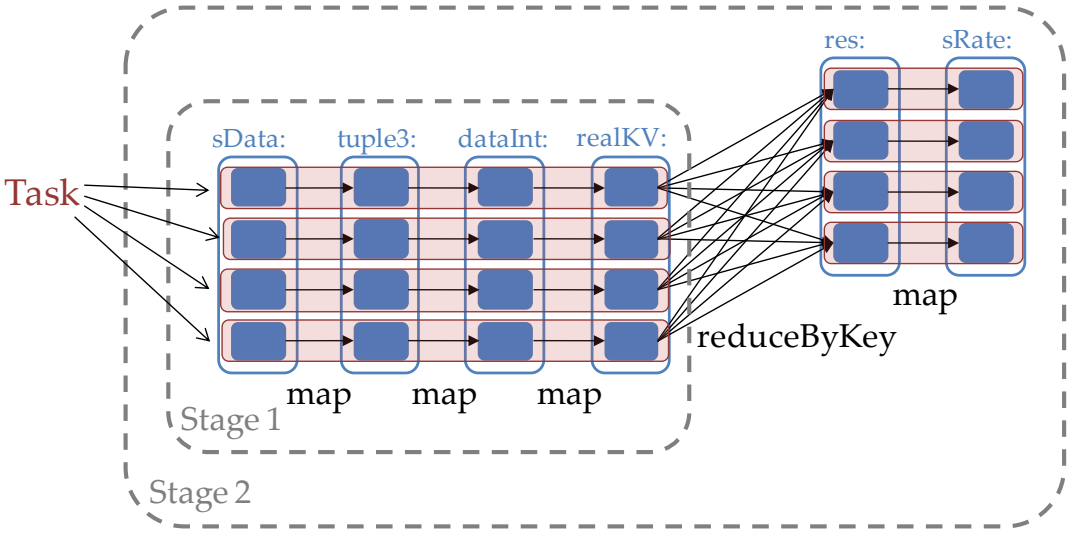


Figura 3: Schema di esecuzione dell'applicazione *SongsCount*. I box in rosso sono i task, sequenze di istruzioni eseguite da un solo executor.

Figura 3 mostra il primo dataset diviso in quattro partizioni e la sequenza delle operazioni svolte da Spark su di esse. L'istruzione reduceByKey, invece, è dipendente dai dati presenti in tutte le altre partizioni, perciò gli executor devono comunicare tra loro per scambiarsi i dati relativi alle chiavi che devono elaborare, con il conseguente aumento del tempo di esecuzione. Infine, abbiamo un altro map e il salvataggio dei risultati su disco.

2.5 MAPREDUCE

Lo abbiamo citato più volte in questa prima parte della tesi. Vista l'importanza che ha questo modello nel nostro lavoro, ne diamo qui una breve descrizione. MapReduce è un paradigma di programmazione per l'elaborazione dei dati in modo parallelo. In origine il termine *MapReduce* identificava un framework software sviluppato da Google per il calcolo distribuito di dataset molto grandi; dopodiché il nome è diventato di uso generale per indicare "quella" specifica modalità di operare sui dati. Il suo nome deriva dalle operazioni di *map* e *reduce*, tipiche dei linguaggi funzionali. Sarà possibile replicare il funzionamento di questo paradigma in Spark utilizzando i metodi messi a disposizione da esso. MapReduce è una sequenza di operazioni applicate a un dataset di grandi dimensioni. Supponendo di avere come hardware un generico cluster con un master e molti slave (qui chiamati *worker*), sintetizziamo la procedura in questo modo:

- Input: All'inizio i dati sono conservati nella memoria secondaria, generalmente dischi rigidi e divisi in partizioni. Quando il *master* avvia la procedura di calcolo, ogni *worker* si prende in carico una o più partizioni.
- Map: In questa fase il *worker* applica una funzione "map()" ai suoi dati. Il risultato di questa funzione è una lista di dati nel formato $(chiave, valore)$. Questi vengono salvati su un dei file temporanei, su disco.

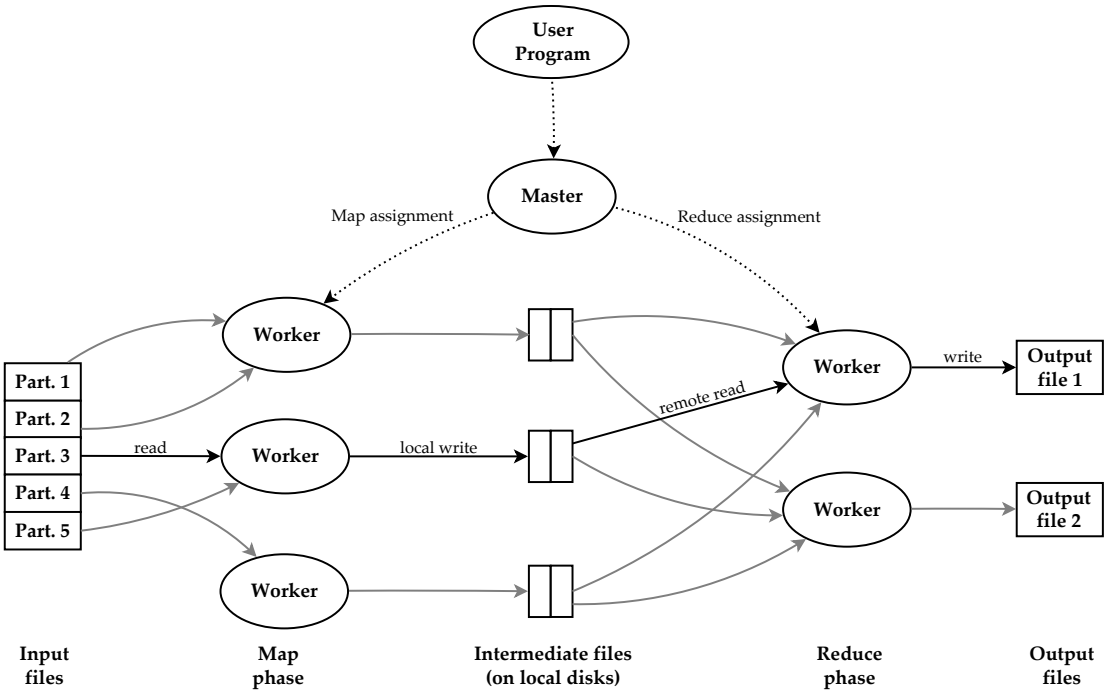


Figura 4: Schema di un'elaborazione MapReduce.

- Shuffle: Una delle operazioni più critiche per quanto concerne la complessità. Si tratta di distribuire in modo ordinato i dati creati in precedenza tra i vari *worker*, in modo che le informazioni con la stessa chiave siano computati il più possibile nella stessa macchina, nel passo successivo di Reduce. Per ottenere questo sono possibili varie strade, dal replicare tutti i dati su ogni macchina (solo per dataset molto piccoli) a servirsi di algoritmi di ordinamento come quicksort, mergesort (come Hadoop) e bucketsort (in Spark).

- Reduce: Ogni gruppo di dati, in base alla chiave, viene elaborato mediante una funzione "reduce()" che, ai fini di parallelizzare il calcolo, deve godere delle proprietà commutativa e associativa. Alla fine si ottiene un solo oggetto per chiave nel formato $(chiave, valore)$.

- Output: I risultati della Reduce sono scritti su disco e completano il processo.

In Figura 4 vediamo lo schema della procedura appena descritta. Questo paradigma di calcolo era stato pensato, inizialmente, perché fosse eseguito una sola volta su dataset di grosse dimensioni. Considerata l'utilità e la sua efficienza, negli anni questo schema si è ampliato per poter operare in sequenza molte istanze di MapReduce, in cui l'output di un ciclo diventa l'input del ciclo successivo.

Osserviamo, però, che l'approccio originale prevede un accesso al disco in modo continuato, dopo ogni operazione sui dati. Ciò porta a un notevole overhead, sia per gli accessi di lettura/scrittura, si per l'utilizzo intenso della rete di comunicazione. Allora possiamo intuire perché le idee di Spark, rispetto alla computazione *in-memory*, siano una strategia vincente quando si devono eseguire molte istanze di MapReduce. Vedremo poi un esempio chiaro quando eseguiremo degli esperimenti su grafi con questo paradigma di programmazione.

2.6 GLOSSARIO

Per chiarezza, descriviamo sinteticamente i termini più utilizzati nell'ambito della programmazione con Spark. Questa lista sarà utile da un lato per non confondere i concetti quando parliamo di task, job, worker (che nel linguaggio dell'informatica possiedono già un significato), dall'altro, per definire un modo standard di esprimere tutte le parti che compongono le nostre applicazioni.

Tabella 4: Glossario di Spark.

Termine	Significato
Applicazione	Il programma scritto dall'utente. Contenere il *driver program* necessario per l'avvio di tutta l'elaborazione. Fanno parte dell'applicazione anche i processi *executor* presenti nei nodi del cluster e che si occupano della realizzazione di questa specifica applicazione.
Driver program	Il processo che esegue la funzione main() e crea lo SparkContext.
Cluster manager	Il servizio esterno che mette a disposizione le risorse, da acquisire, del cluster (Mesos, YARN, ecc.).
Worker node	Qualsiasi nodo del cluster che può eseguire il codice dell'applicazione.
Executor	Processo lanciato da un'applicazione sul worker. Questo processo esegue i task, mantiene i dati in memoria o li registra su disco. Ogni applicazione ha i suoi *executor*.
Task	L'unità di lavoro da inviare a un *executor*.
Job	Il calcolo in parallelo di più task, generato in risposta a un'*azione* Spark.
Stage	Ogni job è diviso in piccoli insiemi di task, chiamati *stage*, che dipendono l'uno dall'altro (come i passi map e reduce in MapReduce).

2.7 CENNI DI CONFIGURAZIONE E ALTRI STRUMENTI

Spark è un framework innovativo e ricco di strumenti per il progettista. Senza scendere in dettagli, è facile intuire che al suo interno presenta molte altre impostazioni, funzioni e metodi che gli sviluppatori hanno creato per soddisfare le esigenze di chi opera nel settore del calcolo parallelo. Perciò è bene rimandare alla documentazione ufficiale [17] tutti gli approfondimenti legati al codice e alle classi. Invece, diamo qui una piccola panoramica sugli strumenti più comuni che Spark mette a disposizione per ottimizzare il lavoro.

2.7.1 La shell

Spark mette a disposizione una shell, la quale carica uno speciale interprete della classe SparkContext (riferendosi ad essa con sc) già impostato per eseguire online i comandi passati dall'utente. Questa modalità è molto comoda per fare piccole prove o eseguire alcuni processi di base.

2.7.2 Sistemi di controllo

Ci sono diversi modi per monitorare le applicazioni Spark. Oltre ai file di log, lo strumento più semplice è l'interfaccia web che ogni SparkContext indirizza sulla porta 4040 e che visualizza le informazioni più comuni di un'applicazione, tra cui:

- Una lista dei task e degli stage;
- Le dimensioni degli RDD e l'utilizzo della memoria;
- Informazioni sull'ambiente di processo;
- Informazioni sul lavoro degli executor.

2.7.3 Configurare Spark

Due argomenti relativi alla configurazione vale la pena citare: la serializzazione dei dati e la gestione della memoria.

La serializzazione dei dati ha un ruolo fondamentale nelle prestazioni di un'applicazione distribuita. Un oggetto che occupa molti byte e difficile da serializzare ha un impatto negativo sul calcolo. L'obiettivo di Spark è quello di mediare tra la convenienza (consentendo l'utilizzo di qualsiasi tipo di dato Java) e le prestazioni. Per questo mette a disposizione due librerie per la serializzazione: Java serialization e Kryo serialization. Quella di Java è l'impostazione predefinita. Kryo è molto più performante, ed è consigliato usarlo, però non supporta tutti i tipi `Serializable` e il suo funzionamento è più complicato.

Per la gestione della memoria, ci sono tre fattori da considerare: la quantità di memoria usata dagli oggetti, il costo per accedervi e l'overhead causato dall'uso intensivo del garbage collector (nel caso di frequenti turnover di oggetti). Per questi motivi è importante creare oggetti che siano più leggeri possibile, per esempio usando i tipi primitivi invece che le *collezioni* messe a disposizione da Java e Scala.

3 HADI E IL DIAMETRO DI UN GRAFO

Un grafo è una struttura discreta formata da due insiemi: un insieme di vertici e uno di archi, i quali collegano tra loro i vertici. Le informazioni sono normalmente contenute nei vertici, ma non sono infrequenti i casi dove anche gli archi hanno un valore (grafi pesati). Nonostante le apparenze, oggi ci scontriamo tutti giorni con questo tipo di oggetti. I social network (Facebook, Twitter...) si basano sull'interazione tra amici, che a loro volta sono "collegati" con altri amici e così via in modo iterativo. Queste informazioni si memorizzano in modo ottimale nella struttura dati grafo. Ma anche le reti stradali, un insieme intricato di città e strade con diverse lunghezze e importanza, oppure l'infrastruttura di comunicazione di un ISP sono tutti esempi che ci riportano alla stessa cosa. Per completezza va ricordato che internet altro non è che un grafo, dove le pagine sono i vertici e i link gli archi.

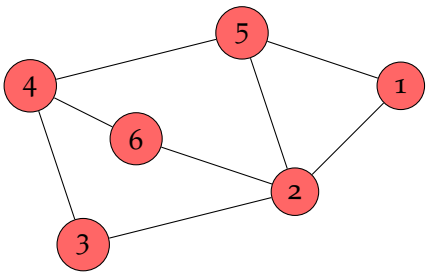
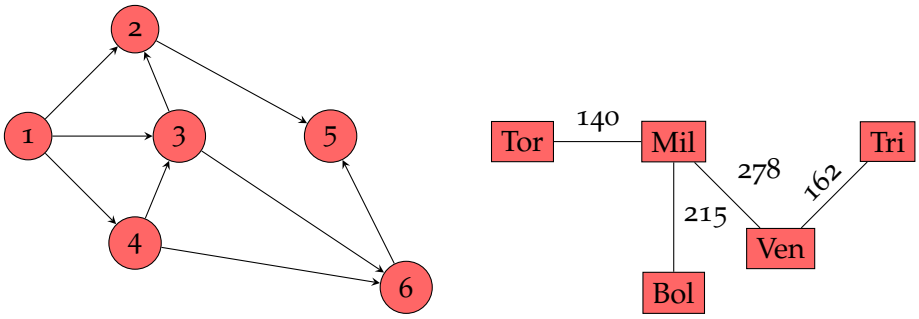


Figura 5: Un piccolo esempio di grafo semplice

Gli esempi sono moltissimi, e soprattutto in tempi recenti con l'esplosione del web, le reti hanno acquistato moltissima importanza e perciò anche il loro studio. I grafi che le descrivono spesso hanno dimensioni molto elevate, nell'ordine dei gigabyte o terabyte, con miliardi di vertici e archi. A fini pratici possiamo chiederci: che distanza hanno i vertici tra loro? Come sono distribuiti nella rete? Esistono dei vertici con una posizione dominante? La risposta non è immediata. Per dare una risposta che sia esatta abbiamo bisogno di "esplorare" il grafo in ogni sua parte e di eseguire delle operazioni come: memorizzare i cammini tra i vertici per ricordare se li abbiamo già attraversati o meno, misurare le lunghezze che intercorrono tra coppie di vertici e altre misure simili. Ora, avendo un grafo come in Figura 5 la cosa non sembra un problema, ma se pensiamo a tutti i nostri amici su Facebook, e per ognuno di essi, tutti i loro amici, allora possiamo intuire che il calcolo comincia a complicarsi. Inoltre, moltissimi grafi di interesse pratico hanno miliardi di archi e vertici, quindi la strada di esaminare a fondo tutta la complessa struttura dati richiederebbe un costo in termini computazionali (soprattutto di spazio) che sarebbe proibitivo per molti, a meno di non usufruire di supercomputer o cluster di grandi dimensioni.

Un'alternativa alla misura esatta dei valori del grafo è fare delle misure approssimate. In molte applicazioni spesso non è necessario il numero esatto, ma piuttosto l'ordine di grandezza, oppure ottenere un intervallo di valori possibili. Sotto quest'ipotesi le strade sono molteplici. Si possono eseguire calcoli solo su una parte del grafo, magari su alcuni vertici scelti casualmente, oppure limitare l'esplorazione del grafo a un suo sottoinsieme che sia il più possibile rappresentativo di tutta la struttura. Nel mio lavoro di tesi seguiremo questa strada, cioè fare delle misure sui grafi con delle tecniche di approssimazione.

In questo capitolo, dopo aver descritto in modo formale cos'è un grafo e definire



(a) *Grafo orientato.*

(b) *Grafo pesato. Distanze tra alcune città italiane.*

Figura 6: Esempi di grafi.

alcune misure su di esso, presenteremo l'algoritmo HADI [12], un metodo per ottenere una misura approssimata del diametro e del diametro effettivo per un grafo di grandi dimensioni. Infine, farò un accenno a una tecnica per la stima della cardinalità di un insieme, legata allo sviluppo di HADI, e di cui mi servirò più avanti nel mio lavoro.

3.1 DEFINIZIONI

Un *grafo* è una struttura discreta che descrive una relazione tra oggetti. Gli oggetti sono i *vertici* mentre le relazioni che intercorrono tra le coppie di vertici sono gli *archi*. Da un punto di vista matematico, diciamo che un grafo G è formato dall'insieme V dei vertici e dall'insieme A degli archi. Gli archi di un grafo possono essere *orientati* o *non orientati*. Un arco (u, v) si dice *orientato* da u a v se la coppia (u, v) è ordinata, con u che precede v. Ne consegue che, in presenza di archi orientati, si può parlare di archi *entranti* e archi *uscenti* a seconda di quale sia l'origine e la destinazione di questi. Se tutti gli archi di un grafo sono orientati, allora si parla di grafo *orientato*, mentre nel caso opposto si dice grafo *non orientato*. Se gli archi di un grafo possiedono informazione, si dice che quello è un grafo *pesato*. In Figura 6 possiamo vedere due esempi di grafi. Infine, se due archi hanno la stessa origine e destinazione, si parla di archi *paralleli* o archi *multipli*; se un arco connette un vertice a se stesso, parliamo di *autoanello*. I grafi senza autoanelli e archi multipli si dicono grafi *semplici*.

Queste sono, nella maggior parte dei casi, le descrizioni che si possono fare dei grafi. Per ulteriori approfondimenti, teoremi e algoritmi di ricerca si veda [10]. Va ricordato che un grafo non orientato può sempre essere trasformato nell'equivalente orientato sostituendo tutti gli archi (u, v) con una coppia di archi (u, v) e (v, u). Questa proprietà è molto importante perché ci permetterà di trattare i grafi non orientati come orientati, vantaggio che vedremo in seguito durante l'implementazione degli algoritmi.

Definizione 3.1. Un grafo si dice *connesso* se per ogni coppia di vertici esiste un percorso che li unisce.

Definizione 3.2. Sia $G = (V, A)$ un grafo connesso. Presi $u, v \in V$, la *distanza* tra u e v, $dist(u, v)$, è il minimo numero di archi in un cammino che va da u a v. Nel caso di grafo non orientato vale $dist(u, v) = dist(v, u)$. Diciamo anche che la distanza tra un vertice e se stesso è $dist(u, u) = 0$.

Nel lavoro seguente, i grafi che analizzeremo saranno sempre grafi semplici, non orientati e connessi, come quelli in Figura 5, e all'occorrenza, ci ridurremo sempre a questo caso. Dopo aver elencato le varie descrizioni del grafo come struttura dati, enunciamo alcune misure che possiamo fare su di esso.

Definizione 3.3. Sia $G = (V, A)$ un grafo connesso. Preso $u \in V$, il *raggio* $r(u)$ di u è la distanza tra u e il vertice più lontano raggiungibile da u.

Definizione 3.4. Sia $G = (V, A)$ un grafo connesso con $|V| = n$. Preso $u \in V$, sia $N(h, u)$ il numero di vertici $v \in V$ per cui $dist(u, v) \leq h$. Allora il *raggio effettivo* si definisce con:

$$r_{\text{eff}}(u) = \min \{h : N(h, u) \geq 0{,}9 \cdot n\}.$$

Il raggio effettivo di un vertice u è il valore della minima distanza per cui u raggiunge almeno il 90% di tutti i vertici del grafo. Nella definizione consideriamo anche il caso in cui il vertice raggiunge se stesso, motivo che porta a $0{,}9 \cdot n$ invece che a $0{,}9 \cdot (n-1)$. Anche se il raggio è una stima importante nell'analisi dei grafi, fissiamo l'attenzione su un altro aspetto ugualmente importante: il diametro.

Definizione 3.5. Sia G un grafo semplice, non orientato e connesso. Il *diametro* $d(G)$ di G è la massima distanza tra due vertici di G. Usando la definizione di *raggio*, il diametro è $d(G) = \max_u\{r(u)\}$.

Definizione 3.6. Sia $G = (V, A)$ un grafo semplice, non orientato e connesso con $|V| = n$. Sia $N(h)$ il numero di coppie $u, v \in V$ tali che $dist(u, v) \leq h$. Allora il *diametro effettivo* di G è definito:

$$d_{\text{eff}}(G) = \min \{h : N(h) \geq 0{,}9 \cdot n^2\}.$$

Il diametro effettivo è la *distanza* minima per cui il 90% di tutte le coppie di vertici di G possono raggiungersi l'un l'altra. Nella definizione di "coppie di vertici" consideriamo sia la coppia (u, v) che quella (v, u) e aggiungiamo anche le n coppie formate da un vertice con se stesso, raggiungibili alla distanza $h = 0$. In totale sono:

$$\binom{n}{2} + \binom{n}{2} + n = 2\frac{n(n-1)}{2} + n = n(n-1) + n = n^2$$

che giustifica la presenza del valore n^2 nella formula.

Nella 3.6 abbiamo utilizzato il 90% come soglia per definire il diametro effettivo. Ciò non è dovuto a vincoli teorici ma semplicemente a una prassi che si è diffusa. Sarebbero valide descrizioni anche con il 60-70%, ma la questione esula da questo lavoro e ci siamo attenuti a ciò che abbiamo trovato in letteratura, dove l'idea è quella di fare una stima "reale" del diametro, senza che questa sia alterata da "code" di vertici statisticamente irrilevanti.

A questo punto ritorniamo alle domande fatte nell'introduzione: come si fa a calcolare il diametro di un grafo di grandi dimensioni? Come fare se si hanno dei vincoli di risorse (poche) e di spazio (ancora meno)? Rispondiamo con la presentazione dell'algoritmo HADI, su cui è centrato buona parte del lavoro di questa tesi.

3.2 L'ALGORITMO HADI

HADI [12] (HAdoop DIameter) è un algoritmo per calcolare in modo approssimato il diametro effettivo di un grafo. Accanto alla versione sequenziale, facile da comprendere e veloce da implementare, ve n'è una che opera in parallelo, sfrutta il paradigma di programmazione MapReduce ed è pensata per essere realizzata con il framework Apache Hadoop. HADI si presenta come efficiente, scalabile e ottimo per gestire grafi di grandi dimensioni, dal giga al terabyte. Vediamo in cosa consiste.

3.2.1 Idea generale

L'idea algoritmica alla base è la seguente. Dato un grafo $G = (V, A)$ con $|V| = n$ e $|A| = m$, calcoliamo le distanze minime che ogni vertice ha con tutti gli altri vertici del grafo. Con questi valori posso computare $N(h)$ al variare di h fino a trovare il valore h^* per cui il 90% delle coppie di vertici sono collegati tra loro. Immaginiamo che gli archi del grafo siano dei canali di comunicazione per mezzo del quale i vertici possono inviare delle informazioni agli altri vertici:

1. All'inizio, ogni vertice possiede come unica informazione l'identificatore di se stesso.

2. All'i-esima iterazione, $i \geq 1$, ogni vertice v invia a tutti i suoi vicini (i vertici a distanza 1 da esso) la lista di vertici che possiede.

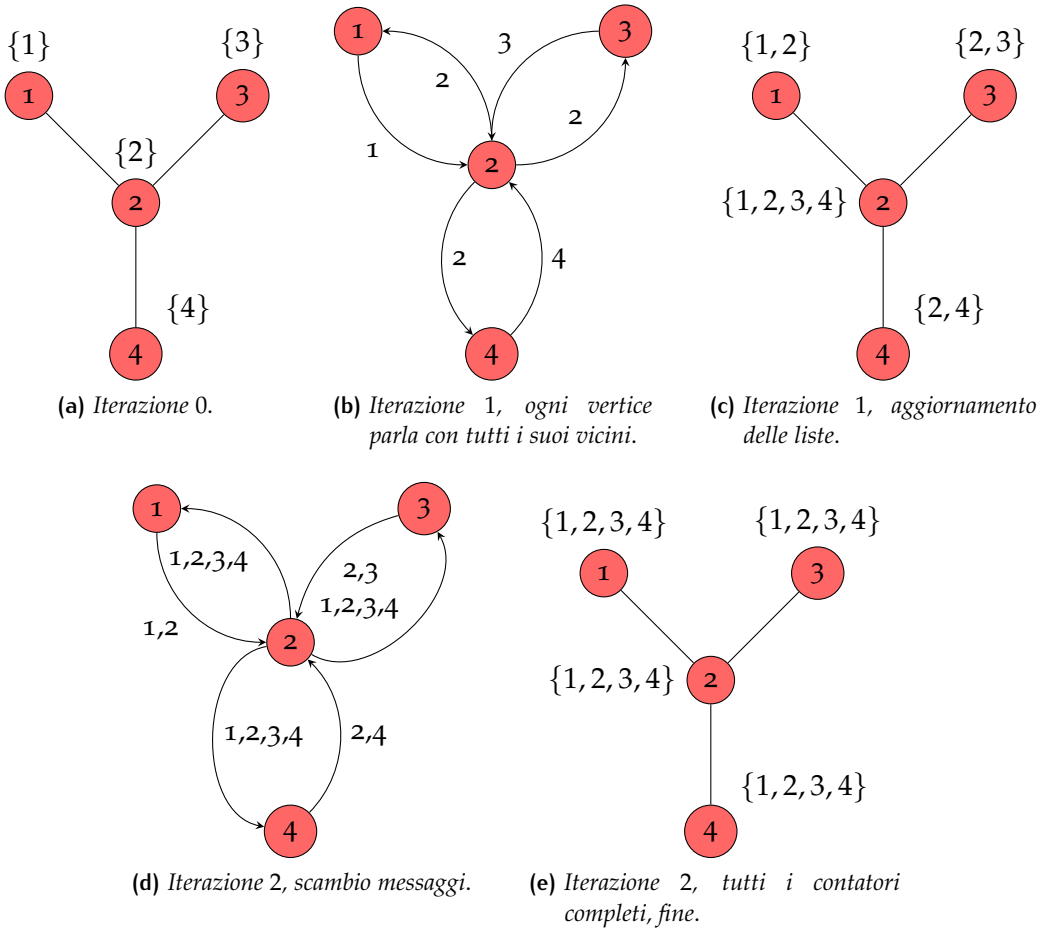
3. Il vertice, che riceve una o più di queste liste, esegue l'unione insiemistica di queste liste e di quella da lui posseduta, ottenendo un nuovo insieme di (identificatori di) vertici. Questo insieme rappresenta $N(i, v)$, il numero di vertici raggiungibili da v a una distanza $\leq i$.

4. Quando un vertice v all'iterazione h, dopo un'operazione di aggiornamento, non modifica il suo insieme, significa che all'iterazione $h-1$ è stato raggiunto dal o dai vertici che sono alla distanza massima da esso e non verrà più aggiornato in future iterazioni. Infatti, supponiamo sia u il vertice con la distanza maggiore, $h-1$, da v. Se all'iterazione h l'insieme di vertici di v fosse aggiornato inserendo un vertice z, significherebbe che esiste un vertice che dista h da v, assurdo perché il vertice più distante è per ipotesi u a distanza $h-1$. Viceversa, supponiamo che all'iterazione h l'insieme di v smette di essere aggiornato. Se esistesse un vertice z la cui distanza massima da v fosse h — e sia u il suo immediato precedessore nel cammino minimo (v, z) — , allora nello sviluppo dell'algoritmo si otterrebbe che:

   - all'iterazione $h-1$, v incrementa il suo insieme inserendo u;
   - all'iterazione h inserisce z;
   - dall'iterazione $h+1$ in poi non riceve nessun altro aggiornamento;

   il che porta a un assurdo, perché per ipotesi l'insieme appartenente a v non viene più aggiornato a partire dall'iterazione h.

   Perciò, quando non avvengono più modifiche, possiamo dire che il vertice v ha raggiunto il suo raggio $r(v)$.



(a) *Iterazione* 0.

(b) *Iterazione* 1, *ogni vertice parla con tutti i suoi vicini.*

(c) *Iterazione* 1, *aggiornamento delle liste.*

(d) *Iterazione* 2, *scambio messaggi.*

(e) *Iterazione* 2, *tutti i contatori completi, fine.*

Figura 7: Esempio di calcolo del diametro con HADI. Il diametro è il valore dell'ultima iterazione.

5. Quando tutti i vertici del grafo non modificano più i loro insiemi, allora l'algoritmo può terminare. Il valore dell'iterazione corrente meno uno è proprio $\max_u\{r(u)\}$ e quindi è il diametro del grafo, mentre dai valori $N(h)$, $0 \leq h \leq d(G)$ ottenuti possiamo calcolare il diametro effettivo.

Questa procedura è semplice e dà una misura esatta del diametro del grafo. In Figura 7 possiamo osservare un esempio su un grafo $n = 4$ e $m = 3$. Tuttavia, la descrizione nasconde un'insidia: ogni vertice possiede una lista di identificatori che, con l'avanzare del ciclo, arriverà a contenere gli identificatori di tutti i vertici del grafo. Ciò significa che serve uno spazio di memorizzazione $O(n^2)$ per far funzionare la computazione! Per fare un esempio, se abbiamo un grafo con $n = 10^9$ vertici e ogni identificatore ha un peso di 32 bit, la memoria totale di cui abbiamo bisogno alla fine dell'ultima iterazione sarebbe $32 \cdot (10^{9 \cdot 2}) = 32 \cdot 10^{18} \approx 4$ exabyte!!! Il supercomputer più potente attualmente in circolazione, il Tianhe-2 della Sun Yat-sen University, in Cina, ha un memoria di un petabyte e una memoria secondaria di 12 petabyte, tre ordini di grandezza inferiori a quello che servirebbe in questo caso.

Possiamo allora intuire che calcolare il valore esatto del diametro è una strada non percorribile. Perché la computazione sia possibile in pratica, dove ho un cluster con più macchine che lavorano in parallelo, serve che lo spazio di memoria utilizzato

Codice 2: L'algoritmo di *Flajolet-Martin* per l'aggiornamento di un registro BITMAP.

```
for i = 0 to L–1 do
    BITMAP[i] <– 0;
end
for each x in M do
    index <– ρ(hash(x));
    if (BITMAP[index] == 0) then
        BITMAP[index] <– 1;
    end
end
```

da ogni macchina sia sublineare nella taglia del grafo e che, in generale, lo spazio complessivo sia lineare o poco più rispetto alla taglia del grafo. Oltrepassando questa soglia, il costo computazionale diventa ingestibile.

Per tutti questi motivi, HADI utilizza un metodo di conteggio dei vertici approssimato e che fa uso di una quantità di memoria limitata: i Probabilistic Counter di Flajolet-Martin.

3.2.2 Probabilistic Counting

HADI permette la stima del diametro con un costo "ragionevole" perché i vertici, che devono memorizzare i molti identificatori, utilizzano dei contatori che approssimano il numero di elementi di un insieme. Questi contatori sono i Probabilistic Counter [8], detti anche contatori di *Flajolet-Martin* dal nome dei loro inventori, e hanno il pregio di dare una stima unbiased della cardinalità di un insieme al costo di $O(\log n)$ spazio.

L'idea che sta dietro a questi contatori è la seguente:
Abbiamo un insieme M di elementi che dobbiamo contare. Supponiamo di avere una funzione hash $h(x) : M \to [0, 1, \ldots, 2^L - 1]$ che distribuisce uniformemente gli oggetti dell'insieme in un valore intero tra 0 e $2^L - 1$. Inoltre, utilizziamo una stringa binaria di L bit come struttura di supporto, con $L \in O(\log_2 n)$. Per seguire la descrizione di Flajolet-Martin immaginiamo la stringa binaria da sinistra verso destra, con il bit meno significativo a sinistra e quello più significativo a destra. Se la distribuzione è uniforme possiamo dire che un qualsiasi bit $\in \langle 0 \ldots L-1 \rangle$ avrà valore 0 o 1 con probabilità $\frac{1}{2}$, quindi un pattern del tipo 0^k1 comparirà con probabilità 2^{-k-1}. Introduciamo anche la funzione $\rho(y)$ che restituisce la posizione del bit a uno meno significativo di y, dove y è un intero non negativo codificato in binario. Allora l'idea è di registrare sulla stessa stringa di bit, una BITMAP$[0 \ldots L-1]$, i risultati della codifica degli elementi da parte della funzione hash, come descritto nel Codice 2.

Osserviamo che BITMAP$[i]$ è uguale a 1 se, dopo l'esecuzione, la funzione hash ha creato un numero del tipo $0^i1 \ldots$ Per costruzione, il vettore BITMAP non dipende dalla frequenza con compaiono gli elementi di M, ma solamente dall'insieme dei valori creati in output dalla funzione di hash. Quindi, tornando all'insieme M di partenza, se n sono gli elementi distinti di M, per l'ipotesi di uniformità posso affermare che sia avvenuto l'accesso a BITMAP[0] circa $n/2$ volte, a BITMAP[1] $n/4$ volte... In generale, possiamo affermare che BITMAP$[i]$ sarà quasi sicuramente 0 per $i \gg \log_2 n$ e 1 per $i \ll \log_2 n$, con un misto di 0 e 1 per $i \approx \log_2 n$. Sia R l'indice del bit a zero più a

sinistra in BITMAP. Sotto l'ipotesi che gli elementi dell'insieme sono uniformemente distribuiti, Flajolet e Martin hanno ottenuto che il valore R, inteso come variabile aleatoria, ha un valore atteso vicino a:

$$\mathbb{E}(R) \approx \log_2 \phi n, \quad \phi = 0{,}773\,51\cdots, \quad \sigma(R) \approx 1{,}12 \tag{1}$$

I risultati del coefficiente correttivo ϕ e della deviazione standard σ in (1) sono stati ottenuti dopo un'analisi rigorosa della distribuzione di probabilità di R (rimandiamo all'articolo originale [8] per la dimostrazione). A questo punto, la stima degli elementi dell'insieme M si può ottenere con:

$$\frac{1}{\phi} 2^R$$

I due scienziati hanno provato che questa stima è circa un ordine di grandezza inferiore al risultato esatto, perciò serve un modo per affinare ancor di più il conteggio. La prima idea è quella di usare K funzioni di hash per calcolare altrettante differenti BITMAP, dove K è un parametro scelto dal progettista. In questo modo otteniamo le variabili aleatorie $R^{\langle 1\rangle}, R^{\langle 2\rangle}, \ldots, R^{\langle K\rangle}$ e facendo la media:

$$A = \frac{R^{\langle 1\rangle} + R^{\langle 2\rangle} + \cdots + R^{\langle K\rangle}}{K} \tag{2}$$

otteniamo una variabile aleatoria con stima e deviazione standard di:

$$\mathbb{E}(A) \approx \log_2 \phi n, \quad \sigma(A) \approx \sigma_\infty / \sqrt{K}, \quad \sigma_\infty = 1{,}121\,27\cdots. \tag{3}$$

In questo modo raggiungiamo una stima più precisa (con $K = 64$ si ottiene una misura con un errore standard del 10%) ma al prezzo di aumentare il costo computazionale a causa dei molteplici calcoli eseguiti dalle funzioni hash. Questa prima idea migliora la stima ma è inefficiente da un punto di vista pratico. L'idea successiva, invece, raggiunge gli stessi livelli di accuratezza senza il bisogno di K funzioni hash. Si utilizzano K BITMAP indicizzate da 0 a $K-1$. Con la nostra unica funzione di hash h, per ogni $x \in M$, calcoliamo $\alpha = h(x) \mod K$. Allora α sarà l'indice del contatore da aggiornare, e verrà aggiornato con il valore $\lfloor h(x)/K \rfloor$. Alla fine ricaviamo sempre gli $R^{\langle j\rangle}$ e ne calcoliamo la media come in (2). Nell'ipotesi che la distribuzione sia uniforme, alla fine del calcolo ogni BITMAP sarà stata scritta in media da n/K elementi. Perciò il valore

$$\frac{1}{\phi} 2^{\frac{1}{K}\sum_{i=1}^{K} R_i}$$

è una buona approssimazione di n/K che, moltiplicata per K, fornisce la stima del numero di elementi di M. L'analisi di questa procedura porta a un errore standard nella stima pari a $0{,}78/K$. Nella Tabella 5 possiamo vedere la percentuale di errore standard al variare del numero di BITMAP utilizzate per la stima.

Da questi risultati osserviamo che effettuare l'unione di due contatori A e B diventa semplice perché basta fare un OR bit a bit tra le stringhe di A e quelle di B. Allora, tornando al nostro problema, ci viene immediato pensare a come poter utilizzare questi contatori per il nostro calcolo del diametro:

- Ogni vertice v possiede, all'h-esima iterazione, un contatore di Flajolet-Martin con K stringhe $b(h, v)$. Questo rappresenta i vertici che v può raggiungere in almeno h passi.

Tabella 5: Valori dell'errore standard al variare del numero di BITMAP utilizzate in un contatore.

K	Errore standard %
2	55,15
4	39
8	27,58
16	19,5
32	13,79
64	9,75
128	6,89
256	4,88
512	3,45
1024	2,44

- All'iterazione successiva, v invia ai suoi vicini il suo contatore $b(h, v)$.
- Riceve dai vertici vicini i rispettivi contatori e calcola il nuovo valore del proprio:

$$b(h+1, v) = b(h, v)\text{BIT-OR}\{b(h, u) : (v, u) \in A\}$$

  .
- All'ultima iterazione, tutti i contatori non vengono più modificati e l'algoritmo termina.

Dopo h iterazioni, il contatore del vertice v rappresenta la *neighborhood function* $N(h, v)$, cioè la stima del numero di vertici raggiungibili da v in almeno h passi.

$$N(h, v) = \frac{1}{0{,}773\,51} 2^{\frac{1}{K}\sum_{j=1}^{K} b_j(v)}$$

dove $b_j(v)$ è la posizione del bit zero più a sinistra della j-esima BITMAP di v. Quando raggiungiamo $h_{\max}$ (che, attenzione, è la stima del diametro...) abbiamo tutte le informazioni per poter calcolare il raggio effettivo per ogni vertice e il diametro effettivo del grafo. Procediamo così:

- $r_{\text{eff}}(v)$ è il più piccolo h tale che $N(h, v) \geq 0{,}9 \cdot N(h_{\max}, v)$.
- $d_{\text{eff}}(G)$ è il più piccolo h tale che $N(h) = \sum_{v=1}^{|V|} N(h, v) = 0{,}9 \cdot N(h_{\max})$. Se $N(h) > 0{,}9 \cdot N(h_{\max}) > N(h-1)$, allora $d_{\text{eff}}(G)$ si può ricavare come interpolazione lineare di $N(h)$ e $N(h-1)$ in questo modo:

$$d_{\text{eff}}(G) = (h-1) + \frac{0{,}9 \cdot N(h_{\max}) - N(h-1)}{N(h) - N(h-1)}$$

Ancora due considerazioni. Affinché la stima $d_{\text{eff}}(G)$ abbia un significato pratico, deve essere arrotondata all'intero successivo più vicino. Un diametro effettivo di 4,1 o

Tabella 6: Alcuni simboli utilizzati nell'algoritmo HADI

Simbolo	Definizione
$N(h)$	Somma del numero di coppie di vertici raggiungibili a distanza $\leq h$
$N(h, v)$	Numero di vertici raggiungibili da v a distanza $\leq h$
$b(h, v)$	Contatore di Flajolet-Martin del vertice v dopo h passi
NewFMCounter(i)	Funzione che crea un nuovo contatore inizializzato a i

4,9 significano entrambi che solo da $d(G) = 5$ in poi raggiungo la soglia del 90%. Un'altra osservazione riguarda il calcolo di $N(h_{\max})$ e $N(h_{\max}, v)$. Dalle definizioni 3.4 e 3.6 risulta che $N(h_{\max}, v) = n$ e $N(h_{\max}) = n^2$. Perché spingere il ciclo fino ad $h_{\max}$? Conoscendo a priori il loro valore, potremmo fermare le iterazioni quando il valore h soddisfa le definizioni, senza arrivare ad $h_{\max}$. Questo non è possibile a causa della natura "approssimata" delle misure di $N(h)$ e $N(h, v)$, che vengono costruite iterativamente con misure che sono solo delle stime. Per quanto possano essere precise, non abbiamo la certezza che corrispondano ai valori teorici.

3.2.3 Pseudocodice

Abbiamo ora tutti gli strumenti per comprendere il codice dell'algoritmo. Il numero K di stringhe di un contatore, che d'ora in poi chiameremo "registri", è impostato di base a 32, e il suo valore è modificabile per ottenere un'accuratezza maggiore (secondo la Tabella 5) al prezzo di incrementare il numero di operazioni bit a bit e perciò di aumentare il tempo di calcolo complessivo. Il valore MaxIter è un limite superiore al valore del diametro, perciò è anche il limite al numero di iterazione che fa l'algoritmo. Nell'algoritmo originale MaxIter è impostato di base a 256, corretto per la computazione sulla maggior parte dei grafi reali, che solitamente non hanno un diametro elevato. Nei casi eccezionali va però opportunamente modificato. Nel Codice 3 è presentata una versione sequenziale di HADI.

Lemma 3.2.1. *La complessità temporale di HADI-sequenziale è*

$$T(n, m) \in O\big(d(G) * m\big) \quad \textit{essendo} \;\; K \in O(1).$$

Dimostrazione. Il costo computazionale maggiore è dato dal triplo ciclo innestato (righe 4-20 in 3) in cui si eseguono i BIT-OR. Il numero di queste operazioni, in un ciclo, è proporzionale al numero di archi e di vertici del grafo, ed è:

$$O\Bigg(\sum_{h=1}^{d(G)}\sum_{l=1}^{K}(2m+n)\Bigg) = O\Bigg(\sum_{h=1}^{d(G)}\sum_{l=1}^{K}(m)\Bigg) = O\big(d(G) * K \cdot m\big) = O\big(d(G) * m\big)$$

supponendo che nel nostro grafo $n \ll m$ e K una costante trascurabile. □

Lemma 3.2.2. *La complessità in spazio di HADI-sequenziale è*

$$O\big(K \cdot n \log n\big)$$

Dimostrazione. Abbiamo n vertici, ciascuno con un contatore di Flajolet-Martin formato da K registri ognuno dei quali è ha un numero di bit dell'ordine di $O(\log_2 n)$. Trascurando i fattori costanti e la base del logaritmo, otteniamo:

$$K \cdot n \cdot \log_2 n \simeq K \cdot n \log n$$

Codice 3: HADI-sequenziale.

```
   Input: G = (V, A), |V| = n, |A| = m; MaxIter; K
   Output: r_eff(i) per ogni vertice; d_eff(G)
1  for i = 1 to n do
2      b(0, i) <- NewFMCounter(i);
3  end
4  for h = 1 to MaxIter do
5      Changed <- 0;
6      for i = 1 to n do
7          for ℓ = 1 to K do
8              b_ℓ(h,i) <- b_ℓ(h-1,i) BIT-OR {b_ℓ(h-1,j) : (i,j) ∈ A};
9              end
10             if (∃ ℓ : b_ℓ(h,i) ≠ b_ℓ(h-1,i)) then
11                 Changed <- Changed + 1;
12             end
13         end
14     end
15     N(h) <- ∑_i N(h, i);
16     if (Changed == 0) then
17         h_max <- h;
18         break;
19     end
20 end
21 for i = 1 to n do
22     r_eff(i) <- *il più piccolo h' : N(h',i) ≥ 0.9 N(h_max,i)*;
23 end
24 d_eff(G) <- *il più piccolo h' : N(h') = 0.9 N(h_max)*;
```

□

Quello appena descritto è l'algoritmo HADI come presentato in [12] e serve per ottenere una stima del raggio effettivo e del diametro effettivo di un grafo.

3.2.4 HADI per la misura del diametro

Nel lavoro che segue ci serviremo di HADI non tanto per queste misure, ma per la stima del solo *diametro*. Infatti, anche se nel codice non viene evidenziato, si ottiene la stima del diametro quando il ciclo principale termina. Alla riga 17 è impostato il valore di h_{max} ad h. A questa iterazione è avvenuto che, dopo lo scambio di messaggi tra i vicini, nessuno di questi ha modificato il proprio contatore. Questo significa che tutti i vertici del grafo hanno raggiunto il valore della distanza massima per arrivare a qualsiasi altro vertice, cioè il diametro. Per questo, in un'ipotetica implementazione basta aggiungere, dopo la riga 17 del codice, un'altra riga con il comando d(G) <- (h−1) e otteniamo il risultato cercato.

HADI fin qui descritto è un buon algoritmo. Rimane il fatto che una procedura del genere non può essere implementata in una singola macchina. Per un grafo con miliardi di nodi e archi, uno spazio di memoria $O(n \log n)$ è comunque troppo per poter eseguire una stima in locale. L'approccio vincente è una parallelizzazione del calcolo fatta con il paradigma MapReduce.

3.2.5 Implementazione in parallelo

Vediamo com'è strutturato HADI nella sua versione parallela. Non presentiamo il codice, che si può trovare nell'articolo originale, ma descriviamo la procedura nelle sue operazioni più significative. L'implementazione si basa sul paradigma MapReduce, in particolare, sul framework Hadoop e il suo file system HDFS.

HADI è un algoritmo *disk based*, pensato per trattare dati di dimensioni troppo elevate per una computazione *in-memory*, per questo si serve del paradigma MapReduce che si appoggia alla memoria secondaria per eseguire il calcolo. Ipotizzando la computazione su un grafo $G = (V, A)$, i due tipi di dato che vengono salvati in memoria durante tutto il processo sono:

- Archi, nel formato $(srcId, dstId)$;
- Contatori, nel formato $(vertexId, flag, Register_1, Register_2, \ldots, Register_K)$;

dove *flag* si compone di due informazione:

- *Changed*, una variabile identificatrice che indica se il contatore è stato modificato;
- $N(h, i)$, il numero di vertici raggiunti da i in almeno h passi.

L'idea generale è quella di replicare il contatore di un vertice j esattamente x volte, dove x è il grado entrante del vertice j. Queste repliche sono chiamate *contatori parziali* e li rappresentiamo con $\hat{b}(h, j)$. Una $\hat{b}(h, j)$ è utilizzata per aggiornare un contatore $b(h, i)$, contatore del vertice i dove (i, j) è un arco del grafo. In HADI-parallelo ci sono tante iterazioni ciascuna composta di tre Stage, ognuno dei quali è un'operazione MapReduce. Usiamo la lettera h per simboleggiare l'iterazione corrente ($h = 1$ all'inizio). Uno schema di tutto il ciclo è descritto in Figura 8, dove ile due linee di *mapper* e *reduce* simboleggiano il fatto che ci sono più elaborazioni in parallelo di map e reduce.

Stage 1

- Input: Dal disco ricevo in ingresso la lista di tutti gli archi del grafo $E = \{(i, j)\}$ e i contatori $C = \{(i, b(h-1, i))\}$, uno per ogni vertice.
- Map: I contatori li lascio inalterati, mentre gli archi (i, j) li restituisco scambiando la chiave con il valore, (j, i).
- Reduce: Per ogni arco (j, i) creo una coppia con chiave i e contatore parziale $\hat{b}(h-1, j)$, $(i, \hat{b}(h-1, j))$. Il contatore parziale è una copia del contatore $(j, b(h-1, j))$. Oltre a questi, creo anche il contatore parziale di i stesso, $(i, \hat{b}(h-1, i))$.
- Output: Scrivo su disco la lista di tutti i contatori parziali associati ai vertici, $P = \{(i, \hat{b}(h-1, j))\}$.

In questo primo Stage generiamo le coppie (chiave, valore), dove la *chiave* è l'identificatore del vertice i, mentre il *valore* è il contatore parziale $\hat{b}(h-1, j)$ dove j spazia su tutti i vertici adiacenti a i.

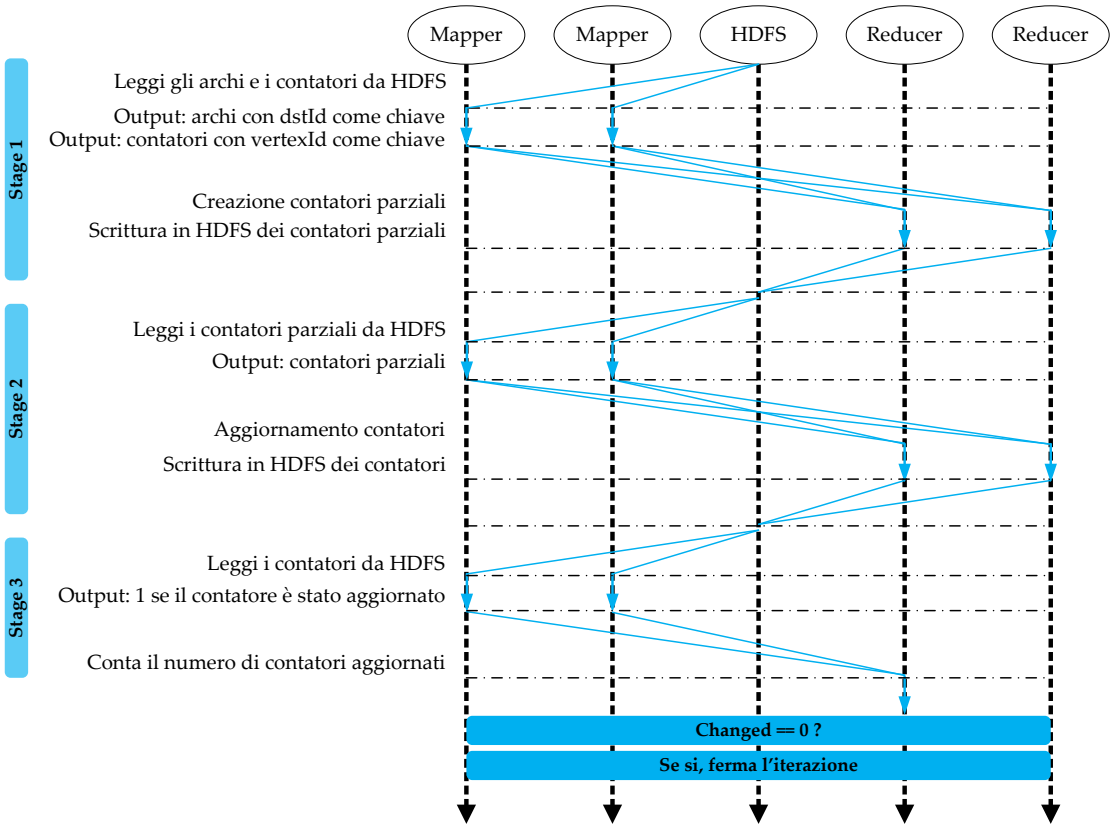


Figura 8: Schema del ciclo di operazione di HADI nella sua implementazione parallela.

Stage 2

- Input: Dal disco prendiamo la lista dei contatori parziali creati nello Stage 1, $P = \{(i, \hat{b}(h-1, j))\}$.
- Map: Lasciamo inalterati gli oggetti, e li passiamo alla reduce (*Identity Mapper*).
- Reduce: Per ogni chiave i, facciamo l'unione di tutti i contatori parziali $\hat{b}(h-1, j))$ e otteniamo $b(h, i)$.
- Output: La lista dei contatori aggiornati $C = \{(i, b(h, i))\}$.

Aggiorniamo il contatore del vertice i combinando il suo contatore parziale $\hat{b}(h-1, i)$ con tutti quelli dei suoi vicini $\hat{b}(h-1, j)$. Dopo quest'operazione aggiorniamo il *flag*, ponendo $Changed = 1$ se il contatore è stato modificato in almeno in uno dei suoi K registri, e memorizzando in $N(h, i)$ l'attuale numero di vertici raggiunti. Notiamo che $N(h, i)$ è impostato solo in questa h-esima iterazione e non sarà più modificato.

L'output ottenuto in questo Stage, i contatori $C = \{(i, b(h, i))\}$, andranno a formare l'input di un nuovo Stage 1 (assieme agli archi E, che rimangono inalterati).

Stage 3

- Input: $C = \{(i, b(h, i))\}$.
- Map: per ogni i, crea $(i, N(h, i))$ e, se $Changed = 1$, $(i, Changed)$.

- Reduce: Somma tutti i *Changed* e somma degli $N(h,i)$ per ottenere $N(h)$.
- Output: $N(h)$.

Calcoliamo il numero di contatori modificati e sommiamo tutti gli $N(h,i)$ per ottenere $N(h)$. Se nessun contatore è stato modificato la procedura si interrompe e l'algoritmo termina.

Lemma 3.2.3. *Sia G un grafo con n vertici, m archi e diametro d. Supponiamo di avere M macchine per il calcolo. La complessità temporale di HADI-parallelo è*

$$O\left(d * \frac{m+n}{M} \log\left(\frac{m+n}{M}\right)\right)$$

Dimostrazione. Nello Stage 1, per passare dal map alla reduce, Hadoop esegue lo shuffle dei dati. Abbiamo $m+n$ elementi ordinati con un *merge sort*. Quest'operazione è la più costosa dell'intero algoritmo, ne domina la complessità ed è eseguita d volte. □

Lemma 3.2.4. *La complessità in spazio di HADI-parallelo è*

$$O\Big(K(m+n)\log n\Big)$$

Dimostrazione. Alla fine dello Stage 1 l'algoritmo crea $2m+n$ contatori parziali, ognuno con un costo di $K \log_2 n$ bit. Trascurando i fattori costanti, otteniamo

$$(2m+n) \cdot K \cdot \log_2 n \in O\Big(K(m+n)\log n\Big)$$

□

3.3 HYPERLOGLOG

L'algoritmo HADI sarà la base per lo sviluppo di questa tesi. Prima di procedere con l'analisi e l'implementazione su Spark, torniamo al problema di stimare il numero di elementi di un insieme. La tecnica descritta in precedenza, Probabilistic Counting, ha il pregio di essere semplice da utilizzare nella pratica e di fornire una misura con un errore standard molto basso. Ha però un difetto: la sua dimensione. Ogni registro ha un peso dell'ordine di $O(\log n)$, cioè può arrivare fino a 32 bit e, specialmente nei calcoli che dobbiamo effettuare qui, con miliardi di vertici e archi, questo peso non è per nulla trascurabile. Anche il fattore K, tralasciato nella analisi teorica dell'algoritmo, nella realtà è un dato fondamentale per valutare le prestazioni; motivo che ci porterà a tenerlo sempre in considerazione per gli studi successivi.

Probabilistic Counting è stato descritto da Flajolet e Martin nel 1985. Da allora sono state inventate altre tecniche per la stima della cardinalità. Un significativo passo avanti è stato fatto nel 2003 con un lavoro di Flajolet e Durand. I due autori hanno creato un nuovo tipo di contatore, Loglog Counter [6], che modifica l'algoritmo Probabilistic Counting e consente di eseguire una stima dell'insieme utilizzando registri dell'ordine di $\log\log n + O(1)$ bit. In pratica, ogni stringa del contatore è formata da 5-6 bit! Un incremento delle prestazioni in spazio di un fattore esponenziale, al prezzo però di un aumento dell'errore standard nella misura di circa il doppio se confrontato con

Codice 4: L'algoritmo HyperLogLog.

Input: M, $m = 2^b$, $b > 0$, h, ρ
Output: E = |M|
for i = 1 to m **do**
    R[i] <- $\{-\infty\}$;
end
for each x in M **do**
    y <- h(x);
    j <- 1 + $\langle x_1 x_2 \dots x_b \rangle_2$;
    w <- $x_{b+1} x_{b+2} \dots$;
    R[j] <- max(R[j], ρ(w));
end
Z <- $(\sum_{j=1}^{m} 2^{-R[j]})^{-1}$
E <- $\alpha_m m^2 Z$

l'algoritmo precedente; $1{,}30/\sqrt{K}$ contro $0{,}78/\sqrt{K}$. Anche se il risultato è comunque soddisfacente, nel 2007 è uscito un nuovo risultato da parte di Flajolet e il suo gruppo di lavoro. Si tratta di una modifica al Loglog Counter che mantiene inalterato lo spazio utilizzato e migliora l'errore standard. Questo tipo di contatore, HyperLogLog [7], è uno degli strumenti cardine del lavoro che sarà presentato successivamente, ed è qui brevemente presentato.

3.3.1 Idea Algoritmica

La struttura di base dell'algoritmo è simile a Probabilistic Counting. Dobbiamo stimare la cardinalità di M e abbiamo la nostra funzione di hash $h(x) : M \to [0, 1, \dots, 2^m - 1]$ che mappa i valori in modo uniforme nell'intervallo $[0, \dots, 2^m - 1]$, $\rho(y)$ che identifica la posizione del bit a uno più a sinistra, in y (sempre la notazione con il bit meno significativo a sinistra e quello più significativo a destra) e poi abbiamo m registri in cui distribuiamo i risultati della funzione di hash. L'idea innovativa di Loglog Counter è stata quella di basare la stima sul più grande valore ottenuto da $\rho(y), \forall x : h(x) = y$ e di utilizzare una funzione di valutazione centrata sulla media geometrica. L'ulteriore modifica di HyperLogLog è quella di servirsi della media armonica invece di quella geometrica. La media armonica qui ha l'effetto di migliorare la distribuzione di probabilità, perciò di ridurre la varianza e aumentare la qualità della stima. L'analisi di HyperLogLog stabilisce che questi contatori hanno un errore standard di $1{,}04/\sqrt{m}$ con un costo in spazio di $\log \log n + O(1)$ bit. Per fare un esempio, usando $m = 2048$, valori di hash su 32 bit, possiamo stimare con un errore del 2% cardinalità dell'ordine di $n = 10^9$ usando 1,5 kilobyte di memoria.

3.3.2 Pseudocodice

Sia $h : M \to [0, 1] \equiv \{0, 1\}^\infty$ una funzione di hash che mappa i valori di M nel dominio dei numeri binari e sia $\rho(y)$, con $y \in \{0, 1\}$, la posizione del bit a uno più a sinistra, più uno ($\rho(0001 \dots) = 4$). I valori risultanti dalla funzione hash sono divisi tra gli m

sottoinsiemi $R_1, \ldots, R_m$ sulla base dei primi b bit del valore $y = h(x)$. Dato $N = R_j$, il risultato che prendiamo in considerazione da ogni R_j per la stima é:

$$Max(N) = \max_{y \in N} \rho(y)$$

L'algoritmo raccoglie, nei registri $R[j]$ i valori $Max(R_j)$ per $j = 1, \ldots, m$. Quando tutti gli elementi dell'insieme M sono stati scansionati, viene calcolata la funzione *indicatrice*:

$$Z = \left(\sum_{j=1}^{m} 2^{-R_j} \right)^{-1}$$

per poi restituire la versione di Z normalizzata secondo la media armonica di 2^{R_j} nella forma:

$$E = \frac{\alpha_m m^2}{\sum_{j=1}^{m} 2^{-R_j}} \qquad \text{con} \qquad \alpha_m = \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1}$$

L'intuizione dietro a questa procedura è la seguente. Sia n la cardinalità sconosciuta di M. Ogni sottoinsieme R_j conterrà circa n/m elementi. In generale, alla fine della computazione ogni valore $Max(R_j)$ sarà vicino a $\log_2(n/m)$. La media armonica (qui mZ) della quantità 2^{Max} sarà dell'ordine di n/m. Per questo motivo, $m^2 Z$ dovrebbe essere una stima del valore n che stiamo cercando. Il valore α_m è una misura imposta dagli autori per correggere un bias moltiplicativo presente in $m^2 Z$. Il teorema seguente ci da un'indicazione dell'accuratezza di questa stima.

Teorema 3.1. *Applichiamo l'algoritmo HyperLogLog 4 a un insieme M ideale ($h(M)$ si distribuisce uniformemente in $[0, 2^m - 1]$) di cardinalità (ovviamente sconosciuta) n, con $m \geq 3$ registri, e sia E la variabile aleatoria che definisce il risultato della stima. Siano $\mathbb{E}_n$ e $\mathbb{V}_n$ il valore atteso e la varianza del nostro modello.*

1. *La stima E è* quasi *priva di bias, cioè*

$$\frac{1}{n}\mathbb{E}_n[E] \underset{n\to\infty}{=} 1 + \delta_1(n) + o(1), \quad \textit{dove} \quad |\delta_1(n)| < 5 \cdot 10^{-5} \ \textit{per} \ m \geq 16$$

2. *L'errore standard definito come $\eta_m = \frac{1}{n}\sqrt{\mathbb{V}_n[E]}$ soddisfa, per $n \to \infty$,*

$$\eta_m = \frac{1}{n}\sqrt{\mathbb{V}_n[E]} \underset{n\to\infty}{=} \frac{\beta_m}{\sqrt{m}} + \delta_2(n) + o(1), \quad \textit{dove} \quad |\delta_2(n)| < 5 \cdot 10^{-4} \ \textit{per} \ m \geq 16.$$

la costante β_m è delimitata, con $\beta_{16} = 1{,}106$, $\beta_{32} = 1{,}070$, $\beta_{64} = 1{,}054$, $\beta_{128} = 1{,}046$ e $\beta_\infty = \sqrt{3\log(2) - 1} = 1{,}038\,96$.

3.3.3 Implementazione

Questa tecnica di conteggio appare essere la migliore attualmente in circolazione, e sarà quella che utilizzeremo per portare l'algoritmo HADI su Spark. Un'implementazione già disponibile è quella prodotta da Paolo Boldi e Sebastiano Vigna dell'Università degli Studi di Milano. Questo contatore HyperLogLog, scritto in linguaggio Java,

Tabella 7: Valori dell'errore standard al variare del logaritmo del numero di registri per un contatore HyperLogLog.

$\log_2 K$	Errore standard %
4	26
5	18,4
6	12,99
7	9,18
8	6,49
9	4,59
10	3,25
11	2,3
12	1,62
13	1,15
14	0,812
15	0,574

è stato utilizzato per la stima delle varie misure di un grafo [3] mediante la creazione dell'algoritmo HyperANF, miglioramento del suo vecchio predecessore ANF. Il lavoro di questo gruppo si è focalizzato sul mining di grafi di grandi dimensioni con un approccio di calcolo parallelo sviluppato per una macchina multiprocessore. L'argomento è differente rispetto a quello che vogliamo studiare su questa tesi, tuttavia il lavoro svolto per implementare HyperLogLog è di pregevole fattura (bisogna darne atto) ed è un contributo prezioso di cui faremo uso.

3.3.4 Accuratezza della stima

Nonostante l'ambiente implementativo sia diverso, per ottenere il diametro del grafo l'algoritmo HyperANF esegue un'operazione presente anche su HADI: l'aggiornamento del contatore di un vertice come unione dei suoi vicini e il calcolo di $N(h), N(h, v)$. Questo è molto importante perché nell'articolo di Boldi-Vigna [3] è presentata un'analisi dell'accuratezza della stima che si ottiene misurando questi valori (analisi che, purtroppo, manca nell'articolo originale di HADI [12]). Allora possiamo servircene anche noi per capire che tipo di precisione otteniamo.

Consideriamo un grafo semplice, non diretto e connesso $G = (V, A)$, $|V| = n$ e sia $\hat{N}(h)$ l'output dell'algoritmo presentato in 3 a una fissata iterazione h. Possiamo vedere questo valore come variabile aleatoria

$$\hat{N}(h) = \sum_{v \in V} X_{v,h}$$

dove ogni $X_{v,h}$ è l'HyperLogLog che conta i vertici raggiunti da v in h iterazioni. Quello che vogliamo fare è limitare superiormente l'errore standard di $\hat{N}(h)$ (definito come

in 3.1). Per prima cosa, da [6] osserviamo che per un numero fissato m di registri, la deviazione standard di $X_{v,h}$ soddisfa

$$\frac{\sqrt{\mathbb{V}[X_{v,h}]}}{N(h,v)} \leq \eta_m,$$

dove η_m è l'errore standard garantito di un HyperLogLog. Per la subadditività della deviazione standard ($\sqrt{\mathbb{V}[A+B]} \leq \sqrt{\mathbb{V}[A]} + \sqrt{\mathbb{V}[B]}$) otteniamo la seguente:

Teorema 3.2. *L'output $\hat{N}(h)$ dell'algoritmo HADI all'h-esima iterazione è asintoticamente una stima* quasi *priva di bias del valore $N(h)$, cioè*

$$\frac{\mathbb{E}[\hat{N}(h)]}{N(h)} = 1 + \delta_1(n) + o(1) \quad \textit{per} \quad n \to \infty,$$

dove δ_1 è la stessa del Teorema 3.1.

Inoltre $\hat{N}(h)$ ha lo stesso errore standard degli X_v, cioè

$$\frac{\sqrt{\mathbb{V}[\hat{N}(h)]}}{N(h)} \leq \eta_m.$$

Dimostrazione. Abbiamo che $\mathbb{E}[\hat{N}(h)] = \mathbb{E}[\sum_{v \in V} X_{v,h}]$. Dal Teorema 3.1, $\mathbb{E}[X_{v,h}] = N(h,v) \cdot (1 + \delta_1(n) + o(1))$ da cui il primo risultato. Per la seconda affermazione, abbiamo che:

$$\frac{\sqrt{\mathbb{V}[\hat{N}(h)]}}{N(h)} \leq \frac{\sum_{v \in V} \sqrt{\mathbb{V}[X_v]}}{N(h)} \leq \frac{\eta_m \sum_{v \in V} N(h,v)}{N(h)} = \eta_m.$$

□

Per concludere, possiamo affermare che l'errore standard commesso da HyperLogLog rappresenta un limite superiore all'errore commesso da HADI per la stima di $N(h)$ e $N(h,v)$, che sono i valori fondamentali per ottenere $r_{\text{eff}}(G)$, $d(g)$ e $d_{\text{eff}}(G)$.

4 HADI SU SPARK

Nel Capitolo 2 abbiamo descritto ad alto livello che cos'è Spark, il suo funzionamento di base, le sue peculiarità e abbiamo fatto degli esempi su come utilizzarlo. Nel capitolo appena concluso è stato presentato l'algoritmo HADI per la stima del diametro di un grafo, aggiungendo anche la descrizione di cos'è un conteggio approssimato, come viene usato nel codice, e ne abbiamo fornito una versione aggiornata che migliora ulteriormente la bontà della stima. Possiediamo, quindi, tutti gli strumenti per arrivare al fulcro del progetto di tesi: portare HADI, pensato per il framework Hadoop, su Apache Spark.

Prima di buttarci a capofitto sulle implementazioni, ci chiediamo tuttavia se abbia senso procedere in questa direzione. È corretto pensare di poter eseguire una computazione *in-memory* con grafi di questo genere? Abbiamo dei reali vantaggi oppure no? Tutto il capitolo e quello successivo servirà per rispondere a questi interrogativi. Possiamo fare qui un piccolo esempio che ci può far intuire che la strada sembra corretta. Nell'articolo originale di HADI [12], gli autori presentano alcuni esperimenti su grafi di notevoli dimensioni tra cui *YahooWeb*, il grafo della rete internet ricavato dal motore di ricerca Yahoo nel 2002. Questo è formato da 1,4 miliardi di vertici e 6,6 miliardi di archi e un peso complessivo di 120 gigabyte. Lo spazio richiesto per memorizzare i contatori di quest'enorme struttura si può stimare, con 32 registri da 8 byte l'uno, a $32 \cdot (1{,}4 + 6{,}6) \cdot 10^9 \cdot 8$ byte $= 2$ terabyte, più di 16 volte il peso dell'input. Anche se tutti i supercomputer moderni non hanno problemi nel gestire terabyte di dati, possiamo immaginare che sia *ancora* troppo per non servirci della memoria secondaria. Tuttavia, con il miglioramento del metodo di conteggio approssimato, un registro di HyperLogLog ha un costo di soli 5 bit, che trasforma il calcolo in $32 \cdot (1{,}4 + 6{,}6) \cdot 10^9 \cdot 5$ bit $= 160$ gigabyte, appena 1,3 volte il peso dell'input originale. Questo miglioramento di un ordine di grandezza apre la strada alla possibilità di gestire lo spazio di memoria diversamente. I gigabyte al giorno d'oggi non sono più un tabù e anche cluster di piccole dimensioni, come quello che sarà qui utilizzato, riescono a processarli in modo efficiente.

Bisogna chiarire che la computazione *in-memory* non è la sola tecnica di elaborazione che può essere usata da Spark. Al pari di Hadoop, nell'infrastruttura di Spark i dischi rigidi sono una componente altrettanto importante e ogni nodo del cluster ne usufruisce — basti pensare che i dati di shuffle sono memorizzati proprio qui — perciò Spark può eseguire tutti i processi desiderati anche in caso di grafi troppo grandi per risiedere in RAM. Semplicemente caleranno le prestazioni a causa delle lettura/scritture in memoria secondaria. Per questo motivo nel discorso, si cerca di far riferimento sempre alla computazione *in-memory*, considerati i vantaggi che può dare al calcolo.

Un'ultima osservazione è la seguente. Fin d'ora abbiamo sempre parlato di grafi di grandi dimensioni con miliardi e miliardi di elementi. Tuttavia, si trovano grafi di interesse pratico anche ben al di sotto di questa dimensione. Per esempio, navigando tra le pagine dello *Stanford Large Network Dataset Collection* [13] osserviamo che la maggior parte dei grafi messi a disposizione per fare misure hanno vertici e archi dell'ordine

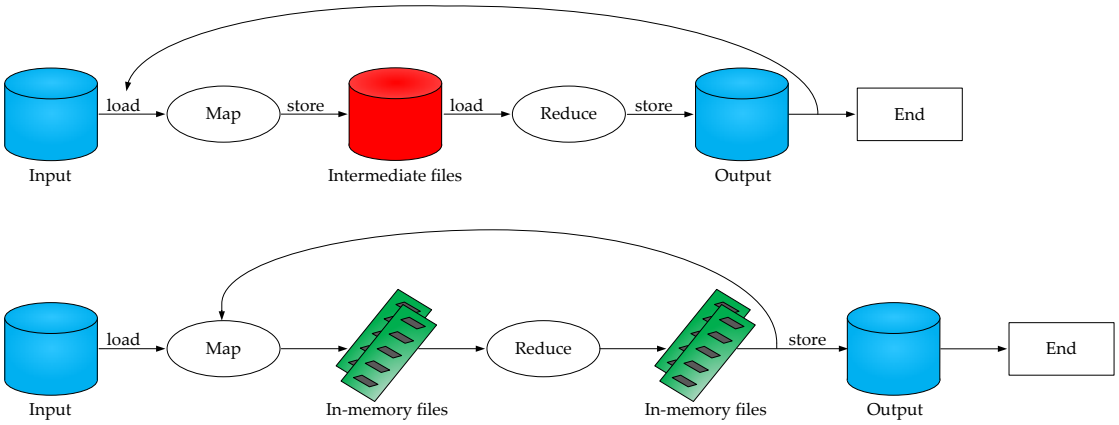


Figura 9: Confronto tra Spark e Hadoop in un ciclo MapReduce.

del milione di elementi. Certo, non è una statistica sicura, ma ci fa osservare che esiste un mondo dell'analisi su grafo che può adottare la tecnica di calcolo che vogliamo qui sviluppare.

4.1 MIGLIORAMENTI E VANTAGGI DI SPARK

La versione originale di HADI si basa interamente su una computazione MapReduce in Hadoop, per cui trasportare l'algoritmo su Spark porta tutti i benefici legati a questo tipo framework. Vediamo dunque quali sono i punti di forza di Spark nel confronto con Hadoop e quali novità esso introduce. Abbiamo già descritto molte caratteristiche nel Capitolo 2 perciò faremo una presentazione ad "alto livello", senza preoccuparci troppo dei dettagli tecnici.

4.1.1 In-memory processing

Il punto di forza di Spark è la computazione *in-memory*. Quando abbiamo un calcolo che può essere paragonato ad Hadoop (stessa memoria, numero di processori e prestazioni della rete), la velocità di esecuzione migliora di un fattore tra il 10 e il 100 [21]. In Figura 9 vediamo l'esempio di un generico processo MapReduce. La prima sequenza è di Hadoop e notiamo che vi è un accesso alla memoria secondaria prima e dopo ogni operazione sui dati. Essendo i dischi rigidi il collo di bottiglia di un normale processo di calcolo, va da sé che questo processo ha prestazioni inferiori rispetto allo stesso processo fatto con Spark. Nella seconda immagine, osserviamo che gli input/output delle operazioni rimangono in memoria primaria finché il calcolo non termina; sempre nell'ipotesi che la spazio di memoria a disposizione sia sufficiente a contenerli tutti (altrimenti serve appoggiarsi sul disco). È soprattutto con algoritmi iterativi che Spark è migliore, e HADI è uno di questi. Infatti, il ciclo di operazioni MapReduce si ripete costantemente finché l'iterazione non raggiunge il valore stimato del diametro. Possiamo quindi ipotizzare che l'uso di Spark porti degli effettivi vantaggi nelle prestazioni.

4.1.2 GraphX

La diffusione dei grafi nei sistemi informatici ha portato a un grande lavoro di analisi su di essi. Facendone un utilizzo sempre più frequente, ci si trova a dover fare ricerche, interrogazioni e misure su questa struttura dati e di trovare un modo per memorizzare efficientemente questi oggetti. Anche Spark si è occupato del problema e ha reso disponibile uno strumento, basato sul framework principale, che ottimizza la gestione dei grafi e consente di applicare ad essi funzioni e metodi in modo molto intuitivo. Si tratta del progetto GraphX. Questo *tool* per la gestione dei grafi consente di fare operazioni su di essi allo stesso modo di qualsiasi altro dataset. Spark ha introdotto l'RDD, un'astrazione comoda per memorizzare i dati e risparmiando al programmatore parecchio lavoro. GraphX estende il concetto di RDD introducendo il Resilient Distribuited Graph (RDG).

È importante accennare a com'è implementato l'RDG visto che l'argomento è collegato con HADI. In particolare, la novità di GraphX è il metodo di partizionamento che utilizza per l'RDG.

Partizionamento

Le tecniche tradizionali per partizionare un grafo si basano sull'*edge-cut*, cioè si "tagliano" gli archi del grafo al fine di memorizzare una sola volta i vertici e di replicare più volte quegli archi che vengono tagliati. In questo modo, l'overhead è direttamente proporzionale al numero di archi tagliati, e per ridurlo bisogna minimizzare questo valore, cercando allo stesso tempo di bilanciare il numero di vertici tra le partizioni. Trovare una soluzione ottima a questo problema è troppo dispendioso (e con grafi grandi lo è ancor di più), per cui la soluzione adottata è un *random edge-cut*, che risulta essere ottimo per il problema della distribuzione dei vertici, ma pessimo per la minimizzazione del numero di archi replicati. GraphX cambia tecnica e adotta un approccio *vertex-cut*, dove si "tagliano" i vertici del grafo in modo da memorizzare una sola volta gli archi, mentre i vertici devono essere replicati più volte in base al numero di tagli effettuati su di essi. Questo tipo di divisione è meno intuitivo ("tagliare" un vertice suona strano all'inizio), ma garantisce prestazioni migliori di *edge-cut*. In Figura 10 il taglio è rappresentato dalla linea tratteggiata. Gli sviluppatori di GraphX hanno creato una funzione di hash "intelligente" per la distribuzione degli archi tra le partizioni [19] e i risultati ottenuti, sia teorici che sperimentali, confermano un overhead per ogni vertice di circa $2\sqrt{M}$, dove M è il numero delle partizioni (oppure il numero delle macchine in cui distribuire gli archi).

Struttura dati

Il Resilient Distribuited Graph ha una rappresentazione centrata sul *vertex-cut* ed è formata da tre tabelle implementate con gli RDD. Seguiamo la Figura 10 per comprendere bene com'è composta la struttura.

1. EdgeTable *(pid, src, dst, data)*: l'insieme di tutti gli archi del grafo memorizzati nella convenzione (src $\rightarrow$ dst). Nel caso di grafo pesato contiene anche l'informazione *data*. Infine *pid* è l'identificatore della partizione dell'RDD contenente l'arco. La distribuzione degli archi tra le partizioni avviene mediante la funzione

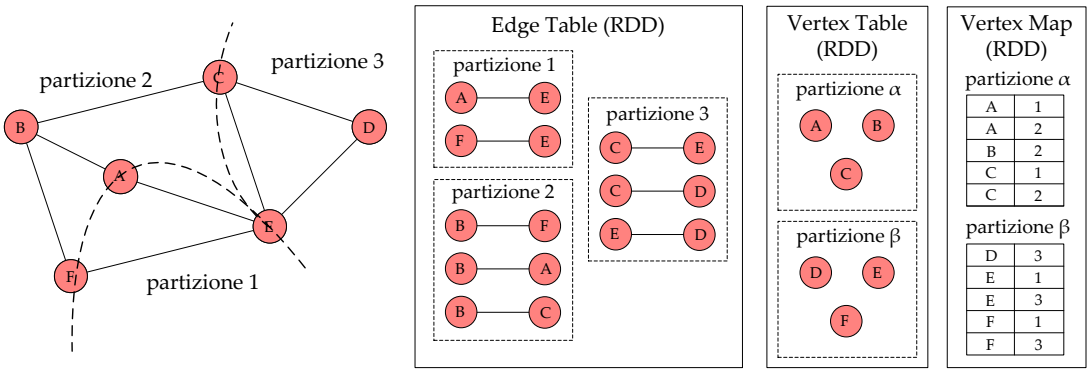


Figura 10: Esempio di *vertex-cut* con il grafo della Figura 5.

hash citata prima, creata appositamente per minimizzare la replicazione dei vertici. Perciò la linea tratteggiata in figura mostra come sono stati "tagliati" i vertici come conseguenza della distribuzione gli archi in quella particolare EdgeTable.

2. VertexTable *(id, data)*: l'insieme dei vertici del grafo, memorizzati in record *(id, data)*. Il partizionamento di questo RDD dipende dal valore *id* del vertice ed è indipendente dal tipo di partizionamento fatto in EdgeTable.

3. VertexMap *(id, pid)*: mappa che definisce, per ogni vertice, le partizioni che contengono gli archi adiacenti a esso. Questo RDD è partizionato allo stesso modo di VertexTable e permette di trovare, per un vertice u, le partizioni di EdgeTable in cui u compare.

La Figura 10 ci mostra un esempio della struttura dati che utilizza GraphX per memorizzare il grafo. Notiamo che le tabelle riferite ai vertici potrebbero essere fuse in una soltanto. Tuttavia, nelle applicazioni i dati associati ai vertici vengono modificati spesso, mentre la mappa di collegamento con gli archi rimane invariata durante tutto il processo. Perciò questa divisione, al prezzo di un leggero overhead, ci permette di ridurre il peso delle comunicazioni quando operiamo il collegamento tra EdgeTable e VertexTable.

4.1.3 Pregel API

GraphX mette a disposizione del programmatore molti metodi di base per fare filtraggi, trasformazioni sui dati e conteggi tipici di un grafo, come il calcolo del grado di un vertice, sia in ingresso che in uscita, oppure trovare le componenti connesse. Ma oltre a questo, GraphX fornisce uno strumento molto potente per implementare algoritmi iterativi su grafo e che devono operare in parallelo: Pregel [14]. Si tratta di una tecnica di *message passing* sincrona su grafo. L'idea è immaginare i collegamenti tra vertici e archi come una rete di comunicazione in cui ci si scambia dei messaggi (ricorda molto la filosofia di HADI...). Ogni vertice v esegue un programma *vProg* parallelamente a tutti gli altri vertici seguendo una sequenza di passi chiamata *super-step*:

1. All'inizio, ogni vertice invia un messaggio ai suoi vicini.

2. Il vertice v, se riceve più di un messaggio, applica una funzione *mergeMsg* per unirli e ottenere un unico messaggio *msg*. Quindi applica la funzione $vProg(v)$ che riceve in ingresso *msg*, l'informazione contenuta nel vertice, e restituisce un nuovo dato per il vertice.

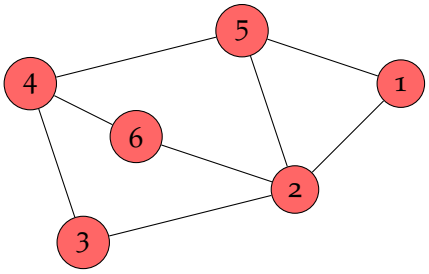
3. Il *super-step* termina se e solo se ogni vertice del grafo ha concluso il suo programma $vProg(v)$.

Terminata questa sequenza di operazioni, si ripete il tutto con un nuovo *super-step*: un vertice v invierà un messaggio al vertice u come risposta al messaggio che u ha inviato a v nel *super-step* precedente. Un'apposita funzione *sendMsg* prepara il messaggio di risposta (ritornando all'operazione 1 della lista precedente). La procedura termina quando non ci sono più messaggi da scambiare nella rete, oppure quando si è raggiunto il limite di iterazioni predeterminate.

4.2 IMPLEMENTAZIONI DI HADI SU SPARK

Le novità introdotte da Spark conducono verso tre strade possibili per implementare HADI. La prima, più semplice, è quella di usare Pregel di GraphX come astrazione di base per costruire la comunicazione tra i vertici del grafo. Com'è stato appena descritto, Pregel si basa sullo scambio di messaggi allo stesso modo in cui HADI intende la computazione sul grafo, con i vertici che "parlano" ai loro vicini. Per questo motivo, il primo codice seguirà quest'idea. La seconda implementazione utilizza GraphX come infrastruttura di gestione del grafo in ingresso. Grazie ad esso possiamo chiamare gli operatori di map e reduce sul grafo esattamente come fosse un qualsiasi RDD, lasciando a GraphX l'onere di aggiornare e modificare le tabelle che abbiamo visto in Figura 10. L'ultima strada è quella di utilizzare i metodi nativi di Spark e di creare le funzioni e gli RDD che servono per costruire HADI. Questo approccio è certo più complesso dei precedenti, anche se, operando a "basso livello", potrebbe migliorare l'efficienza del calcolo. Gli esperimenti sui grafi del capitolo successivo ci permetteranno di apprezzare le differenze tra le implementazioni e di decidere quel è scelta vincente.

Tutte tre le implementazioni hanno delle caratteristiche comuni. Innanzitutto lo scopo: utilizziamo HADI per la stima del *diametro* del grafo. Tutti i grafi che analizziamo sono grafi semplici, non orientati e connessi. L'input del programma è un file di testo in cui ogni riga è una stringa che rappresenta un arco *orientato* del grafo nel formato <u '\t' v '\n'>, dove u, v sono due numeri interi positivi (gli identificatori dei vertici u e v), '\t' è il carattere di tabulazione e '\n' il carattere di nuova riga. Perciò, avendo grafi non orientati, ogni arco deve essere rappresentato da due stringhe <u '\t' v '\n'> e <v '\t' u '\n'>. Ne consegue che un grafo con m archi è memorizzato in un file con $2m$ stringhe. Nella Figura 11 vediamo il file descrittore del grafo 5. Utilizziamo i contatori HyperLogLog per tenere traccia del numero di vertici osservati durante le iterazioni importando la libreria `it.unimi.dsi.util` e `it.unimi.dsi.fastutil` che contengono, tra le tante, la classe HyperLogLogCounterArray scritta da Boldi e Vigna dell'UniMi.



(a) *Il grafo in input.*

```
1   2
1   5
2   1
2   3
2   5
2   6
3   2
3   4
4   3
4   5
4   6
5   1
5   2
5   4
6   2
6   4
```

(b) *File di testo associato al grafo.*

Figura 11: Il grafo semplice di Figura 5 con annesso il suo file descrittore. Per ogni arco del grafo ci sono due stringhe che lo rappresentano nelle due direzioni opposte.

4.2.1 HADI-Pregel

Per mostrare l'implementazione usiamo uno pseudocodice simile al linguaggio Scala, così da tralasciare le sfumature di un vero programma, ma al tempo stesso garantire chiarezza nei vari passaggi delle istruzioni. Nella prima parte del programma (Codice 5), prepariamo tutti gli strumenti necessari al funzionamento di Pregel. Per prima cosa importiamo le librerie di Spark necessarie e creiamo un nuovo processo di calcolo distribuito con SparkContext. Poi inzializziamo il grafo, prima con Graphx.edgeListFile che riceve in ingresso un file descrittore del grafo e ritorna un valore di tipo Graph, su cui possiamo fare tutte le operazioni che desideriamo al pari di un RDD. Dopodiché, con un map associamo a ogni vertice del grafo una coppia (HyperLogLog, Boolean). Il primo è il contatore, già incrementato con l'identificatore del vertice, il secondo è un valore booleano che dice se il contatore è stato modificato dopo l'ultima iterazione. Quest'informazione ci servirà per interrompere il ciclo nel momento in cui nessun contatore viene aggiornato.

Le tre definizioni successive sono istruzioni che useremo dentro Pregel. counterUpd è una funzione che prende il contatore del vertice e il messaggio in ingresso e ne restituisce l'unione. Inoltre esegue il test per verificare se il vertice (il suo contatore) è stato modificato oppure no, nel qual caso si imposta il valore booleano a **false**. Il controllo msg.count = 0! serve per evitare di impostare **false** quando il messaggio in arrivo è il primo del ciclo. Infatti, per com'è strutturato Pregel, il primo messaggio inviato ai vertici è uguale per tutti ed è costruito appositamente. Perciò, senza questo controllo, tutti i vertici sarebbero tutti etichettati **false** e il programma terminerebbe all'iterazione zero! msgUpd esegue l'unione tra due messaggi in arrivo allo stesso vertice e initialMsg è, come appena descritto, il messaggio iniziale.

La seconda parte del codice (la 6) è il metodo Pregel applicato al nostro grafo. Esso riceve due blocchi di istruzioni in ingresso. Il primo blocco è il messaggio iniziale e il numero massimo di iterazioni. Il secondo blocco sono le funzioni *vProg*, *sendMsg* e *mergeMsg*. *vProg* è la funzione che viene eseguita da ogni vertice nel *super-step*, così definita: (VertexId, VertexData, Message) ⇒ VertexData. Nel nostro caso, per restitui-

Codice 5: HADI-Pregel. Inizializzazione

Input: grafo.txt; maxIter; numRegister
Output: $d(G)$

```
1  import {SparkContext, SparkConf, Graphx, HyperLogLogCounter}

3  object HADI_Pregel {

5     /**
6      * Il valore sc crea l'ambiente Spark in cui eseguire il calcolo,
7      * mentre gr è il grafo inizializzato nel programma attraverso la
8      * funzione edgeListFile.
9      */
10    val conf = new SparkConf().setAppName("HADI_Pregel");
11    val sc = new SparkContext(conf);
12    val gr = Graphx.edgeListFile(sc, grafo.txt);

14    /**
15     * Inizializzazione del grafo. A ogni vertice associo un valore booleano e
16     * un contatore in cui inserisco l'identificatore del vertice.
17     */
18    var graph = gr.mapVertices((id, _) => {
19       val c = new HyperLogLogCounter(numRegister);
20       c.add(id);
21       (c, true)
22    })

24    /**
25     * Funzione counterUpd. Riceve in input il contatore memorizzato
26     * nel vertice e il messaggio in arrivo a esso, calcola il nuovo
27     * contatore e lo restituisce. Il valore booleano restituito ci dice se
28     * è cambiato o meno il valore del contatore del vertice.
29     */
30    def counterUpd(counter: (HyperLogLogCounter, Boolean), msg: HyperLogLogCounter):
31    (HyperLogLogCounter, Boolean) = {
32       val res = new HyperLogLogCounter(numRegister);
33       res.max(counter._1, msg);
34       if (res == counter._1 && msg.count != 0)
35          return (res, false);
36       else
37          return (res, true);
38    }

40    /**
41     * Funzione msgUpd. Fonde i messaggi che arrivano a uno stesso vertice.
42     */
43    def msgUpd(msg1: HyperLogLogCounter, msg2: HyperLogLogCounter):
44    (HyperLogLogCounter) = {
45       val newMsg = new HyperLogLogCounter(numRegister);
46       return newMsg.max(msg1, msg2);
47    }

49    /**
50     * Il messaggio (vuoto) che verrà inviato a tutti alla prima iterazione.
51     */
52    val initialMsg = new HyperLogLogCounter(numRegister);
```

Codice 6: HADI-Pregel. Funzione Pregel applicata al grafo.

```
53      /**
54       * Pregel applicato al grafo in ingresso. Il metodo è formato da tre
55       * funzioni: vProg che restituisce il nuovo valore del vertice,
56       * sendMsg che prende in ingresso tutte le triplette (vertice,arco,vertice)
57       * e decide quali messaggi inviare a quali vertici, mergeMsg che
58       * fonde due messaggi e ne restituisce uno solo.
59       */
60      graph.pregel(initialMsg, maxIter)(
61          /** vProg */
62          (id, vCounter, msg) => counterUpd(vCounter._1, msg),
63          /** sendMsg */
64          triplet => {
65            if (triplet.vertexDstData._2)
66              Iterator(triplet.vertexDstId, triplet.vertexSrcData._1)
67            else
68              Iterator.empty
69          },
70          /** mergeMsg */
71          (a, b) => msgUpd(a, b)
72      );

74      d(G) = *ultima iterazione di Pregel*;
75  }
```

re un VertexData usiamo la funzione counterUpd descritta in precedenza. La funzione *sendMsg* è la regola per l'inoltro dei messaggi da un vertice che ha eseguito *vProg*, così definita: EdgeTriplet(VertexData, EdgeData) ⇒ Iterator(VertexId, Message). Allora, dovendo bloccare il ciclo quando tutti i contatori non ricevono aggiornamenti, inviamo il contatore al nostro vicino solo se quest'ultimo è stato aggiornato nell'ultima iterazione (**if** (triplet.dstAttr._2)). Nel caso contrario, non è necessario inviargli nulla giacché non verrà più modificato (Iterator.empty). L'ultima funzione, *mergeMsg*, semplicemente esegue l'unione di due messaggi in arrivo a uno stesso vertice, servendosi di msgUpd.

Per ottenere il diametro del grafo dobbiamo usare un piccolo trucco. Per com'è stato implementato Pregel, ad oggi (Spark 1.2.0) non si dispone di un modo per contare le iterazioni e restituirle al programmatore. Tuttavia, al momento dell'esecuzione, l'interfaccia grafica di Spark visualizza tutti i dati relativi al calcolo, compresa questa misura. Allora ricaveremo facilmente l'informazione dell'ultimo ciclo effettuato controllando, alla fine della computazione, il valore sul monitor *Web UI*. Questa caratteristica desiderata potrebbe essere implementata in futuro anche abbastanza facilmente, perché Pregel già prevede una variabile intera che conti il numero di cicli *sendMsg* → *vProg* → *sendMsg*, solo che, per il momento, non è possibile richiamarla.

4.2.2 HADI-GraphX

La comodità principale che ci fornisce Pregel è di non dover gestire la comunicazione tra i vertici. Implementando HADI con GraphX puro, invece, dobbiamo farci carico di questo problema. In particolare, emerge un fatto molto significativo per la computa-

Codice 7: HADI-GraphX. Inizializzazione

Input: grafo.txt; maxIter; numRegister
Output: $d(G)$

```
1  import {SparkContext, SparkConf, Graphx, HyperLogLogCounter}

3  object HADI_Pregel {

5     /**
6      * Il valore sc crea l'ambiente Spark in cui eseguire il calcolo,
7      * mentre gr è il grafo inizializzato nel programma attraverso la
8      * funzione edgeListFile.
9      */
10    val conf = new SparkConf().setAppName("HADI_GraphX");
11    val sc = new SparkContext(conf);
12    val gr = Graphx.edgeListFile(sc, grafo.txt);

14    /**
15     * Inizializzazione del grafo. A ogni vertice associo un intero nullo e
16     * un contatore in cui inserisco l'identificatore del vertice.
17     */
18    var graph = gr.mapVertices((id, _) => {
19       val c = new HyperLogLogCounter(numRegister);
20       c.add(id);
21       (c, 0)
22    });

24    /**
25     * Funzione counterUpd. Riceve in input due contatori e ne ritorna uno nuovo
26     * che è l'unione dei due. L'intero restituito, se maggiore di zero,
27     * dice che almeno uno dei due contatori è stato modificato.
28     */
29    def counterUpd(c1: HyperLogLogCounter, c2: HyperLogLogCounter, cntr: Int):
30     (HyperLogLogCounter, Int) = {
31       val res = new HyperLogLogCounter(numRegister);
32       res.max(c1, c2);
33       if (res == c1 && res == c2)
34          return (res, cntr);
35       else
36          return (res, cntr + 1);
37    };
```

zione. In Pregel, se u è il contatore su cui fare l'aggiornamento, e x, y, z i suoi vicini, avviene prima l'operazione $x \cup y \cup z = k$ con *mergeMsg* e poi l'operazione $u \cup k = res$ con *vProg*. Anche se l'unione dei contatori, al pari dell'unione insiemistica, gode delle proprietà commutativa e associativa — e quindi sarebbe corretto anche scrivere $x \cup y \cup z \cup u = res$ —, questo modo di operare risulta vantaggioso perché posso fare il confronto tra u e res per osservare eventuali modifiche a u.

Usando GraphX le cose cambiano. Lo scambio di comunicazione tra vertici avviene con la funzione map, mentre l'unione dei contatori con la funzione reduce. Entrambe le funzioni lavorano in modo commutativo e associativo, perciò, prendendo l'esempio appena fatto, non esiste un ordine prestabilito con cui fare $x \cup y \cup z \cup u$. Potrebbe avvenire prima $x \cup y$, poi $z \cup u$ e infine l'unione dei due risultati. In generale, una qualsiasi

Codice 8: HADI-GraphX. Ciclo principale.

```
38      /**
39       * Variabile in cui verrà memorizzato il valore dell'ultima iterazione.
40       */
41      var hMax = 0;

43      /**
44       * Inizio del ciclo dell'algoritmo HADI.
45       */
46      for h = 1 to maxIter do {

48          /**
49           * Creiamo un nuovo grafo, grTmp, identico al grafo iniziale ma con
50           * gli interi associati ai contatori azzerati.
51           */
52          val grTemp = graph.mapVertices((id, data) => (data._1, 0));

54          /**
55           * Al grafo precedente applichiamo la funzione mapReduceTriplets che
56           * invia a ogni vicino di ogni vertice il proprio contenuto informativo.
57           * Con una funzione di riduzione, tutti i contatori ricevuti da un vertice
58           * vengono uniti con la funzione counterUpd. Questo processo ritorna
59           * un VertexRDD.
60           */
61          val vRDD = grTmp.mapReduceTriplets(et => Iterator((et.dstId, et.srcAttr)),
62              (a, b) => counterUpd(a._1, b._1, a._2 + b._2)
63          );

65          /**
66           * Somma del numero delle modifiche effettuate su tutti i contatori.
67           */
68          val res = vRDD.map(a => a._2._2).reduce((a, b) => a + b);

70          /**
71           * Controllo sul valore res. Se è nullo significa che nell'iterazione
72           * corrente non è avvenuto alcun aggiornamento ai contatori. Il programma
73           * può terminare e il valore h-1 è la stima del diametro.
74           */
75          if (res == 0) {
76              hMax = h - 1;
77              break;
78          }

80          /**
81           * Nella prossima iterazione, il grafo in input deve avere i vertici
82           * che sono stati aggiornati in questo ciclo.
83           */
84          graph = [vRDD, graph.edges];
85      }

87      d(G) = hMax;
88  }
```

permutazione dell'ordine dei quattro operandi è valida, e l'ordine dipenderà solo dalla schedulazione interna di Spark. In quest'ottica, non posso distinguere il vertice su cui fare l'aggiornamento, dai suoi vicini: per il programma sono quattro HyperLogLog indipendenti su cui fare l'unione. La domanda conseguente è: come faccio a capire se u è stato modificato? Per esempio, supponiamo $u = \{1,2\}$, $x = \{1,2,3,4\}$, $y = \{2\}$ e $z = \{3\}$. Dopo la reduce abbiamo $x \cup y \cup z \cup u = \{1,2,3,4\} = res$. Controllando le modifiche avvenute ai contatori nell'ultima operazione di unione (come avviene in Pregel) allora:

- se l'ultima operazione è $u \cup r\hat{e}s = res$, con $r\hat{e}s = x \cup y \cup z$, allora quando controllo se sono avvenute modifiche ottengo $(res == u) \Rightarrow false$;
- se l'ultima operazione è, ad esempio, $x \cup r\hat{e}s = res$ con $r\hat{e}s = u \cup y \cup z$, allora quando controllo se sono avvenute modifiche ottengo $(res == x) \Rightarrow true$.

Siccome non posso scegliere di eseguire a priori il confronto $(res == u)$, c'è una possibile ambiguità nel calcolo. Per risolverla, adotteremo una strategia simile a quella di HADI-parallelo, cioè contare tutte le modifiche avvenute agli HyperLogLog durante la reduce, indipendentemente dall'ordine con cui queste si sono svolte. Questo è ragionevole perché il programma deve terminare solo quando tutti i contatori hanno raggiunto, idealmente, ogni altro contatore del grafo. A quella condizione, l'operazione $x \cup y \cup z \cup u$, essendo $x = y = z = u = \{1,2,3,4\}$, non produrrà alcun incremento al contatore delle modifiche indifferentemente dalla permutazione degli operandi.

Un'altra differenza rispetto a Pregel è che il file descrittore del grafo deve avere un *autoanello* per ogni vertice, cioè per ogni vertice v devo inserire una stringa <v '\t' v '\n'>. Questo accorgimento ci permetterà di risparmiare un'operazione di aggiornamento su tutto il grafo, migliorando l'efficienza del programma. Vedremo in seguito i dettagli.

HADI-Graphx prende in ingresso il file `grafo.txt`, il numero di registri per contatore `numRegister` e il massimo numero di iterazioni `maxIter`. La prima parte del codice, in Figura 7, è simile a quella di HADI-Pregel. La creazione dell'ambiente Spark e l'inizializzazione del grafo sono quasi identiche tranne che, associato ai vertici, c'è un valore intero invece di un booleano. Questo numero servirà durante il ciclo principale per contare le volte in cui l'HyperLogLog viene modificato e consentirà di valutare quando l'algoritmo termina. La funzione `counterUpd` presenta alcune novità rispetto a quella dell'implementazione precedente. Riceve in ingresso un intero e due HyperLogLog, calcola un nuovo contatore che sia l'unione di questi e lo restituisce. Il contatore intero è ritornato e incrementato di un'unità nel caso in cui almeno uno dei due HyperLogLog iniziali sia stato modificato. Come spiegato prima, non possiamo stabilire quale sia, dei due HyperLogLog in ingresso, quello del vertice principale e quale sia quello del suo vicino. Perciò serve considerare tutte le possibili cause di aggiornamento, anche nel caso queste siano superflue.

Passiamo alla seconda fase del programma (Codice 8). Dobbiamo implementare il ciclo principale di HADI e questa volta dobbiamo farlo "a mano". Quindi ogni iterazione è così composta:

52 Al nostro grafo in input, applichiamo un `map` e azzeriamo i valori interi associati ai vertici (alla prima iterazione, e solo in questa, sarà un'operazione superflua), mentre i contatori approssimati non vengono modificati.

61-63 Utilizziamo il metodo mapReduceTriplets per far si che ogni vertice riceva i contatori dei suoi vicini. Il metodo ha come parametri due funzioni, una di map e una di reduce. La prima è nel formato EdgeTriplet(VertexData, EdgeData) $\Rightarrow$ Iterator(VertexId, Data), e la scriviamo in modo che, per un arco u, v il vertice u invii il proprio contatore al vertice v. La funzione di reduce ha il compito di raccogliere tutti i contatori in arrivo a uno stesso vertice e di farne l'unione. Due particolari sono degni di attenzione:

- La funzione counterUpd che deve unire i due contatori dei vertici u e v, riceve in ingresso come terzo parametro la *somma* degli interi associati a u, v. Questo ci permette di trasportare l'informazione di "avvenuta modifica" per una particolare chiave k anche nel caso in cui, in una delle chiamate di counterUpd, ciò non avvenisse.
- Aver aggiunto un *autoanello* per ogni vertice, porta, alla fine della reduce, ad aver aggiornato il contatore del vertice u principale senza fare un'operazione aggiuntiva di massimizzazione tra il vertice u e la somma dei suoi vicini.

Alla fine del processo, il metodo restituisce un VertexRDD con gli HyperLogLog di ogni vertice aggiornato all'h-esima iterazione. Questo significa, che ogni contatore rappresenta il numero di vertici raggiunti in al più h passi.

68 Con un'altro processo MapReduce, otteniamo res, la somma di tutte le modifiche effettuate ai contatori.

75-78 Se e solo se res è uguale a zero, allora il programma può terminare, perché siamo nel caso in cui tutti i vertici hanno raggiunto (idealmente) tutti gli altri vertici del grafo e il valore $h - 1$ è la stima del diametro. Se ciò non accade, allora ripetiamo il ciclo e incrementiamo h.

84 Prima di ripetere il tutto, dobbiamo far si che il grafo in ingresso all'iterazione successiva abbia i vertici aggiornati allo stato raggiunto ora. Con l'istruzione graph =[vRDD, graph.edges] possiamo ottenerlo senza sforzo.

L'ultima iterazione del ciclo è la stima del diametro del grafo in input, perciò il programma termina e restituisce $d(G)$.

4.2.3 HADI-Spark

GraphX ha fornito metodi e astrazioni per facilitare la gestione interna del grafo, mentre, per quel che riguarda lo scambio di informazione tra un vertice e i suoi vicini, è servito un approccio più meticoloso per trattare senza errore i contatori e gli aggiornamenti. Arriviamo ora all'ultima idea implementativa. Ci stacchiamo del tutto da GraphX al fine di utilizzare i metodi nativi di Spark per la manipolazione degli RDD. Questa soluzione si è rivelata leggermente più complessa dal punto di vista della scrittura del codice, ma ha il vantaggio di essere più efficiente delle precedenti, come vedremo nel capitolo successivo.

Nella prima parte del programma (Codice 9) abbiamo l'inizializzazione delle strutture che descrivono il grafo. Questa volta siamo noi a doverle costruire a partire dal file descrittore in input (anche qui, un *autoanello* per ogni vertice). Seguendo l'approccio

Codice 9: HADI-Spark. Inizializzazione

Input: grafo.txt; maxIter; numRegister
Output: $d(G)$

```
1  import {SparkContext, SparkConf, HyperLogLogCounter}

3  object HADI_Spark {

5     /**
6      * Il valore sc crea l'ambiente Spark in cui eseguire il calcolo.
7      */
8     val conf = new SparkConf().setAppName("HADI_Spark");
9     val sc = new SparkContext(conf);

11    /**
12     * Inizializzazione dell'edgeRDD. Creo un RDD di stringhe con textFile e
13     * poi spezzo ogni stringa in due con un map. Infine trasformo, con un map,
14     * gli elementi dell'RDD in oggetti chiave/valore (Int, Int) che descrivono
15     * gli archi del grafo nella direzione a->b.
16     */
17    val edgeTmp1 = sc.textFile(grafo.txt);
18    val edgeTmp2 = edgeTmp1.map(a => a.split('t'));
19    val edgeRDD = edgeTmp2.map(a => (parseInt(a._1), parseInt(a._2)));

21    /**
22     * Inizializzazione del vertexRDD. Prima ricavo tutti gli identificatori
23     * distinti dei vertici partendo dall'RDD degli archi. Poi a ogni
24     * vertice associo un intero nullo e un contatore in cui inserisco
25     * l'identificatore del vertice. Otteniamo un RDD del tipo (Int, (HLL, Int)).
26     */
27    val vertexTmp = distinct(edgeRDD.keys);
28    var vertexRDD = vertexTmp.map(id => {
29      val c = new HyperLogLogCounter(numRegister);
30      c.add(id);
31      (id, (c, 0))
32    });

34    /**
35     * Funzione counterUpd. Riceve in input due contatori e ne ritorna uno nuovo
36     * che è l'unione dei due. L'intero restituito, se maggiore di zero,
37     * dice che almeno uno dei due contatori è stato modificato.
38     */
39    def counterUpd(c1: HyperLogLogCounter, c2: HyperLogLogCounter, cntr: Int):
40     (HyperLogLogCounter, Int) = {
41       val res = new HyperLogLogCounter(numRegister);
42       res.max(c1, c2);
43       if (res == c1 && res == c2)
44          return (res, cntr);
45       else
46          return (res, cntr + 1);
47    };
```

utilizzato da GraphX, descriviamo il grafo mediante due RDD, uno di vertici e uno di archi. Si tratta di una soluzione vantaggiosa poiché gli archi non sono mai modificati durante i calcoli di HADI e la struttura, una volta memorizzata, rimane a disposizione

Codice 10: HADI-Spark. Ciclo principale.

```
48      /**
49       * Inizio del ciclo dell'algoritmo HADI.
50       */
51      for h = 1 to maxIter do {

53          /**
54           * Join tra gli archi del grafo e i vertici. Otteniamo un nuovo RDD
55           * del tipo (Int, (Int, (HLL, Int))). Esempio: join tra un arco (a, b) e
56           * il vertice (a, (HLL(a), 0)). Risultato (a, (b, (HLL(a), 0))).
57           */
58          val tmp1 = edgeRDD.join(vertexRDD);

60          /**
61           * Trasformiamo il risultato precedente in un RDD (Int1, (HLL, Int2)) in
62           * cui ogni HLL è il contatore di un vicino del vertice etichettato con
63           * Int1. Esempio: (a, (b, (HLL(a), 0))) ⇒ (b, (HLL(a), 0)). Il valore
64           * nullo è tale perché azzeriamo il contatore delle modifiche.
65           */
66          val tmp2 = tmp1.map(a => (a._2._1, (a._2._2._1, 0)));

68          /**
69           * Aggiornamento dei contatori. Se u -> v è un arco del grafo, allora
70           * l'RDD tmp2 contiene la coppia (v, (HLL(u), 0)). Con una
71           * reduceByKey otteniamo l'unione di tutti i contatori che sono i
72           * vicini di un dato vertice. Il risultato è un nuovo vertexRDD.
73           */
74          val newVertexRDD = tmp2.reduceByKey(
75              (a, b) => counterUpd(a._1, b._1, a._2 + b._2));

77          /**
78           * Somma di tutte le modifiche fatte ai contatori.
79           */
80          val change = newVertexRDD.map(b => b._2._2).reduce((a, b) => a + b);

82          /**
83           * Controllo sul valore change. Se è nullo significa che nell'iterazione
84           * corrente non è avvenuto alcun aggiornamento ai contatori. Il programma
85           * può terminare e il valore h-1 è la stima del diametro.
86           */
87          if (change == 0) {
88              d(G) = h − 1;
89              break;
90          }

92          /**
93           * Associamo l'RDD appena calcolato alla variabile vertexRDD.
94           */
95          vertexRDD = newVertexRDD;
96      }

98      return d(G);
99  }
```

del programma. Un altro approccio potrebbe essere creare un unico RDD con gli archi e associare ad ogni chiave un HyperLogLog. Questo vorrebbe dire modificare l'RDD a ogni ciclo, ma risparmiando lo spazio dell'RDD dei vertici. Qui adottiamo la prima soluzione.

All'inizio del programma, come sempre, prepariamo l'ambiente di calcolo di Spark creando lo SparkContext. Poi creiamo l'RDD degli archi. [17-19] Partendo dal file in input, otteniamo un RDD di stringhe, una per ogni arco e poi, con un doppio map, trasformiamo ogni stringa in una coppia chiave/valore con entrambi gli elementi di tipo Int. Per avere gli identificatori dei vertici sfruttiamo il risultato appena ottenuto. [27-32] Raccogliamo tutte le chiavi diverse presenti in edgeRDD. Esse sono una per ogni vertice perché, nel file descrittore del grafo, abbiamo gli archi in entrambe le direzioni, oltre all'*autoanello*, per cui esistono almeno due chiavi che identificano uno stesso vertice all'interno di edgeRDD. Il map successivo crea l'RDD dei vertici, nel formato (Int, (HyperLogLog, Int)) in cui il primo elemento è l'identificatore del vertice, mentre il secondo è il contatore del numero di vicini del vertice e il contatore delle modifiche avvenute. [39-47] La funzione counterUpd è del tutto identica a quella di HADI-GraphX ed esegue l'unione di due HyperLogLog e incrementa il contatore delle modifiche se lo ritiene opportuno. Nel secondo listato (Codice 10) vediamo il ciclo principale di HADI:

58 Eseguiamo un *join* tra edgeRDD e vertexRDD. Otteniamo un nuovo RDD dove ogni elemento è nel formato (Int, (Int, (HLL, Int))). Per esempio, se abbiamo l'arco $(5,7)$ e il vertice $(5,(HLL(5),11))$ otteniamo $(5,(7,(HLL(5),11)))$.

66 Con un map trasformiamo il risultato precedente in un RDD con elementi (Int, (HyperLogLog, 0)), quindi l'esempio precedente diventa

$$(5,(7,(HLL(5),11))) \Rightarrow (7,(HLL(5),0)).$$

Per capire, se abbiamo gli archi $(5,7),(6,7),(12,7)$ (che significa 5 –> 7, 6 –> 7, 12 –> 7) otteniamo, dopo le due trasformazioni, gli oggetti

$$(7,(HLL(5),0)),(7,(HLL(6),0)),(7,(HLL(12),0)).$$

È come avere 7 <– HLL(5), 7 <– HLL(6), 7 <– HLL(12). Questo passo è fondamentale perché vediamo che si associa ad ogni vertice tutti i contatori dei suoi vicini. In questo modo possiamo fare la reduce in base alla chiave dell'elemento di questo RDD. Notiamo che il valore intero che sta assieme all'HyperLogLog è impostato a zero perché, come in HADI-GraphX, azzeriamo il numero di modifiche effettuate ai contatori.

74-75 Con una reduceByKey eseguiamo l'unione di tutti i contatori in "arrivo" a un vertice. Per fare questo ci serviamo di counterUpd. Alla fine dell'operazione, otteniamo un nuovo RDD di vertici con gli HyperLogLog aggiornati e con il contatore cntr che rappresenta il numero di modifiche fatte in quel gruppo di reduce.

80 Sommiamo tutti i cntr per ottenere il totale numero di modifiche fatte ai contatori.

87-90 Se non sono avvenute modifiche, arrestiamo il ciclo. Il numero dell'iterazione precedente è la stima del diametro del grafo. Altrimenti continuiamo con le istruzioni successive.

95 Alla prossima iterazione, vertexRDD in input deve essere l'RDD di vertici che abbiamo modificato fino a questo punto. Una semplice assegnazione garantisce questo vincolo.

Ripetiamo questo ciclo di istruzioni finché il break lo interromperà. A quel punto otteniamo la stima del diametro del grafo come numero di iterazioni effettuate meno uno.

Analisi Round-Work-Communication

Delle tre implementazioni, HADI-Spark è quella descritta a più basso livello. Per giungere al risultato serve procedere un passo per volta e mostrare tutte le trasformazioni che avvengono sui dati. È probabile che anche Pregel e GraphX, al loro interno, eseguano operazioni simili a quelle viste qui e con un costo computazionale paragonabile, se non addirittura inferiore grazie alle ottimizzazioni. Per questo motivo procediamo con l'analisi delle prestazioni di quest'ultimo codice (9 e 10), che ci darà un buon riferimento sulla complessità dell'algoritmo HADI portato su Spark. Partendo da quest'analisi potremo poi stabilire, con gli esperimenti, se GraphX e Pregel siano, in pratica, migliori o peggiori di HADI-Spark.

Per valutare la complessità prendiamo spunto dal JáJá [11] e del suo *Work-Time Presentation Framework of Parallel Algorithms*, adattandolo alle nostre esigenze. Le misure che utilizziamo sono:

- *Round*. Il numero di volte in cui ripeto il ciclo principale dell'algoritmo.
- *Work*. Il numero totale di operazioni elementari eseguite, aggregato su tutti i round e su tutte le macchine.
- *Communication*. Il volume complessivo dei messaggi scambiati, aggregato su tutti i round.

Consideriamo il caso di un grafo $G = (V, A),\ |V| = n,\ |A| = m,\ m > n$ con diametro $d(G) = d$ e k il numero di registri di cui è formato ogni contatore HyperLogLog.

Lemma 4.2.1. *Il numero di round di HADI-Spark è*

$$R \in O(d).$$

Dimostrazione. Un round per la creazione di *edgeRDD* e uno per *vertexRDD*. Dopodiché, in ogni iterazione del ciclo principale otteniamo, con le operazioni map e reduce, un nuovo *vertexRDD* e il conteggio delle modifiche agli HyperLogLog. L'ultima iterazione è quella successiva al raggiungimento del diametro del grafo. In totale

$$1 + 1 + (d + 1) = 3 + d = O(d).$$

□

Lemma 4.2.2. *Il numero complessivo di operazioni effettuate in HADI-Spark sono dell'ordine di*

$$W \in O\Big(d * (m + n)\Big)$$

Dimostrazione. Servono m operazioni per creare *edgeRDD*, una per ogni arco del grafo in ingresso. L'operazione distinct (edgeRDD.key) su valori interi costa al più m, utilizzando un approccio *Counting Sort* [4], mentre il map sull'RDD dei vertici richiede n operazioni. Arriviamo al ciclo principale, in cui eseguiamo d volte:

- Il *join* tra *edgeRDD* e *vertexRDD* con un costo di $m + n$. Questo costo è giustificato dal fatto che Spark utilizza l'algoritmo *bucket sort* [4] per ridistribuire, tra le macchine del cluster, i vertici e gli archi. Quest'algoritmo è lineare nel caso in cui i numeri da ordinare siano distribuiti equamente in un intervallo fissato. Utilizzando identificatori di vertici nell'intervallo $[0, 1, \ldots, n-1]$, siamo nel caso corretto;
- m operazioni per il map;
- m per la reduceByKey;
- $n + m$ per ottenere il valore change del numero di modifiche fatte ai contatori;
- infine un'operazione controllo su change.

Allora, con un buon grado di precisione, possiamo stimare il numero totale di operazioni come

$$2m + n + d\Big(3m + n + 1 + (m + n)\Big)$$

che dal punto di vista asintotico, tralasciando i fattori costanti e gli ordini di grandezza inferiori, significa

$$2m + n + d\big(4m + 2n)\big) = O\Big(d * (m + n)\Big).$$

□

Lemma 4.2.3. *Il volume totale di messaggi scambiati in HADI-Spark è dell'ordine di*

$$C \in O\Big(d * (m \cdot K \log\log n)\Big)$$

Dimostrazione. In questa analisi, consideriamo il "caso peggiore" in cui ogni operazione che richiede una forma di comunicazione scambi il massimo numero di messaggi. In pratica, con le ottimizzazioni di Spark nel gestire gli RDD, questo fattore sarà drasticamente ridotto.

La funzione distinct alla riga 27 richiede al più m scambi di valori interi. Poi, a ogni iterazione del ciclo principale, il *join* richiede al più m scambi di HyperLogLog e la *reduceByKey* altri m. Perciò il costo maggiore è $d * m$ messaggi di contatori HyperLogLog. Se ognuno di questi è formato da K registri, e ogni registro pesa $\log\log n$ bit, possiamo ipotizzare che il volume totale dei messaggi scambiati è dell'ordine di

$$O\Big(d * (m \cdot K \log\log n)\Big).$$

□

5 ESPERIMENTI SUI GRAFI

Ricapitoliamo il lavoro fin qui svolto. Abbiamo cominciato con una descrizione di Apache Spark come nuovo framework per elaborare grandi quantità di informazioni su cluster. Poi è stato approfondito il tema dei *grafi*, fino a presentare l'algoritmo HADI per la stima del diametro e una sua versione per il calcolo parallelo su Hadoop. Infine abbiamo idealmente "unito" i due argomenti con l'implementazione del suddetto algoritmo in ambiente Spark.

Perciò, in questo capitolo ci occuperemo di eseguire degli esperimenti sui grafi al fine di valutare le prestazioni, la correttezza e la scalabilità di questa nuova versione di HADI. Le prove effettuate e i relativi risultati saranno presentati nell'ottica di un percorso, in cui ogni test verrà modellato sulla base dell'esito, positivo o negativo, dell'esperimento precedente.

Alla fine del percorso potremo, a ragione, decidere qual è la migliore infrastruttura algoritmica per risolvere il nostro problema computazionale.

5.1 AMBIENTE DI LAVORO

Gli esperimenti sono stati effettuati su un cluster di 16 calcolatori presente nel Dipartimento di Ingegneria dell'Informazione di Padova. Ogni macchina possiede una tecnologia di tipo *consumer* del 2009 con le seguenti caratteristiche:

- CPU: Intel Nehalem i7-950, Quad core, Processor Base Frequency = 3,07 GHz, Max Memory Bandwidth = 25,6 GB/s;
- RAM: 6 × Kingston 2 GB, 1600 MHz tri-channel;
- Motherboard: Asus P6T SE;
- Hard disk: 6 × Samsung HD103SJ, 7,200 RPM, 1TB, Max Data Transfer Rate = 300 MB/s;
- Network interface controller: Myricom High Speed 10G-PCIE-8B-S, Standard throughput = ~9,9 Gb/s.

Dai valori elencati possiamo notare un particolare legato alle comunicazioni tra i nodi. Anche se il collo di bottiglia tipico di un sistema di elaborazione è rappresentato dalla rete di trasmissione dati, nel nostro cluster sono i dischi rigidi a rappresentare il vincolo temporale più elevato, con un tempo di lettura/scrittura di 300 MB/s (nominale) molto più lenta rispetto alla nostra Ethernet con un throughput di quasi 10 Gb/s. In ogni caso, quando parleremo di "tempo di comunicazione" ci riferiremo al tempo per trasportare un qualsiasi dato da un punto all'altro del cluster, formato dal tempo trasporto nella rete e quello necessario per la scrittura su disco, senza distinguere quali delle due parti è quella più critica.

Figura 12: Il cluster a 16 macchine con cui sono stati svolti gli esperimenti.

In ogni macchina è installato Spark 1.2.0 su un sistema operativo Debian con kernel Linux 3.16.7. In generale il cluster, riprendendo il modello master/slave di Figura 2, è formato da 16 *worker* distribuiti uno per macchina, e da un *driver* che risiede in uno dei computer. Perciò una delle macchine è condivisa da entrambi i processi. Per eseguire i nostri test, abbiamo modificato alcune impostazioni del cluster nel seguente modo:

- *spark.driver.memory = 4g*. Dovendo condividere le risorse con un processo *worker* limitiamo la memoria a 4GB.
- *spark.executor.memory = 9g*. Memoria associata a ogni *worker*.
- *spark.default.parallelism* = 256. Questa impostazione regola il numero di task in cui è divisa una qualsiasi operazione fatta sui dati. Per migliorare le prestazioni, i progettisti di spark hanno consigliato di impostare il valore a circa il doppio del numero di core del sistema. Nel nostro cluster abbiamo 16 processori da 4 core l'uno, che con la virtualizzazione diventano 8, da cui arriviamo a 256.
- spark.shuffle.blockTransferService = nio.
- spark.shuffle.manager = hash.
- spark.shuffle.consolidateFiles = true.
- spark.io.compression.codec = lz4.

Queste impostazioni sono state ottenute dopo una fase iniziale di test, in cui cercavamo le prestazioni migliori sia come velocità di calcolo sia come minimizzazione di errori di esecuzione (*file_not_ found*, *stack_overflow*, *failed_to_uncompress*).

Tabella 8: Elenco dei grafi utilizzati negli esperimenti.

Grafo	vertici	archi	taglia (MB)	$d(G)$	$d_{\text{eff}}(G)$
dblp	317 080	2 416 812	31	21	8
orkut	3 072 441	237 442 607	3400	9	4,8
road-CA	1 957 027	7 477 803	107	849	500

5.1.1 Grafi test

Nella Tabella 8 osserviamo i grafi che abbiamo utilizzato per i nostri esperimenti. Questi, distribuiti da Stanford [13] per il progetto SNAP, hanno caratteristiche diverse tra loro che ci consentono di valutare le prestazione dei programmi al variare delle misure più significative di un grafo. Il primo grafo è costruito sulla base di DBLP, un database di articoli scientifici nell'ambito della *computer science*. I vertici sono gli autori degli articoli, mentre un arco tra due vertici significa che gli autori associati a quei vertici hanno collaborato in almeno una pubblicazione. Questo è un grafo di "piccole" dimensioni, sia in termini di spazio che in termini di diametro. Sarà utile per fare le prime stime con relativa facilità, oltre a servire da "cavia" per scovare gli errori di programmazione. Il grafo successivo rappresenta le relazioni di amicizia tra gli utenti del *social network* Orkut, creato da Google e dismesso nel 2014. Si tratta di un grafo molto pesante, se paragonato a dblp, e con un diametro piccolo. Infine, roadNet-CA rappresenta la rete stradale della California, con un vertice per ogni intersezione o punto di arrivo (città, incroci stradali) connessi tra loro con gli archi. Nonostante le dimensioni non eccessive (~100 MB) questo grafo è particolarmente ostico da gestire per via del diametro molto elevato. Come abbiamo sempre ipotizzato, tutti i grafi sono semplici, non orientati e connessi.

Le misure del diametro fornite da Stanford sono state anch'esse ricavate con un metodo approssimato. Per ogni grafo è stata eseguita una *DFS* su 1000 vertici scelti casualmente e, tra tutte le profondità massime ricavate, si è preso quella più grande come diametro del grafo; motivo per cui i valori in tabella sono un limite inferiore alla stima del diametro. Non è quindi impossibile che, usando un metodo di conteggio accurato, si possa ricavare un limite ancora maggiore.

5.2 CONFRONTO TRA LE IMPLEMENTAZIONI DI HADI

Nel Capitolo 4 abbiamo presentato tre idee per portare HADI su Spark. La prima utilizza l'astrazione di calcolo Pregel per creare la comunicazione tra i vertici del grafo. La seconda si serve della libreria GraphX di Spark per gestire il grafo; la terza si basa solo sui metodi nativi di Spark per la gestione e l'elaborazione degli RDD.

La domanda ovvia è: qual è il migliore? La risposta ha molteplici aspetti. Innanzitutto, lo scopo dell'algoritmo è stimare il diametro del grafo, perciò ci chiediamo se la stima è accurata per tutte le implementazioni. Prima di vedere i risultati possiamo ipotizzare che, al pari del numero di contatori, la stima del diametro sarà uguale per

Tabella 9: Risultati del confronto tra le implementazioni.

Algoritmo	Grafo	$\log_2$ reg	Diametro	Tempo (s)	Iterazione (s)	$\frac{\text{Dati utilizzati}}{\text{Peso grafo}}$
HADI-Pregel	dblp	10	[19, 23]	333	16	263
HADI-GraphX	dblp	10	[19, 22]	354	17	50
HADI-Spark	dblp	10	[19, 22]	130	6,3	25
HADI-Pregel	orkut	5	n.r.	+8h	n.r.	15
HADI-GraphX	orkut	8	[8, 9]	1510	185	3,4
HADI-Spark	orkut	8	8	1435	180	3
HADI-Pregel	road-CA	5	n.r.	n.r.	n.r.	34
HADI-GraphX	road-CA	6	n.r.	$\approx$ 56 000	68	12
HADI-Spark	road-CA	6	[817, 849]	10 579	12	10

tutti e tre i programmi. Infatti l'algoritmo è identico, i contatori sono gli HyperLogLog con lo stesso errore standard perciò, a parte le fluttuazioni statistiche, il valore del diametro ricavato dovrebbe essere lo stesso. Le altre risposte alla domanda sono il tempo di esecuzione e lo spazio necessario alla computazione, inteso come peso degli RDD dei vertici e degli archi utilizzati dal sistema per la computazione.

5.2.1 Esperimenti

L'ambiente di lavoro in cui sono stati eseguiti i test è quello descritto nella Sezione 5.1 con una piccola differenza. Per ovviare alla necessità di memoria del processo *driver* è stato assegnato a questo un intero calcolatore, perciò la computazione è stata svolta da 15 macchine (un *worker* ognuno). Non sono state modificate altre impostazioni.

Sono stati elaborati, per ogni implementazione, tutti e tre i grafi test. L'unica differenza da ricordare è che HADI-GraphX e HADI-Spark necessitano di un autoanello per ogni vertice del grafo, fatto che aumenta leggermente il peso del file in input rispetto a quello usato da HADI-Pregel. Tuttavia, aggiungere n archi a un grafo con $m \gg n$ non comporta grosse differenze, soprattutto se ci permette di migliorare le prestazioni del calcolo, come nel nostro caso.

Dei diversi test fatti, alcuni sono serviti per verificare la presenza di errori di programmazione e di corretta impostazione del cluster. Dopodiché, per ogni grafo e per ogni implementazione abbiamo preso in considerazione sei esperimenti e abbiamo calcolato la media aritmetica delle stime per elaborare i risultati. I valori dell'errore standard dei contatori HyperLogLog si trovano nella Tabella 7 del Capitolo 3. Quand'è stato possibile, per ogni esperimento abbiamo usato lo stesso numero di registri per tutte e tre le implementazioni.

5.2.2 Risultati

Nella Tabella 9 possiamo osservare le misure ricavate dagli esperimenti. Nella colonna "Diametro" è stato inserito l'intervallo dei valori ricavati, in "$\frac{\text{Dati utilizzati}}{\text{Peso grafo}}$" vediamo il rapporto tra lo spazio di dati creati dall'algoritmo per eseguire il calcolo e il peso del

file in input del grafo, infine "n.r." significa che non si è ottenuto un risultato.

dblp

I test su *dblp* hanno prodotto una buona stima del diametro se paragonata con la descrizione di Stanford, ottenendo in parecchi casi un valore migliore, raggiungendo 22 e in un caso anche 23. Tutte le implementazioni hanno risultati praticamente identici, che confermano l'ipotesi fatta prima. Rispetto alle prestazioni in tempo e spazio, HADI-Spark è l'algoritmo più performante dal punto di vista temporale, impiegando quasi un terzo del tempo di calcolo di HADI-GraphX e HADI-Pregel. Anche per lo spazio utilizzato la classifica è simile, con Spark che richiede circa 25 volte lo spazio dell'input, il doppio per GraphX e dieci volte tanto per Pregel. Tutte computazioni sono state eseguite con un contatore HyperLogLog a 1024 registri e un errore standard del 3,25 per cento.

Possiamo affermare che, per grafi di piccole dimensioni e diametro contenuto, l'implementazione di HADI-Spark è la migliore.

orkut

Qui cominciamo a osservare le prime importanti differenze. Il grafo ha un peso più corposo (3,4 GB) se paragonato a quello precedente, e un diametro piccolo. Spark e GraphX hanno ottenuto risultati abbastanza simili, con un leggero "vantaggio" di HADI-Spark sulle misure di tempo e spazio. Esce invece sconfitto Pregel, che dopo otto ore di lavoro non è riuscito a restituire un risultato valido. Complice ciò è anche il numero di dati da esso creato per eseguire la computazione: ben 15 volte l'input; un'enormità rispetto alle tre di Spark e GraphX, e un risultato ancor più deludente dato che il numero di registri per contatore usati era "solo" 32, ben otto volte meno delle altre due implementazioni.

Questo test allontana Pregel dall'essere una scelta algoritmicamente valida, mentre ottengono una buona valutazione sia Spark puro che GraphX.

road-CA

Il grafo più "complicato" su cui fare i test è stato senza dubbio questo. Anche se di soli 107 MB, il diametro elevato ha reso la computazione particolarmente ostica per tutte le implementazioni a causa dei ripetuti cicli di HADI, oltre ad aver stressato il cluster molto a lungo. Parecchi giorni sono stati dedicati alla manutenzione e impostazione del cluster per poter lavorare su questo grafo in scioltezza. La compressione *lz4* e l'impostazione che consolida i file di shuffle sono alcuni dei risultati di questo processo. Questi test hanno mostrato che HADI-Spark ha un comportamento buono anche su grafi con diametro elevato (come questo caso) e ha riportato una stima del diametro molto vicina a Stanford, fino a 849, davvero niente male se pensiamo che gli HyperLogLog utilizzati avevano 64 registri e un errore standard di quasi il 13 per cento. Pregel, come in precedenza, non ha raggiunto alcun risultato, mente GraphX "avrebbe" raggiunto una stima del diametro dopo circa 56 000 secondi, ma un errore sistematico interno a Spark bloccava la computazione dopo qualche ora di calcolo.

Dai risultati ottenuti possiamo dire che:

- HADI-Pregel ha buone prestazioni solo per grafi di piccole dimensioni e per un valore del diametro contenuto; qualche decina di unità al massimo. Oltre questo limite diventa inutilizzabile. Il limite evidente di Pregel è la grande quantità di informazioni create per il calcolo. In Tabella 9 possiamo vedere che, per qualsiasi tipo di grafo, i dati utilizzati sono più di un ordine di grandezza maggiori, rispetto al peso del file in ingresso. Se per *dblp* la cosa è ancora tollerabile, con grafi più grandi non lo è più. Quando abbiamo ottenuto risultati di tipo "n.r.", Pregel ha continuato a eseguire HADI per più di 10 ore senza arrivare a una qualche misura del diametro, perciò abbiamo deciso di fermarlo.

- HADI-GraphX lavora bene con grafi pesanti, dell'ordine del gigabyte, però soffre i diametri elevati. Inoltre, capitano spesso errori inerenti alla sua implementazione in Spark, su cui poco si può fare per risolverli (una nuova versione di Spark potrebbe forse correggerli). In particolare, le continue iterazioni di HADI in grafi come *road-CA* portano a errori sulla scrittura dei file di shuffle e della loro memorizzazione su disco. Altro problema è l'aumento di tempo di calcolo con l'aumento del numero di iterazioni. Idealmente, ogni ciclo principale di HADI esegue le stesse operazioni sullo stesso numero di dati, perciò il tempo che serve per ogni iterazione è idealmente costante. Nella pratica questo dovrebbe essere vero a meno di costi dovuti alla distribuzione dei dati in Spark e al tempo di comunicazione tra le macchine. In HADI-GraphX non è così, e tra le prime iterazioni e le ultime c'è un aumento del tempo di calcolo fino a un fattore 10.

- HADI-Spark ha buone prestazioni sotto ogni punto di vista. L'algoritmo scala molto bene al variare della taglia dell'input e si sono verificati pochissimi errori, in gran parte corretti dopo le prime prove. In Figura 13 osserviamo un esempio di computazione su *road-CA* con i tempi impiegati dall'algoritmo per eseguire ogni ciclo. Con l'avanzare del numero di iterazioni l'aumento del tempo di calcolo è contenuto, e ciò conferma la bontà della nostra implementazione. Infatti dal punto di vista teorico ogni ciclo dovrebbe durare lo stesso lasso di tempo, perché sono coinvolti sempre gli stessi dati e si eseguono le stesse operazioni. Allora, considerando che

  - gli shuffle sono le operazioni più costose nel calcolo;
  - il peso degli shuffle aumenta con l'aumento del peso dei contatori, perché sono i dati scambiati tra i vertici del grafo;
  - i contatori aumentano di peso quando, con l'avanzare delle iterazioni, i registri hanno i valori $0, 1$ sempre più sparsi, limitando la possibilità di compressione dati.

  Perciò è ragionevole osservare un leggero aumento del tempo di esecuzione. Dal grafico si evidenzia un incremento di un fattore 2 alla fine del calcolo.

Per tutti questi motivi, l'implementazione HADI-Spark è quella migliore. In tutti gli esperimenti successivi ci serviremo di quest'ultima come programma di base per calcolare il diametro e fare ulteriori prove.

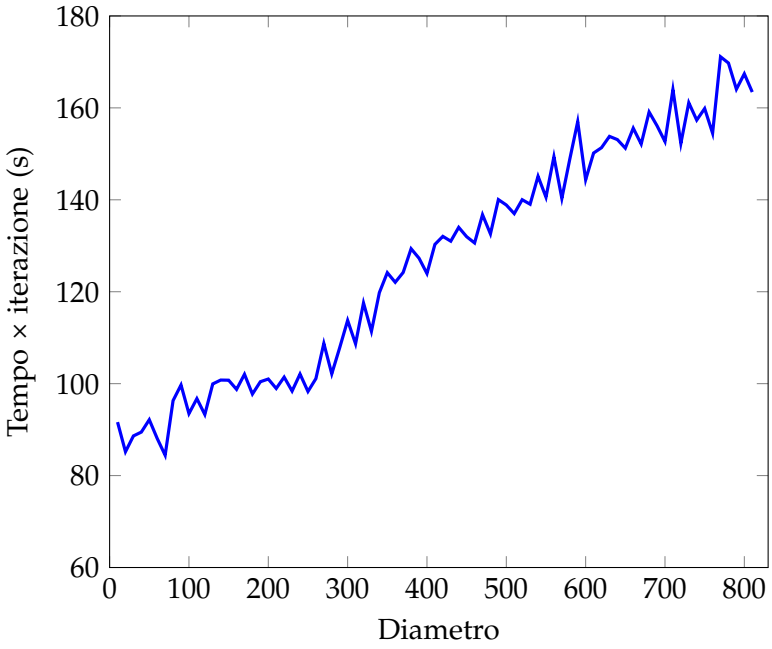


Figura 13: HADI-Spark nel calcolo di *road-CA*. Tempo impiegato per eseguire un'iterazione del ciclo di HADI al variare del diametro raggiunto.

Tabella 10: Confronto tra HADI-Spark e HADI originale implementato in Hadoop.

	$d(G)$		Tempo (s)	
	HADI-Spark	HADI	HADI-Spark	HADI
dblp	18	18	100	1140
orkut	8	8	881	4380
road-CA	823	808	11 770	50 400

5.2.3 Confronto con HADI originale

Se i teoremi sulla complessità di HADI del Capitolo 3 e 4 già suggerivano che l'implementazione in Spark era più performante, gli esperimenti hanno definitivamente confermato questo fatto. Nella Tabella 10 osserviamo le stime del diametro e i tempi di esecuzione tra HADI originale e la nostra implementazione HADI-Spark, ottenuti usando come input i nostri tre grafi. Il numero di contatori è impostato per entrambi a 32. Vediamo che l'algoritmo originale eseguito su Hadoop ha un tempo di esecuzione che è quasi un ordine di grandezza superiore al tempo di esecuzione di HADI-Spark.

5.3 PROBABILISTICCOUNTER VS HYPERLOGLOG

Dopo aver discusso a lungo sulle caratteristiche dei contatori HyperLogLog, sottolineandone l'efficienza e l'utilità, verifichiamo dal punto di vista sperimentale se le indicazioni fornite dalla teoria sono corrette. Allora, scelto HADI-Spark come algoritmo per il calcolo, creiamo degli esperimenti per valutare le differenze di prestazioni

tra l'uso di ProbabilisticCounter, i contatori usati in HADI originale, e i nuovi HyperLogLog. Quello che ci aspettiamo è che HADI-Spark abbia prestazioni migliori per il fatto che lo spazio richiesto per memorizzare i contatori è "significativamente" inferiore rispetto ai ProbabilisticCounter. Infatti, i primi hanno registri dell'ordine di $O(\log \log n)$, mentre gli altri $O(\log n)$ (n numero di vertici del grafo), quindi un abbassamento esponenziale dello spazio richiesto.

5.3.1 Esperimenti

Ci focalizziamo su due esperimenti. Il primo è fatto sul grafo *orkut*. Lanciamo HADI-Spark per calcolare il suo diametro e confrontiamo il comportamento dell'algoritmo quando usa HyperLogLog e quando usa ProbabilisticCounter, a parità di errore standard nella misura. Ripetiamo più volte l'esperimento variando il numero di registri per contatore, e quindi il valore dell'errore standard. Ci aspettiamo che l'accuratezza della stima sia la stessa, visto che l'errore statistico dei contatori è impostato ogni volta identico per entrambi; mentre dovrebbe cambiare, e di molto, il tempo impiegato per il calcolo. Il secondo esperimento serve per valutare l'efficienza della stima a parità di spazio occupato dai contatori. In quest'ottica, gli HyperLogLog riescono ad avere un maggior numero di registri rispetto ai ProbabilisticCounter, e ci aspettiamo che le stime fatte da HADI-Spark che implementa HLL siano migliori delle altre.

Tutte le prove sono state eseguite nelle stesse condizioni di lavoro, con il cluster formato da 15 *worker*, uno per macchina, e il *driver* caricato su una macchina a sé stante.

5.3.2 Risultati

In Figura 14 osserviamo i risultati del primo esperimento. Innanzitutto entrambe le implementazioni hanno restituito, a parità di precisione, lo stesso valore del diametro (fino a 8-9) e del diametro effettivo (fino a 4,8), come ci si attendeva dalla teoria. L'ipotesi fatta in precedenza è stata verificata, in questo caso, per valori dell'errore standard che partono da circa il 12-10%. Al di sotto di questi valori, l'algoritmo che implementa ProbabilisticCounter aumenta super-esponenzialmente il tempo di calcolo, mentre HyperLogLog avanza in modo lineare. Il momento in cui HyperLogLog diventa più performante si ottiene quando si utilizzano un numero di registri per contatore tra 64 e 128. Osservando il grafico, sembrerebbe che utilizzare i PC sia più performante fino a una precisione del 15% dell'errore standard. Come conseguenza di questo, si potrebbe pensare di modificare HADI, implementando ProbabilisticCounter e usarli nel caso di grafi di piccole dimensioni, oppure quando non serve che la precisione della stima sia troppo alta.

Rispetto a questo bisogna fare alcune considerazioni. L'implementazione di HyperLogLog che abbiamo utilizzato non è stata scritta da zero, ma si tratta di un'implementazione fatta in Java da Boldi e Vigna per i loro esperimenti su HyperANF, di cui abbiamo già parlato nel Capitolo 3. Il loro HyperLogLog è pensato perché sia un singolo oggetto Java che contiene tutti i contatori e tutti i registri, dovendo lavorare non nel cluster ma in una macchina parallela. Perciò questo oggetto contiene molte informazioni sulla struttura interna dei contatori che, per la nostra implementazione di

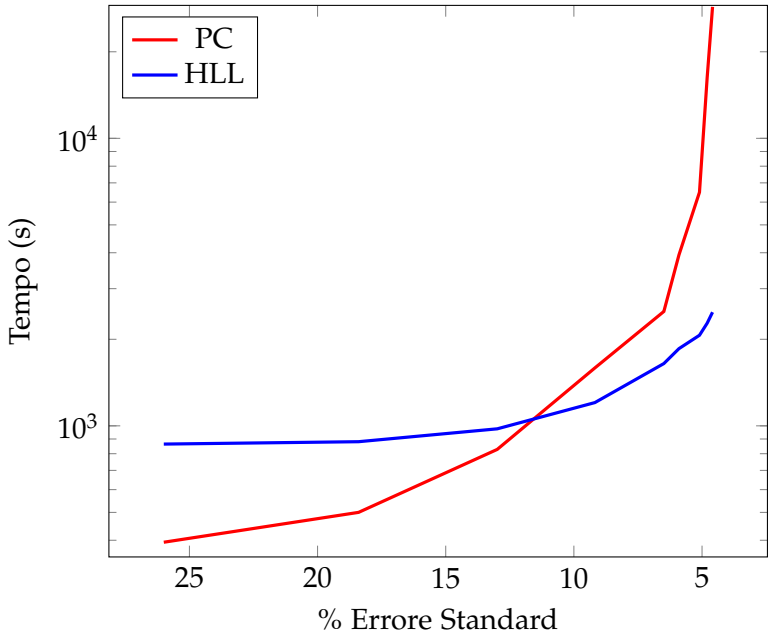


Figura 14: Confronto tra i tempi di esecuzione di HADI-Spark con HyperLogLog e con ProbabilisticCounter. Il grafo in ingresso è *orkut*.

Tabella 11: PC vs HLL a parità di peso dei registri.

		$\log_2 reg$	$d(G)$	$d_{\text{eff}}(G)$	Tempo (s)
dblp	HLL	2048	21	8,09	168
	PC	1018	20,71	7,21	179
orkut	HLL	64	8	4,76	714
	PC	87	8	4,84	1404

HADI, sono superflue. Il nostro HADI-Spark crea uno di questi oggetti per ogni singolo vertice del grafo, aggiungendo perciò un certo grado di overhead alla computazione. ProbabilisticCounter, invece, è un'implementazione pensata per HADI, molto leggera e che contiene i soli metodi di unione, conteggio e creazione che usiamo nel nostro algoritmo. Allora è ragionevole osservare, negli esperimenti, che i contatori PC con un numero basso di registri, abbiano prestazioni migliori. In futuro, con un'implementazione apposita di HLL per HADI, si potrà valutare ancor meglio questa differenza tra i due contatori.

Per quel che riguarda il secondo esperimento, ci siamo serviti di *dblp* e di *orkut* e abbiamo cercato il numero di registri dei due contatori tali che il peso di vertexRDD che fosse uguale per entrambi. Per il primo grafo, con PC di 1018 registri e HLL di 2048 registri, il vertexRDD pesava 1265,5 MB sia per l'uno che per l'altro. Allora abbiamo ripetuto dieci esperimenti per ogni tipo di contatore e abbiamo ottenuto in media i valori della Tabella 11. Vediamo che, anche se di poco, i valori ricavati sono migliori con l'uso HyperLogLog, mentre i tempi di esecuzione sono quasi uguali. Per il secondo grafo abbiamo ottenuto un vertexRDD di 1398,7 MB con PC di 87 registri e HLL con 64. Qui le differenze di stima sono minime per il calcolo del diametro, però

si osserva che l'algoritmo implementato con ProbabilisticCounter impiega il doppio del tempo per ottenere i risultati.

In sostanza, HyperLogLog è più performante di ProbabilisticCounter, come ci attendevamo, sia per quel che riguarda il tempo di calcolo, sia per quanto concerne la stima del diametro e del diametro effettivo. Va sottolineato che, per quest'ultima considerazione, sarebbero opportuni dei test aggiuntivi per evidenziare meglio la differenza nella misura del diametro.

Per concludere, questi due test ci confermano quanto abbiamo ipotizzato all'inizio, cioè che HyperLogLog è una scelta vincente per migliorare ulteriormente l'algoritmo HADI per il calcolo del diametro di un grafo, considerando sia il tempo di esecuzione sia lo spazio da esso utilizzato. Per gli esperimenti successivi ci baseremo, d'ora in poi, sulla combinazione di HADI-Spark e HyperLogLog.

5.4 TEST SU MESH

In questa sezione vogliamo valutare l'impatto che hanno le comunicazioni tra i vertici sul costo computazionale. Il risultato ci serve per comprendere, in modo più generale, in che misura lo scambio di messaggi influisce su una qualsiasi computazione effettuata su questo tipo di framework.

La versione Spark di HADI è formata principalmente da un ciclo di tre operazione: join, map, reduce. Di queste tre, il join e la reduce sono quelle più costose in quanto:

- durante il join bisogna associare i due RDD di vertici e archi mediante la chiave *vertexID*;
- nella reduce serve eseguire, per ogni contatore con *vertexID*, la massimizzazione tra questi.

Gli RDD dei vertici e degli archi sono memorizzati nel cluster e distribuiti tra le macchine. Cercare i valori associati alle chiavi e, soprattutto, trasportare i dati da una macchina all'altra comporta un significativo incremento del tempo di esecuzione dell'algoritmo.

Tuttavia, esiste un caso in cui la comunicazione tra i nodi del cluster si riduce al minimo: quando ogni macchina contiene le partizioni degli RDD di vertici e archi i quali hanno tutti le stesse chiavi. In questo modo ogni macchina del cluster sarebbe "quasi" autonoma e potrebbe eseguire le tre operazioni *in-memory* senza richiedere dati all'esterno. Per ottenere questo risultato serve conoscere quali sono le "isole" del grafo (gruppi densi di vertici collegati tra loro) e memorizzare gli identificatori associati ad essi nello stesso nodo del cluster. Siamo in presenza, purtroppo, di un altro problema la cui soluzione è complicata, perché servirebbe conoscere a priori la topologia del grafo. Questo esclude la possibilità di usare uno dei grafi della Tabella 8.

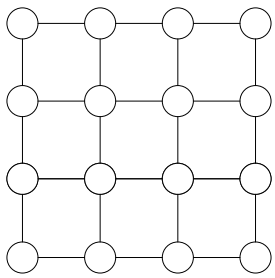


Figura 15: Una mesh con $\ell = 4$.

Un grafo di cui conosciamo la topologia e che fa al caso nostro è la *mesh* multidimensionale. In particolare, useremo la *mesh-2D*, un grafo in cui gli n vertici sono posizionati idealmente su una griglia quadrata $\sqrt{n} \times \sqrt{n}$, in cui ogni vertice è collegato

Tabella 12: Mesh utilizzate negli esperimenti con $\ell = 4 \cdot 2^i,\ i \geq 0,\ i \in \mathbb{N}$.

Lato	n	m	$d(G)$	peso (MB)
32	1024	1984	62	0,039
64	4096	8064	126	0,19
128	16 384	32 512	254	0,85
256	65 536	130 560	510	3,7
512	262 144	523 264	1022	17
1024	1 048 576	2 095 104	2046	70

ai suoi vicini *nord, sud, est, ovest* (tranne i vertici di confine). La misura fondamentale della mesh è il lato ℓ, da cui si ricavano tutti i dati successivi. Infatti una qualsiasi mesh di lato ℓ ha $n = \ell^2$ vertici e $2\ell(\ell - 1)$ archi. Inoltre, il diametro è semplicemente $2(\ell - 1)$, la distanza minima tra i vertici più lontani del grafo, cioè quelli posti ai margini della diagonale della griglia. La Figura 15 è un esempio semplice di mesh quattro per quattro.

Con un grafo di questo tipo risulta immediato ipotizzare una distribuzione dei vertici in partizioni che sia vantaggiosa per il calcolo, mantenendo su ogni nodo del cluster un sottoinsieme quadrato della mesh. Viceversa, otteniamo una distribuzione "pessima" se distribuiamo i vertici in modo che i vicini siano sempre su nodi diversi. Partiamo da queste osservazioni per sviluppare due partizionatori, ottimo e pessimo. Dopodiché faremo gli esperimenti con HADI-Spark per verificare se una migliore distribuzione riduce la quantita di shuffle richiesti e quindi se riduce il tempo di esecuzione dell'algoritmo.

5.4.1 Partizionamento equilibrato

Per rendere l'esperimento il più possibile preciso, ci interessa distribuire equamente i vertici tra i nodi del cluster, indipendentemente dal fatto che sia ottima o pessima dal punto di vista computazionale. Avendo 16 macchine, ci chiediamo quali siano le mesh che ci consentono di ottenere questo tipo di distribuzione. Procedendo intuitivamente, il caso di base è quello di una mesh di lato quattro e 16 vertici. Ogni nodo del cluster memorizza un vertice e la distribuzione tra mesh e cluster è simmetrica. Il fatto che 16 sia un quadrato perfetto ci permette di estendere facilmente la distribuzione base perché basta associare ad ogni vertice, invece di un solo vertice, una mesh di vertici. Una mesh 2×2 in ogni nodo corrisponde a una mesh di $2 \times 2 \times 16 = 64$ vertici, quella 3×3 una mesh di 144 vertici e così via. I corrispettivi lati di queste mesh sono $\ell = 4, 8, 12, 16, 20, \ldots$ multipli di quattro. Nella Tabella 12 vediamo quelle utilizzate per gli esperimenti.

Per questi motivi, torniamo alla configurazione originale del cluster, con 16 *worker* uno per macchina e un *driver* che condivide una macchina con un altro *worker*.

5.4.2 Distribuzione ottima

Cerchiamo ora un modo per distribuire i vertici nelle singole macchine in modo efficiente. Dividiamo la mesh in 16 blocchi, cioè mesh $\frac{\ell^2}{16} \times \frac{\ell^2}{16}$, così da memorizzare ogni singolo blocco su un nodo del cluster. In questo modo, con l'esclusione degli archi che uniscono i vertici confinanti, ogni sotto-mesh risiede completamente in una singola macchina. Per arrivare a questo risultato c'è bisogno di una funzione che associ ogni vertice alla partizione corretta.

A ogni vertice della mesh $\ell \times \ell$ è associato un identificatore, un valore intero nell'intervallo $[0, \ldots, \ell^2 - 1]$, e ordinati in *row-major* seguendo questa numerazione. I 16 blocchi sono anch'essi numerati in *row-major* a formare una rete 4×4. Allora possiamo pensare a una funzione che computi il numero di riga e di colonna della partizione sulla base del suo identificatore.

Definizione 5.1. Sia $V = \{0, 1, \ldots, \ell^2 - 1\}$ l'insieme degli identificatori dei vertici di una mesh $\ell \times \ell$ e $P = 0, 1, \ldots, 15$ l'insieme delle partizioni. Una funzione di distribuzione dei vertici della mesh $\ell \times \ell$ su 16 partizioni è

$$f : V \to P$$

$$f_{\text{opt}}(x) = 4\left(\left\lfloor \frac{x}{(\ell^2)/4} \right\rfloor\right) + \left\lfloor \frac{x \bmod \ell}{(\ell/4)} \right\rfloor \tag{4}$$

Facciamo un esempio. Nella Figura 16 osserviamo una mesh di lato 128 con $128^2 = 16384$ vertici. Per distribuirli in una griglia 4×4 notiamo che orizzontalmente gli identificatori sono multipli di 32, mentre verticalmente sono multipli di 4096. L'Equazione (4) diventa:

$$f_{\text{opt}}(x) = \underbrace{4\left(\left\lfloor \frac{x}{(4096)} \right\rfloor\right)}_{\text{indice di riga}} + \underbrace{\left\lfloor \frac{x \bmod 128}{32} \right\rfloor}_{\text{indice di colonna}}$$

Ogni partizione è formata da 1024 vertici (quadrato 32×32) e 1984 archi, i quali collegano i vertici interni alla partizione in tutte le direzioni. L'unica eccezione riguarda i vertici che formano il bordo. Nel caso di una partizione centrale, ci sono 128 archi che collegano i vertici del bordo con i vertici del bordo di altre partizioni, e questo sarà l'unico costo vero di comunicazione, visto che il resto delle informazioni della partizione risiede tutto in una singola macchina.

5.4.3 Distribuzione pessima

Considerando sempre una distribuzione uniforme tra le macchine, cerchiamo un modo per assegnare i vertici e gli archi ai nodi del cluster che sia "pessimo" dal punto di vista delle prestazioni. L'idea è la seguente. Se v è un qualsiasi vertice della mesh, memorizzato in un nodo del cluster a, allora tutti i vertici vicini ad esso devono risiedere in un nodo diverso da a. Così facendo, si obbligano tutti i vertici a utilizzare la rete per ricevere e/o inviare il proprio contatore HyperLogLog ai vicini.

Nel nostro caso abbiamo 16 nodi a disposizione su cui memorizzare i vertici. Possiamo pensare il nostro cluster come una mesh 4×4 con i nodi etichettati da 0 a 15 e

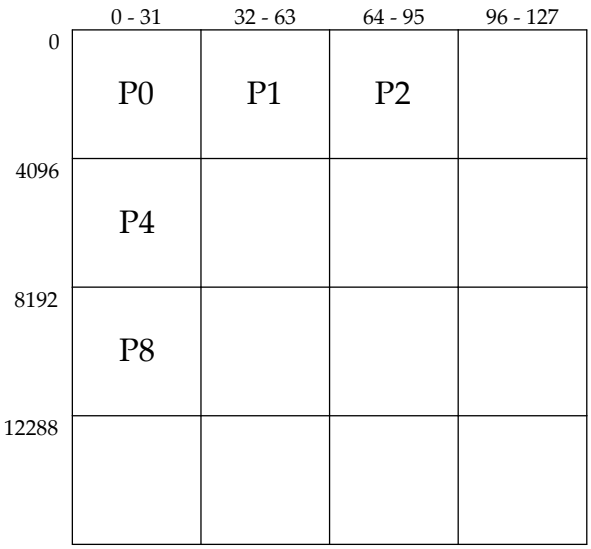


Figura 16: Partizionamento ottimale di una mesh con $\ell = 128$.

ordinati in *row-major*. Se a questa mesh sovrapponiamo una mesh 4×4, ogni vertice sarà memorizzato su una macchina diversa e otteniamo il risultato voluto. Ampliando l'idea, se ho una mesh divisibile in sotto-mesh 4×4, ed è questo il nostro caso, allora associo ognuno di queste sotto-mesh alla mia rete di nodi 4×4. I vertici interni sono chiaramente distribuiti su macchine diverse, ma tra sotto-mesh confinanti? È facile osservare che, se le $\ell^2/16$ sotto-mesh sono assegnate ai nodi in *row-major* allora non ci possono essere vertici di sotto-mesh che confinano che appartengono alla stessa macchina.

Definizione 5.2. Sia $V = \{0, 1, \ldots, \ell^2 - 1\}$ l'insieme degli identificatori dei vertici di una mesh $\ell \times \ell$ e $M = 0, 1, \ldots, 15$ l'insieme delle macchine del cluster. Una funzione di distribuzione dei vertici della mesh $\ell \times \ell$ sulle 16 macchine è

$$f : V \to M$$

$$f_{\text{wor}}(x) = 4\left(\left\lfloor \frac{x}{\ell} \right\rfloor \bmod 4\right) + \left(x \bmod 4\right) \tag{5}$$

Facciamo un esempio. Nella Figura 17 osserviamo l'angolo in alto a sinistra di una mesh di lato 128 con $128^2 = 16384$ vertici. Per distribuirli in una griglia di macchine 4×4 possiamo sfruttare la divisione e il resto per 4 da cui ricavare l'indice di riga e di colonna. L'Equazione (5) diventa:

$$f_{\text{wor}}(x) = \underbrace{4\left(\left\lfloor \frac{x}{128} \right\rfloor \bmod 4\right)}_{\text{indice di riga}} + \underbrace{\left(x \bmod 4\right)}_{\text{indice di colonna}}$$

5.4.4 Esperimenti

Oltre ai due partizionatori descritti sopra, per i test utilizziamo anche il partizionatore di default di Spark, che distribuisce i vertici in modo casuale. In questo modo, per

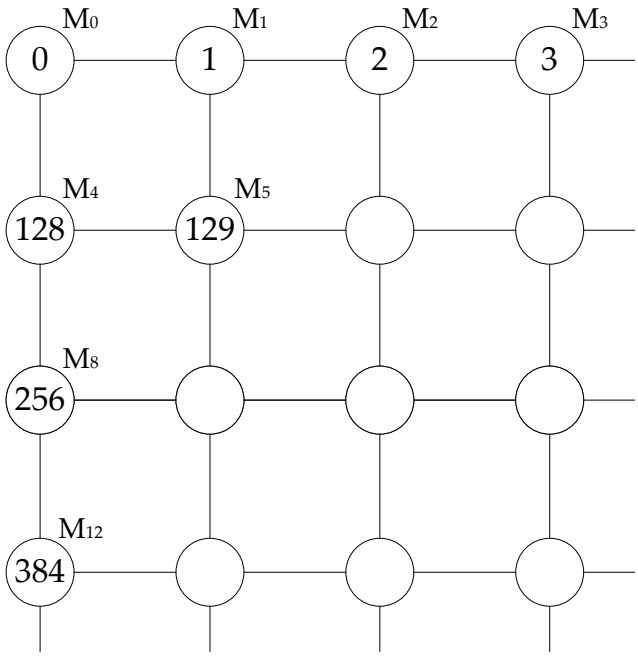


Figura 17: Partizionamento pessimo per una mesh con $\ell = 128$.

ogni mesh in input valutiamo il comportamento di HADI al variare della funzione di ripartizione dei vertici nel caso ottimo, medio e peggiore. Per quanto riguarda gli archi — ricordando che ogni arco non orientato è descritto da due archi orientati nel file descrittore del grafo — le funzioni di distribuzione si applicano in modo identico perché gli identificatori degli archi sono gli stessi dei vertici. Per esempio, sia A il nodo di un cluster e v un vertice generico della mesh. Supponiamo che questo vertice abbia due archi orientati (v, u) e (v, x) che lo collegano a due vertici u e x (per come abbiamo definito i nostri grafi, esisteranno anche gli archi (u, v), (x, v)). Se v finisce nel nodo A, allora A è anche il nodo in cui finiscono gli archi (v, u), (v, x), perché il primo elemento della coppia (v, u) è la chiave per effettuare la distribuzione. Allo stesso modo, se u e x sono vertici memorizzati in A allora anche gli archi (u, v), (x, v) saranno in A (lo stesso ragionamento vale per gli autoanelli).

Un'altro particolare riguarda l'impostazione del cluster. Per mantenere l'uniformità usiamo 16 *worker* distribuiti uno per macchina. Su una sola di queste sarà presente anche il processo *driver*, come descritto in precedenza. Inoltre, serve limitare il numero di partizioni a una per macchina, così da valutare le prestazioni con le distribuzioni uniformi che abbiamo preparato per i test. Perciò limitiamo il livello di parallelismo a 16 con l'impostazione "spark.default.parallelism = 16". Così facendo gli RDD dei vertici e degli archi saranno suddivisi in 16 partizioni, che Spark distribuirà uniformemente una per nodo del cluster.

Gli esperimenti sono stati effettuati sulle mesh in Tabella 12 e replicati per cinque volte, dopodiché abbiamo ricavato i risultati facendo la media aritmetica dei valori ritornati dal programma. L'algoritmo è HADI-Spark con i contatori HyperLogLog; che abbiamo decretato, per risultati precedenti, essere l'accoppiata migliore per la stima del diametro.

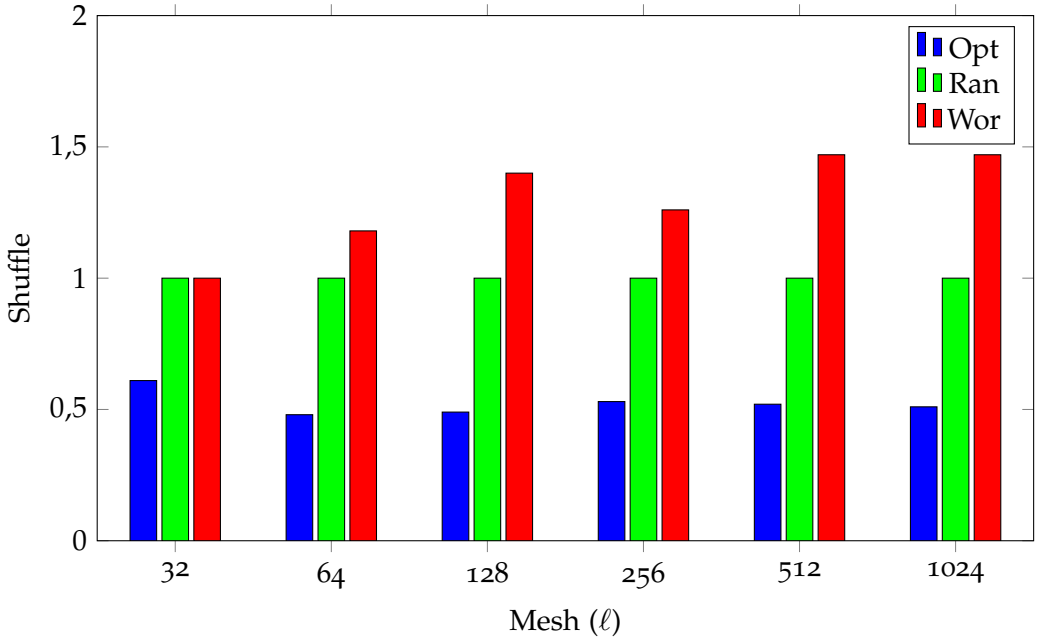


Figura 18: Numero di dati creati per lo shuffle, con diverse mesh e divisi per funzione di ripartizione. Valori normalizzati sulla distribuzione random.

5.4.5 Risultati

In Figura 18 presentiamo i primi risultati. Il grafico rappresenta gli esperimenti fatti su diversi tipi di mesh: $\ell = 32, 64, 128, 256, 512, 1024$. Ognuna di queste mesh è stato l'input per HADI-Spark testato con i tre diversi partizionamenti: ottimo, random, pessimo. Nelle ordinate troviamo il rapporto tra i dati shuffle creati dall'algoritmo al variare dei tipi di distribuzione. Questi valori sono normalizzati in base a quella random. Perciò la colonna verde vale sempre 1, mentre le altre due aumentano o diminuiscono se sono stati creati più o meno dati shuffle della distribuzione casuale. Le misure sono state così ottenute: quando HADI raggiunge l'iterazione che coincide con la metà del diametro (tutte le prove fatte hanno restituito il diametro esatto della mesh), allora contiamo, in quel ciclo, i dati creati dall'algoritmo per lo shuffle. Per accentuare le differenze tra le distribuzioni, abbiamo aumentato il peso degli HyperLogLog utilizzando ben 2^{15} registri per contatore.

Possiamo osservare che, come previsto dalla teoria, il partizionamento ottimo dei vertici diminuisce il peso degli shuffle creati durante il calcolo del diametro rispetto alla distribuzione standard operata da Spark. Anche la distribuzione pessima si comporta "bene", nel senso che comporta un aumento degli shuffle maggiore di quella random. L'unica eccezione riguarda la mesh 32×32 che è troppo piccola per poter appurare una differenza tra la distribuzione random e quella pessima. Nella Figura 19 vediamo i tempi di esecuzione. Questi seguono pari passo il rapporto che si vede in Figura 18, anche se con una differenza meno accentuata a causa dell'efficienza di Spark nel gestire le comunicazioni tra le macchine.

Un dubbio che nasce da questo esperimento è il seguente: prendendo istanti di campionamento diverso valgono ancora le considerazioni fatte? Cosa succede agli shuffle all'inizio e alla fine del calcolo del diametro? Per rispondere a queste domande vedia-

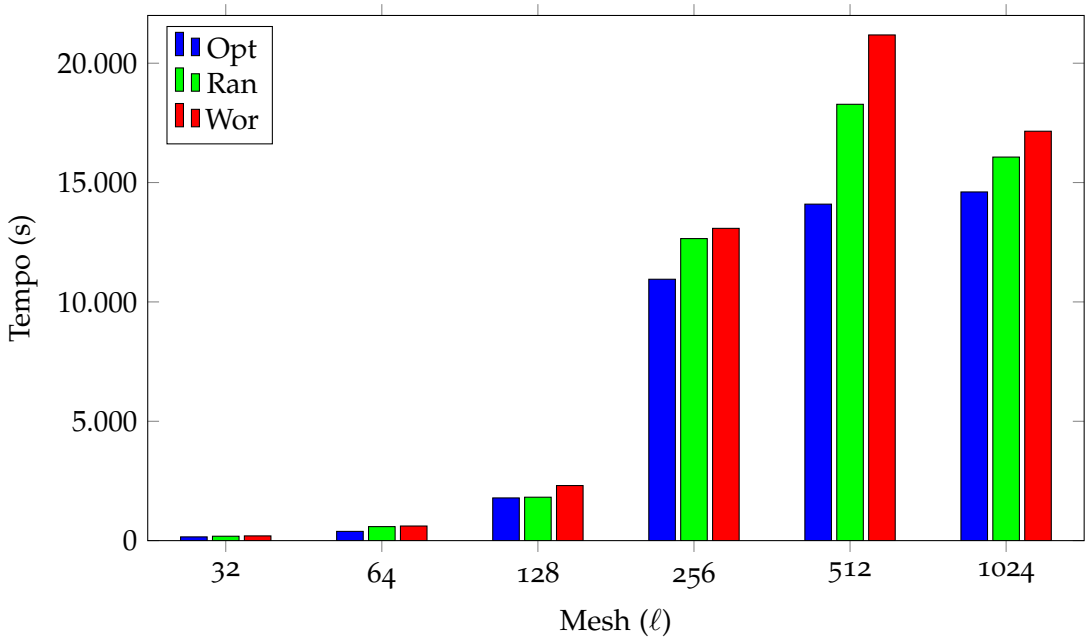


Figura 19: Tempi di esecuzione per l'esperimento in Figura 18. Le misure tra mesh diverse non vanno rapportate tra loro perché sono stati utilizzati numeri di registri diversi.

mo il risultato di un altro esperimento, in cui abbiamo registrato le variazioni degli shuffle creati per ogni iterazione dell'algoritmo. Prendiamo una mesh con $\ell = 128$ e 2^{14} registri per contatore. In queste condizioni abbiamo il vertexRDD di 646 MB. In Figura 20 possiamo vedere che la produzione di dati shuffle segue le stesse regole viste nell'esperimento precedente, con *Opt* < *Ran* < *Wor*, valide in tutti gli istanti della computazione. Notiamo che le tre curve hanno la stessa forma, con un aumento del peso degli shuffle sempre maggiore fino a circa la metà del diametro, per poi decrescere fino ad assestarsi. Questo comportamento è giustificato dall'uso delle tecniche di compressione che riducono, quando possibile, il peso delle lunghe stringhe di zero o di uno dei registri degli HyperLogLog. Infatti, supponendo sia $\log_2 n = k$ il numero di bit dei registri, questi sono formati, nella parte iniziale del processo, da stringhe di bit del tipo $[\{0,1\}^j 0000\ldots]$ con $j \ll k$, mentre verso la fine da stringhe di tipo $[1111\ldots\{0,1\}^j]$. Allora è possibile comprimere le stringhe di bit contigui a zero (all'inizio) e a uno (alla fine) risparmiando spazio e aumentando l'efficienza. Considerate queste variazioni, il rapporto tra il caso ottimo e il caso pessimo della distribuzione è circa 2,83, ottenuto come media di tutti i rapporti tra le due funzioni al crescere della stima del diametro (Figura 21).

Dagli esperimenti svolti possiamo concludere che l'impatto delle comunicazioni tra i nodi del cluster è evidente a causa della dimensione dello shuffle, che aumenta quando la distribuzione dei vertici e degli archi è "pessima", e diminuisce quando questa divisione è "buona". Per quel che riguarda HADI, questo risultato porta a pensare a un'ulteriore miglioramento dell'algoritmo. Infatti, considerando un grafo generico, se avessimo una tecnica per valutare, in tempi ragionevoli, quali sono gli insiemi più densi di vertici e archi, potremmo operare una distribuzione intelligente e migliorare ulteriormente le prestazioni.

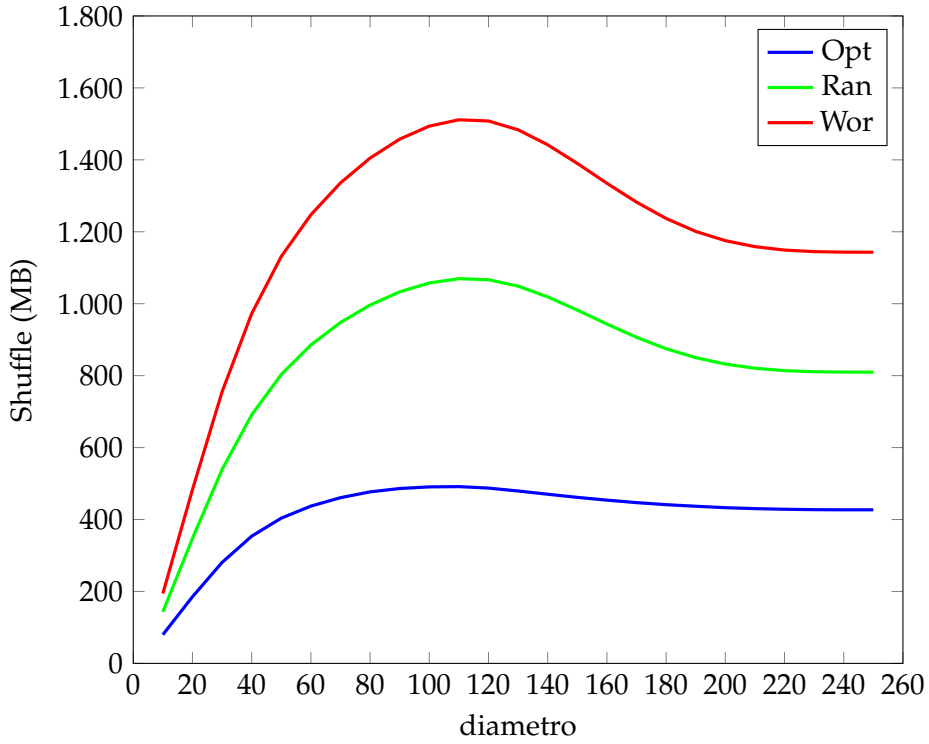


Figura 20: Valori degli shuffle per la mesh 128×128 campionati su tutte le iterazioni di HADI-Spark.

5.5 ALTRE PROVE

Concludiamo questo capitolo di esperimenti descrivendo alcuni test che sono stati eseguiti per valutare alcune idee implementative relative ad HADI e Spark.

Uso degli Accumulator

Spark mette a disposizione dei progettisti gli *Accumulator*, un tipo di variabili condivise (descritte nel secondo capitolo), che consentono ai nodi del cluster di inviare degli aggiornamenti di somma allo stesso oggetto in parallelo. L'oggetto *Accumulator* è memorizzato nella macchina processo *driver*, che riceve la comunicazione di incremento e aggiorna il valore della variabile. Pensando ad HADI-Spark e al nostro codice in 10, una procedura identica a questa appena descritta è il conteggio delle modifiche dei contatori HyperLogLog. L'ultima funzione del ciclo di HADI-Spark, alla riga 80, è la reduce che somma il numero di tutte le modifiche avvenute. Allora ci chiediamo se, invece di una reduce, non sia vantaggioso creare una variabile *Accumulator* da incrementare ogniqualvolta in counterUpd scopro che un contatore è stato modificato.

Sono stati eseguiti test con HADI-Spark implementando questa modifica, considerando tutti i grafi della Tabella 8, confrontandoli con i risultati prodotti da HADI-Spark originale. Il risultato ha mostrato che non vi è alcuna differenza tra i due modi di procedere. Né dal punto di vista del tempo di esecuzione, né dal punto di vista dello spazio richiesto dal programma. Anche variando il numero di registri per contatore non è stato possibile osservare alcuna significativa differenza. Ciò ha portato a lasciare l'implementazione di HADI-Spark così come la si era pensata all'inizio.

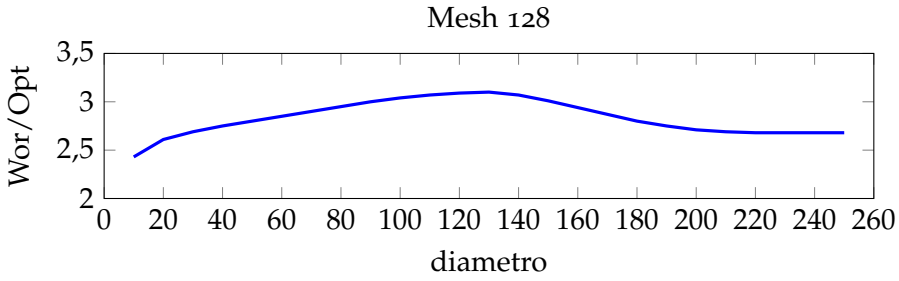


Figura 21: Rapporto Opt/Wor per la mesh 128×128.

Eliminare le operazioni superflue

Come abbiamo dimostrato nel Capitolo 3, quando un contatore di un vertice non è modificato dopo un'iterazione di HADI, allora quel contatore non verrà più modificato per tutte le iterazioni successive. Da quest'osservazione si è pensato di creare un'implementazione di HADI che distingua i vertici in due insiemi: quelli che devono continuare gli aggiornamenti e quelli che non devono più farlo. In questo modo si potrebbe ottenere un incremento delle prestazioni dovuto al taglio delle operazioni di aggiornamento dei contatori (e quindi di massimizzazione dei registri) dei vertici che non hanno più bisogno di aggiornamenti. In HADI-Pregel questo modo di operare è già presente, in quanto il discriminante per inviare o meno a un vertice i contatori dei suoi vicini è che esso sia stato modificato nell'iterazione appena trascorsa. La domanda che nasce è la seguente: includendo questa caratteristica in HADI-Spark vengono incrementate le sue prestazioni (già migliori di HADI-Pregel)? La risposta è no per il motivo seguente. Per evitare l'aggiornamento di un determinato vertice, in HADI-Spark serve eliminare i suoi archi in ingresso. Per questo c'è bisogno di aggiungere due funzioni:

- Un'operazione che filtri l'RDD risultante da edgeRDD.join(vertexRDD) (riga 58) in modo da eliminare tutti gli archi la cui chiave è presente nell'insieme dei vertici da escludere dall'aggiornamento; il che implica tenere traccia nel vertexRDD quali sono questi vertici.
- Dopo la reduceByKey (riga 74) l'RDD ottenuto non contiene tutti i vertici, perché abbiamo escluso quelli da non aggiornare. Per cui serve un'operazione che aggiunga a newVertexRDD i vertici di vertexRDD che non si trovano nel nuovo RDD appena creato.

I risultati che abbiamo ottenuto utilizzando questa nuova implementazione hanno prestazioni inferiori rispetto ad HADI-Spark originale. Utilizzando i nostri tre grafi standard si è osservato un tempo di calcolo 4-5 volte superiore al normale, e non è stato possibile osservare quel miglioramento che si era ipotizzato all'inizio.

Rispetto a questo, bisogna ammettere che non è stato fatto uno studio approfondito. Non possiamo perciò affermare che il calo delle prestazioni valga per ogni input. Potrebbe esserci un caso in cui utilizzare questo tipo di implementazione risulti vantaggioso, e solo uno studio rigoroso (con analisi teorica) fatto in futuro potrà dire se esistono benefici derivanti da quest'implementazione.

6 CONSIDERAZIONI FINALI

Concludiamo il lavoro di tesi presentando alcuni risultati, teorici e pratici, che abbiamo ricavato dopo il lungo processo di analisi, implementazione e sperimentazione di HADI. Correlato a questo risultato, gettiamo le basi per costruire un modello di calcolo per Spark, prendendo spunto dai modelli già esistenti per Hadoop e per il paradigma di programmazione MapReduce. Infine, selezioniamo alcuni argomenti e idee che potranno essere approfonditi in futuro come ulteriore sviluppo dello studio fatto qui.

6.1 MIGLIORAMENTO DI HADI

L'implementazione *HADI-Spark*, la migliore tra quelle create, ha prodotto risultati soddisfacenti in fase di test, con grafi di ogni tipo e dimensione. I motivi sono molteplici. Innanzitutto l'uso di Spark e della sua computazione *in-memory* ha permesso di diminuire la complessità computazionale dell'algoritmo, passando da un numero di operazioni dell'ordine di $O\big(d(G) * (m+n)\log(m+n)\big)$ a una complessità $O\big(d(G) * (m+n)\big)$.

Confrontando la nostra realizzazione di HADI con l'originale, abbiamo appurato che anche nella pratica il nostro algoritmo ha prestazioni maggiori. Certo, questo risultato si basa sull'ipotesi di riuscire a gestire un grafo in memoria primaria. Tuttavia, abbiamo accennato nel Capitolo 3 che questa possibilità è più che mai reale e quotidiana, perciò possiamo considerare ragionevole il confronto. Inoltre, va ricordato che HADI-Spark si basa su un solo ciclo di tipo MapReduce, mentre l'algoritmo originale ne prevedeva ben tre. Se nella teoria è un fattore costante e quindi trascurabile, nella pratica è rilevante, dato che ogni gruppo di operazioni MapReduce produce accessi in lettura/scrittura su disco.

L'altro fattore rilevante dell'implementazione è stato servirsi di HyperLogLog. I nuovi contatori sono stati fondamentali sotto ogni aspetto. La complessità in spazio è stata abbassata da $O\big(d(G) * K(m+n)\log n\big)$ come costo delle comunicazioni per HADI originale a $O\big(d(G) * (m \cdot K \log\log n)\big)$ per la nostra implementazione, diminuendo lo spazio richiesto per la creazione degli RDD su cluster e diminuendo il tempo di calcolo per ogni ciclo. Non va dimenticata la stima del diametro. Tutti i valori ricavati dagli esperimenti hanno portato a una misura estremamente vicina al valore reale del diametro, arrivando spesso al valore esatto, e il limite inferiore all'errore standard, che abbiamo descritto nel Teorema 3.2, si è dimostrato, sperimentalmente, essere un vincolo corretto, forse con un margine di sicurezza anche troppo elevato.

6.2 MODELLO DI COMPUTAZIONE PER SPARK

Conoscere i vincoli teorici di un paradigma di calcolo è uno strumento molto utile per i programmatori. Visto l'uso che abbiamo fatto di Spark, iniziamo ad abbozzare un modello di computazione per esso. Per definire il modello ci viene in aiuto la descrizione fatta da Bilardi e Pietracaprina [2], in cui descrivono un generico modello di computazione come formato da quattro parti: l'architettura di sistema, il modello di esecuzione, il modello di funzionamento (scheduling) e la funzione di costo. Presentiamo ogni pezzo del modello e ne costruiamo una bozza per Spark.

Architettura del sistema

L'architettura del sistema è intesa come l'insieme dei moduli interconnessi tra loro, ognuno con le specifiche funzioni. Allora consideriamo come ambiente di lavoro l'architettura master/slave tipica dei processi distribuiti. Il *master* è incaricato di coordinare la computazione e di assegnare i *task* alle varie macchine *slave*. Queste ricevono le istruzioni dal *master*, leggono i dati da elaborare ed eseguono le operazioni sui di essi. Ogni *slave* è un calcolatore indipendente dagli altri, con una sua memoria primaria e un suo spazio di memorizzazione su disco. Non è presente uno spazio di memoria comune, perciò la condivisione delle informazioni avviene tramite lo scambio di messaggi nella rete.

Modello di esecuzione

Si tratta di descrivere la sequenza di stati del sistema che costituiscono un'esecuzione valida. Pensando a Spark e alle operazioni sugli RDD, possiamo modellare l'esecuzione del sistema con un *grafo orientato aciclico* (DAG). Ogni algoritmo definisce, quando eseguito su una istanza di input, un DAG. Ogni suo vertice rappresenta una o più operazioni su una partizione di RDD che non richiede letture e/o scritture di shuffle, perciò è eseguita su una singola macchina dell'architettura (*narrow dependencies*). Un arco, invece, rappresenta un'operazione più complessa (*wide dependencies*). Supponendo che questo arco vada dal vertice u al vertice v, si tratta di un'operazione che produce dati di shuffle, e il cui risultato non dipende solo dalle informazioni presenti nel suo vertice input u, ma anche dai dati shuffle prodotti dagli tutti gli altri archi in ingresso al vertice v.

Riprendendo lo schema di Figura 3 nel Capitolo 2, vediamo che le tre operazioni iniziali di *map* sono eseguite su una partizione in modo indipendente, mentre la *reduceByKey* comporta uno shuffle e una distribuzione dei risultati, mentre l'ultimo *map* è ancora indipendente. A questo processo possiamo associare il DAG in Figura 22:

- Un vertice per ogni tripletta di *map* iniziali, replicato per ogni partizione di RDD (vertici 1-4 in figura);
- Quattro archi in uscita per ogni vertice, che rappresentano la funzione *reduceByKey* e lo shuffle;
- Quattro vertici per l'ultimo *map* (vertici 5-8).

Questo modello di esecuzione ci permette di visualizzare il cammino "critico" del processo — il percorso minimo più lungo nel DAG dall'input al risultato finale —

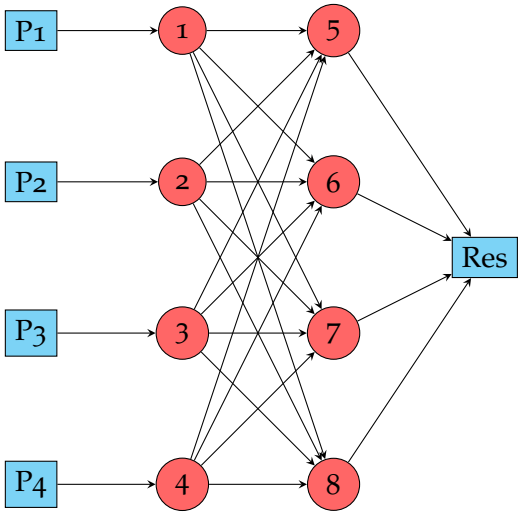


Figura 22: Modello di esecuzione DAG per il processo di Figura 3. I vertici azzurri sono i dati in input/output, mentre i vertici in rosso sono quelli che descrivono la computazione.

e valutare la complessità dell'algoritmo sulla base del numero di passi da compiere per terminare la computazione. Nella Figura 22 abbiamo differenziato i vertici di input e output perché questi non descrivono "operazioni" di computazione. Inoltre, nella definizione formale del DAG bisognerà specificare che anche gli archi uscenti dall'input e quelli entranti nel vertice finale non sono parte del processo.

Scheduling

Dal modello di esecuzione, bisogna spiegare come il DAG viene effettivamente eseguito sulla macchina parallela. Allora, rispetto a Spark, possiamo affermare che la distribuzione dei task avviene in modo uniforme tra i vari nodi della macchina. Il *master*, ogniqualvolta deve associare a un task una macchina, cerca di farlo con l'obiettivo di bilanciare il carico di lavoro complessivo di tutti i nodi.

Funzione di costo

Si tratta di trovare delle metriche di costo che hanno maggior impatto sul tempo di esecuzione e di descrivere il costo computazionale complessivo del programma in funzione di esse. Ad esempio Goodrich, per MapReduce, ha definito una funzione di costo [9] basata su:

- lunghezza del cammino "critico" del DAG associato all'algoritmo;
- volume totale dei messaggi scambiati;
- tempo di calcolo interno alla macchina parallela;

e altre misure legate alla dimensione del *buffer*, alla latenza e alla larghezza di banda. Perciò, pensando ai recenti esperimenti condotti sui grafi e sulle mesh, possiamo ipotizzare alcune metriche rilevanti per il costo computazionale.

Senza dubbio la lunghezza del cammino critico del DAG è fondamentale. Calcolando il diametro dei grafi è palese che il tempo di esecuzione aumenti linearmente

con questo valore (che nel caso di HADI coincide con il numero di iterazioni del ciclo principale). Poi ci sono il numero di operazioni sui dati, che in Spark si dividono in trasformazioni e azioni. Qui l'analisi è più difficile per via delle caratteristiche di Spark. L'elaborazione effettiva dei dati "scatta" quando il programma, avanzando nella lista di operazioni, arriva a eseguire un'azione. Le trasformazioni precedenti all'azione vengono eseguite in sequenza perciò risulta difficile distinguerle, oltre al fatto che Spark, quand'è possibile, ottimizza il processo unendo certe trasformazioni, rendendo di fatto impossibile stabilire la durata per ognuna di essa.

Ciò che distingue bene il tempo di esecuzione di un blocco di operazioni dall'altro sono gli shuffle. Quando serve leggere/scrivere dati shuffle allora si osserva un aumento significativo del tempo di calcolo. Questo è stato evidente con gli esperimenti relativi alla mesh, dove abbiamo cambiato il tipo di partizionamento. Abbiamo osservato un aumento degli shuffle (e in proporzione, un aumento del tempo di calcolo) dovuto al maggior numero di comunicazioni necessarie tra le macchine. Perciò possiamo affermare che il costo delle comunicazioni è un valore di rilievo per il nostro modello.

Con queste considerazioni, proviamo a definire una prima serie di metriche per la funzione di costo:

- t: numero di operazioni di tipo *trasformazione*;
- a: numero di operazioni di tipo *azione*;
- r: lunghezza del cammino critico del DAG associato;
- s_w: numero di messaggi prodotti;
- s_r: numero di messaggi letti;
- M_i: tempo di calcolo impiegato da una macchina nella i-esima iterazione, assumendo che sia il valore massimo tra le macchine del cluster.

Ipotizzando di avere un algoritmo che ripete sempre uno stesso gruppo di operazioni, nello stile MapReduce, possiamo dire che il tempo di esecuzione di un algoritmo implementato in Spark è:

$$T \in O\left(\sum_{i=1}^{r}\left(M_i + \sum_{k=1}^{t}(s_{r_k} + s_{w_k}) + \sum_{j=1}^{a}(s_{r_j} + s_{w_j})\right)\right).$$

Questa funzione, apparentemente molto precisa, è però di difficile applicazione per quanto detto prima. Dividere le comunicazioni tra le *azioni* e le *trasformazioni* non è semplice oltre a non essere nella filosofia di Spark entrare a questo livello di dettaglio del processo. Perciò, con un approccio ad alto livello, definiamo le seguenti metriche:

- r: lunghezza del cammino critico del DAG associato;
- o: numero di operazioni;
- s: numero di messaggi prodotti/letti;
- M_i: tempo di calcolo massimo interno alla macchina nel round i.

In quest'ottica, abbiamo una funzione più generica per il costo computazionale di Spark che è:

$$T \in O\left(\sum_{i=1}^{r}\left(M_i + \sum_{k=1}^{o} s_k\right)\right).$$

6.3 SVILUPPI FUTURI

Concludiamo con alcune ipotesi di sviluppo di questo lavoro di tesi, nate durante i vari mesi di studio e di implementazione degli algoritmi.

Innanzitutto il modello di calcolo per Spark. Tutte le quattro parti del modello descritte sopra sono una bozza di un lavoro molto più grande di formalizzazione che deve essere fatto per ritenere valido il modello. È necessario fornire degli esempi completi e adeguati di applicazione del modello a problemi comuni, come la moltiplicazione tra matrici o la somma di interi. Inoltre bisogna valutare il modello in termini di portabilità, facilità di utilizzo ed efficacia nel descrivere le prestazioni di un programma.

Rispetto all'implementazione di HADI, ci sono molti aspetti che possono essere modificati e testati:

- I grafi in input sono stati gestiti con due RDD, uno di vertici e uno di archi. Un'implementazione differente si ottiene con le liste di adiacenza. Si crea un solo RDD contenente n elementi, uno per ogni vertice u, che rappresenta la lista di adiacenza di quel vertice. Ogni lista è formata da coppie $(v, counter(v))$ dove v è un vicino del vertice u. In quest'ottica, si modifica l'ordine delle tre operazioni *join,map,reduce* di HADI-Spark e si usa un solo RDD invece che due. Dal punto di vista teorico, la complessità non cambia, mentre non è detto che lo sia in pratica. Solo gli esperimenti potranno dire se questo tipo di implementazione è più o meno efficiente del nostro HADI-Spark.

- Per gli esperimenti abbiamo utilizzato i contatori HyperLogLog scritti da Boldi e Vigna per HyperANF. Il loro uso è stato prezioso e i risultati ottenuti lo dimostrano. Resta da capire se un'implementazione *ad hoc* per Spark possa essere ancora più efficiente. Dalle considerazioni fatte nel Capitolo 5 osserviamo che, almeno per valori dei registri inferiori a 128, potrebbe essere una valutazione corretta.

- Alla fine del Capitolo 5 abbiamo accennato ad alcuni esperimenti relativi a delle idee di implementazioni diverse da HADI-Spark. Una di queste era quella di evitare gli aggiornamenti degli HyperLogLog che hanno raggiunto il loro stato finale, e che non saranno modificati in iterazioni successive. I risultati raggiunti fanno pensare che questa modifica di HADI-Spark non sia utile per migliorare le prestazioni. Tuttavia, solo studiare il problema nella sua totalità, soprattutto con un'analisi teorica, potrebbe confutare ogni dubbio.

- I risultati degli esperimenti su mesh hanno evidenziato che, una distribuzione ottimale dei vertici e degli archi tra i nodi del cluster, ha come effetto la diminuzione del numero di comunicazioni tra le macchine, con un conseguente abbassamento del tempo di calcolo. Si potrebbe ipotizzare un'implementazione

di HADI che includa un algoritmo di ricerca su grafo che trovi, in tempo ragionevole, quali sono le isole di vertici fortemente connesse. Conoscendo questo dato potremmo creare una funzione di partizionamento che distribuisca in modo intelligente i vertici — idealmente come abbiamo fatto con la mesh — per abbattere il volume delle comunicazioni e migliorare ulteriormente le prestazioni. Sarà fondamentale in questo lavoro capire se l'overhead dovuto alla funzione di ricerca inserita è ragionevole rispetto al tempo di calcolo totale. Non è un'idea semplice da realizzare, tuttavia, se si dimostrasse che le ipotesi fatte sono corrette, otterremmo un risultato davvero notevole.

BIBLIOGRAFIA

[1] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, Sebastiano Vigna (2012), Four degrees of separation, *Web Science Conference*, pagine 33–42.

[2] Gianfranco Bilardi, Andrea Pietracaprina (2011), Theoretical Models of Computation, *Encyclopedia of Parallel Computing*, Springer, pagine 1150–1158.

[3] Paolo Boldi, Marco Rosa, Sebastiano Vigna (2011), HyperANF: Approximating the Neighbourhood Function of Very Large Graphs on a Budget, *Proceedings of the 20th international conference on World Wide Web*, pagine 625–634.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009), Sorting in Linear Time *in* Introduction to Algorithms (3. ed.), *MIT Press*, ISBN 978-0-262-03384-8, pagine 194–197.

[5] Jeffrey Dean, Sanjay Ghemawat (2008), MapReduce: Simplified Data Processing on Large Clusters, *Communication of the ACM*, Vol. 51, no. 1, pagine 107–113.

[6] Marianne Durand, Philippe Flajolet (2003), Loglog Counting of Large Cardinalities (Extended Abstract), *11th Annual European Symposium (ESA)*, Springer, pagine 605–617.

[7] Philippe Flajolet, Éric Fusy, Olivier Gandouet (2007), Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm, *Proceedings of the 13th conference on analysis of algorithm (AofA 07)*, pagine 127–146.

[8] Philippe Flajolet, G. Nigel Martin (1985), Probabilistic Counting Algorithms for Data Base Applications, *Journal of Computer and System Sciences*, 31(2), pagine 182–209.

[9] Michael T. Goodrich (2010), Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry, *The Computing Research Repository (CoRR)*, abs/1004.4708.

[10] Michael T. Goodrich, Roberto Tamassia (2007), Data structures and algorithms in Java (4. ed.), *Zanichelli*, pagine 519–523.

[11] Joseph JáJá (1992), An Introduction to Parallel Algorithms, *Addison-Wesley*, ISBN 0-201-54856-9.

[12] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, Jure Leskovec (2011), HADI: Mining Radii of Large Graphs, *ACM Trans. Knowl. Discov. Data*, ACM.

[13] Jure Leskovec, Andrej Krevl (2009), SNAP Datasets: Stanford Large Network Dataset Collection, `http://snap.stanford.edu/data/`.

[14] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski (2009), Pregel: a system for large-scale graph processing, *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, New York, USA, pagina 48.

[15] Martin Odersky, Lex Spoon, Bill Venners (2010), Programming in Scala (2. ed.), *Aritma*, ISBN 9780981531649.

[16] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, Eli Upfal (2012), Space-round Tradeoffs for MapReduce Computations, *International Conference on Supercomputing*, pagine 235–244.

[17] The Apache Software Foundation (2014), Spark 1.1.0 Documentation, https://spark.apache.org/documentation.html.

[18] The Apache Sortware Foundation (2014), Spark 1.1.0 API Specification, https://spark.apache.org/docs/1.1.0/api/scala/index.html.

[19] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica (2013), GraphX: a resilient distributed graph system on Spark, *First International Workshop on Graph Data Management Experiences and Systems, GRADES, co-loated with SIGMOD/PODS*, New York, USA, pagina 2.

[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, Michael J Franklin, Scott Shenker, Ion Stoica (2012), Fast and Interactive Analytics over Hadoop Data with Spark, *USENIX ;login:*, Vol. 37, no. 4, pagine 45–51.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, Ion Stoica (2012), Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *Networked Systems Design and Implementation (NSDI)*, pagine 15–28.

[22] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica (2010), Spark: Cluster Computing with Working Sets, *HotCloud 2010*.