UNIVERSITÀ DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
TESI DI LAUREA MAGISTRALE

VERIFICA SPERIMENTALE DI UN MODELLO PER MAPREDUCE

RELATORI: Ch.mo Prof. Geppino Pucci
Ch.mo Prof. Andrea Alberto Pietracaprina
CORRELATORE: Dr. Francesco Silvestri

LAUREANDO: *Paolo Rodeghiero*

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Padova, 12 Marzo 2012

Indice

Sommario 1

Introduzione 3

1 MapReduce 7

1.1 Il Paradigma . . . 8

1.1.1 Le Radici nella Programmazione Funzionale . . . 8

1.1.2 Formato dei Dati . . . 9

1.1.3 Le funzioni *map* e *reduce* . . . 10

1.1.4 Fasi della Computazione . . . 11

1.1.5 Un esempio: Word Count . . . 11

1.2 L'Implementazione Originale . . . 13

1.3 Motivazioni Pratiche . . . 14

1.3.1 Parallelizzazione implicita . . . 14

1.3.2 Scalabilità semplice . . . 15

1.3.3 Elaborazioni batch data-intensive . . . 15

1.3.4 Muovere il calcolo verso il dato . . . 16

1.3.5 Scaling out, not up . . . 16

1.3.6 Guasti Frequenti . . . 17

1.4 Critiche: MapReduce e i Database Distribuiti . . . 17

1.5 Altre Implementazioni . . . 18

2 Il Modello 21

2.1 Il Problema della Nomenclatura: Mapper e Reducer . . . 22

2.2 Il Modello di Riferimento: MR(m,M) . . . 22

2.2.1 Eliminazione della fase di Map . . . 22

2.2.2 Definizione di Algoritmo . . . 23

2.2.3 Vincoli 24
2.2.4 Modello di costo 25
2.3 Lavori precedenti 26

3 Apache Hadoop 29
3.1 Introduzione 30
3.1.1 Breve storia di Hadoop 30
3.1.2 Versioni disponibili 31
3.1.3 L'ecosistema Hadoop 31
3.2 Introduzione ad Hadoop 32
3.2.1 Tipi di nodi 33
3.3 HDFS: il FileSystem Distribuito 33
3.3.1 Infrastruttura 35
3.3.2 Il Ruolo della Replicazione 37
3.3.3 Anatomia delle operazioni principali 38
3.4 MapReduce: l'Infrastruttura di Calcolo 40
3.4.1 Infrastruttura 40
3.4.2 Job, Task e Task Attempt 42
3.4.3 Dalle funzioni agli oggetti 43
3.4.4 Parallelismo e Numero di Task 45
3.4.5 Formato dei Dati 46
3.4.6 Input e Output di un job 49
3.4.7 Anatomia di un job 51
3.5 Differenze Rispetto a Modello e Paradigma 55

4 Moltiplicazione Matriciale Densa 57
4.1 L'algoritmo in MR(m,M) 57
4.2 Implementazione 64
4.2.1 Scelte comuni 64
4.2.2 Versioni: Strategie e Livelli 67

5 Verifica Sperimentale 73
5.1 Apparato Sperimentale 74
5.1.1 Hardware 74
5.1.2 Configurazione di Hadoop 74

5.1.3 Resilienza e replicazione 76
5.2 Generazione dei Dati 78
5.3 Misure 79
5.4 Limitazioni riscontrate ai valori dei parametri 82
5.5 Versioni altamente scalabili 88
5.6 Risultati 90
5.6.1 Il numero di round, il numero di chiavi distinte ed il fattore di replicazione 91
5.6.2 Serie 1 ($n = 2^{24}$,$M = 2^{30}$) 93
5.6.3 Matrici di grandi dimensioni 98
5.7 Considerazioni sui risultati 101

Conclusioni **103**

A Nomenclatura Essenziale **107**
A.1 Nomenclatura Hadoop 107
A.1.1 Principale 107
A.1.2 Schedulazione 108
A.1.3 Shuffle esteso 109
A.1.4 Infrastruttura 109

B Note su Hadoop **111**
B.1 Hadoop e Slurm 111
B.2 Requisiti per la risoluzione dei nomi 113

C Codice Sviluppato **115**
C.1 Problematiche di debug 115
C.2 Analisi dei dati: MapReduceAnalytics 118
C.3 Struttura del Codice 119
C.4 Esempi di Codice 120
C.4.1 Driver 120
C.4.2 Job Factory 123
C.4.3 Mapper 125
C.4.4 Reducer 125

Sommario

> Al tempo dei pionieri questi usavano i buoi per trainare i carichi pesanti, e quando un bue non riusciva a tirare un tronco essi non tentavano di allevare buoi più grossi. Noi non dobbiamo provare a costruire computer più grandi, ma sistemi di più computer - Grace Hopper,

Proposto nel 2005 a partire dall'esperienza implementativa di Google, MapReduce rappresenta a tutt'oggi uno dei più utilizzati paradigmi per il calcolo parallelo. A causa della sua origine empirica manca però di un modello condiviso, così come di una terminologia consistente. Dato il modello per MapReduce MR(m,M), sviluppato all'interno del Dipartimento, in questo lavoro ne viene esplorata l'implementabilità e validità rispetto ad Hadoop, il principale framework Open Source per MapReduce, utilizzando un algoritmo per la moltiplicazione tra matrici quadrate dense. Dopo uno studio dei meccanismi interni di Hadoop, l'algoritmo viene implementato in alcune varianti, che sono confrontate poi in termini di prestazioni su un cluster reale al variare dei parametri. Infine sono individuate alcune indicazioni di carattere generale sul rapporto tra Hadoop e modello.

Introduzione

Per molti anni il calcolo parallelo è stato relegato ad essere un settore di nicchia: la maggior parte dei problemi comuni infatti risultava risolvibile semplicemente tramite l'uso di singole macchine mono processore, fatta eccezione per alcune necessità particolari (primariamente in ambito scientifico). Negli ultimi anni però la possibilità di aumentare le prestazioni di singoli processori tramite le tecniche utilizzate fino ad allora (aumento della frequenza di clock, processori superscalari) si è dimostrata economicamente poco remunerativa, ed i produttori hanno iniziato a proporre processori con un numero di core crescente [18].

Contemporaneamente, in molti contesti è emersa la necessità di svolgere elaborazioni di grandi quantità di dati. Se nel passato solo alcune organizzazioni necessitavano di queste elaborazioni, ora la necessità di risolvere problemi su grandi quantità di dati è abbastanza diffusa a causa di molteplici fattori, come ad esempio il diminuire dei costi di memorizzazione e l'insorgere di applicazioni che generano dati molto velocemente. Questi dati posso essere di molti tipi: ad esempio dati scientifici, finanziari, informazioni di monitoraggio. Sono moltissime le organizzazioni che possono trarre vantaggio dall'analizzare ed estrarre informazioni significative da questi dati.

Gestire ed elaborare queste grandi quantità di dati risulta però complesso. Il diminuire dei costi di memorizzazione, e quindi le dimensioni delle memorie di massa, non è andato di pari passo alle possibilità di accedervi. Ad esempio nel 1990 un tipico disco aveva la capacità di 1.37 MB abbinato ad una velocità di trasferimento di 4.4 MB/s, rendendo possibile leggere l'intero contenuto in 5 minuti. Al giorno d'oggi, dopo più di 20 anni, dischi di capacità intorno al TeraByte sono la norma: la capacità è aumentata di un fattore 1000. Le velocità di trasferimento attuali sono invece aumentate solo di un fattore 25 e si attestano dell'ordine di 100 MB/s, rendendo quindi necessarie più di due ore per accedere all'intero contenuto del disco [29]. Una possibile soluzione ai tempi di accesso è

utilizzare la banda di accesso aggregata di più dischi distribuiti su più macchine, le quali a loro volta possono eseguire elaborazioni parallele, sfruttando tutti i core disponibili. In questo nasce e trova molte applicazioni pratiche il paradigma MapReduce.

Proposto nel 2005 da Dean e Ghemawat [4], MapReduce è uno dei principali fattori che hanno reso il calcolo parallelo su grandi quantità di dati molto più diffuso di quando non lo fosse in passato. Il paradigma nasce dalla esperienza acquisita da parte di Google in questo campo, ed è attualmente utilizzato in molte applicazioni all'interno dell'azienda. Esso permette di specificare le computazioni nascondendo molti dei complessi dettagli riguardanti la comunicazione e la parallelizzazione, delegando questi aspetti ad una libreria esterna. Il framework originario che permette l'esecuzione di programmi MapReduce non è disponibile al pubblico, ma ne esiste una implementazione open-source, chiamata Hadoop, che è utilizzata in molti settori e rappresenta attualmente il sistema parallelo più rilevante in termini di numero di installazioni. Il principale utilizzo di MapReduce è fortemente legato alle sue origini: viene infatti impiegato principalmente per applicazioni di web mining, o per l'estrazione di informazione da dati non normalizzati. Per sua struttura, MapReduce si sposa particolarmente bene con il concetto di *utility computing*, permettendo l'accesso al calcolo parallelo su grandi moli di dati a soggetti che prima ne erano esclusi a causa del costo proibitivo dell'infrastruttura.

MapReduce assume inoltre importanza in quanto rappresenta il primo paradigma computazionale largamente diffuso che si allontana dal modello di Von Neumann. Esso può considerato come un paradigma *bridging*, ossia un ponte concettuale tra l'hardware fisico ed il software destinato ad esservi eseguito [18, p.11]. Il suo scopo è semplificare, astraendo, l'utilizzo delle risorse aggregate del cluster, accelerando i tempi di sviluppo, test e messa in produzione di applicazioni parallele. Esso non rappresenta un approccio strettamente innovativo, ma un approccio di successo che prende spunto da molta letteratura precedente e dall'esperienza applicativa maturata all'interno di Google.

Dal punto di vista dell'informatica teorica, MapReduce manca però di un modello condiviso, a causa della sua origine fortemente empirica. Non è chiaro quali problemi siano effettivamente risolvibili utilizzando questo paradigma così come risulta difficile studiare la complessità temporale e la scalabilità di un programma

MapReduce.

Obbiettivo

All'interno del Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, ed in collaborazione con la Brown University, è stato proposto un modello per MapReduce, detto MR(m,M) [22]. L'obbiettivo di questa Tesi è verificare questo modello rispetto all'implementazione Hadoop, individuare eventuali differenze, cercare di caratterizzarne sperimentalmente la complessità temporale ed identificare i costi nascosti che possono emergere. A questo scopo è stato utilizzando l'algoritmo per la moltiplicazione matriciale densa proposto insieme al modello. Questa scelta ha due risvolti importanti: da una parte la moltiplicazione matriciale non è uno dei problemi di elezione che hanno portato allo sviluppo di MapReduce; dall'altra essa è una importante primitiva del calcolo, utilizzata in molte applicazioni reali.

Panoramica della Tesi

Nel **Capitolo 1** è presentata una introduzione dettagliata del paradigma MapReduce ed alle ragioni alla base del suo successo ed alcune critiche che sono state poste. Successivamente, nel **Capitolo 2**, viene introdotto il modello MR(m,M), il quale è poi confrontato con alcuni modelli precedenti. Nel **Capitolo 3** è invece descritto Hadoop nel dettaglio, evidenziando alcuni aspetti che si ritengono rilevanti per la comprensione dei risultati sperimentali e le differenze rispetto al modello; nel **Capitolo 4** viene poi introdotto l'algoritmo di moltiplicazione matriciale, utilizzato in fase sperimentale, e la sua implementazione in Hadoop. Infine nel **Capitolo 5** è descritta l'infrastruttura sperimentale e sono presentati e discussi alcuni risultati. Al termine della Tesi sono infine presentate delle conclusioni di carattere generale, utili ad orientare il lavoro futuro.

Durante lo sviluppo del lavoro è risultato evidente un importante problema di nomenclatura: molti termini assumono anche significati contrastanti nei vari contesti, siano essi di paradigma, modellistico o relativo allo specifico framework. Durante la stesura della Tesi si è cercato di separare i vari concetti ed uniformare la terminologia, cercando così di sviluppare un corpus il più omogeneo e chiaro. Per questo motivo in **Appendice A** è presentata una lista dei termini essenziali e l'accezione con cui sono utilizzati.

Inoltre, essendo questo il primo lavoro sperimentale su MapReduce svolto all'interno del Dipartimento, è stata necessario raccogliere un imponente quantità di informazioni e maturare esperienza implementativa rispetto ad Hadoop. Parte di questa esperienza, che non trovava una collocazione naturale all'interno della parte principale della Tesi, è raccolta in **Appendice B**.

Codice sviluppato

Si è ritenuto opportuno non allegare in formato cartaceo il codice sviluppato. Questo codice contiene sia l'implementazione Hadoop di tutte le versioni della Tesi, sia uno strumento sviluppato per analizzare e correlare i dati prodotti. L'intero codice è disponibile in formato digitale insieme alla Tesi e pubblicamente presso http://www.linux.it/~rod/thesis, rilasciato sotto licenza Affero GNU Public License, versione 3. [1] In **Appendice C** è inserita una breve documentazione dello stesso, e sono presentati alcuni estratti di codice a titolo di esempio.

[1] http://www.gnu.org/licenses/agpl.html

Capitolo 1

MapReduce

> [...] as soon as you think of map and reduce as functions that everybody can use, and they use them, you only have to get one supergenius to write the hard code to run map and reduce on a global massively parallel array of computer.
>
> Joel Spolsky [27]

Come accennato in introduzione, MapReduce deriva dall'esperienza implementativa maturata all'interno di Google nel calcolo parallelo su grandi moli di dati [4]. Le forti esigenze di parallelizzazione presenti all'interno dell'azienda hanno portato a creare una astrazione che potesse accelerare lo sviluppo di applicazioni parallele, relegando i dettagli relativi alla parallelizzazione, gestione dei guasti e distribuzione dei dati ad una libreria. A causa di questa origine empirica, MapReduce manca di un insieme coerente di terminologia e di definizioni condivise a livello generale. Il termine MapReduce stesso viene utilizzato sia per indicare il modello computazionale, sia il framework concettuale necessario, sia le specifiche implementazioni. Esiste quindi una certa confusione su cosa sia in realtà il paradigma e dove questo sia applicabile o efficace.

L'obbiettivo principale di questo capitolo è introdurre il paradigma MapReduce e la relativa nomenclatura nel modo più generale ed astratto possibile, cercando di eliminare quei dettagli che fanno riferimento a specifiche implementazioni. Seguono poi le motivazioni di carattere pragmatico alla base della scelta progettuale ed al successo della sua implementazione originale, ed una sezione riguardo ad alcune critiche avanzate al paradigma. Alla fine del capitolo, in Sezione 1.5, vengono inoltre elencate alcune implementazioni. Il problema dell'espressi-

vità del paradigma invece va oltre lo scopo di questa Tesi: alcuni riferimenti a questo proposito possono essere trovati nel lavoro di Karloff [16].

1.1 Il Paradigma

1.1.1 Le Radici nella Programmazione Funzionale

L'approccio utilizzato in MapReduce non è innovativo, ma risulta una combinazione di concetti noti nella letteratura. In particolare prende spunto dalle funzioni map e $fold$ tradizionalmente presenti all'interno dei linguaggi di programmazione funzionale. In questo tipo di programmazione è possibile utilizzare funzioni che utilizzano come parametri altre funzioni. Le funzioni map e $fold$ sono funzioni di questo tipo; esse sono spesso applicate in coppia, prima la funzione map e poi la funzione $fold$, permettendo di trasformare ed aggregare un array.

La funzione map prende come parametri un array ed una funzione f con un solo parametro, e restituisce un array contente gli elementi dell'array di partenza a cui è applicata la funzione$f$:

$$map : ([x_0, x_1, ..., x_{n-1}], f(.)) \mapsto [f(x_0), f(x - 1), ..., f(x_{n-1})]$$

La funzione $fold$ invece prende come parametri un array, un valore iniziale, ed una funzione g con due parametri. Per prima cosa $fold$ chiama la funzione g usando come parametri il valore iniziale ed il primo elemento dell'array, memorizzando il risultato in una variabile temporanea. Chiama poi ancora la funzione usando come parametri la variabile temporanea ed il secondo valore dell'array, memorizzando sempre il risultato nella variabile temporanea. Procede poi in questo modo su tutti gli elementi, e restituisce il valore della variabile in uscita:

$$fold : ([y_0, y_1, ..., y_{n-1}], g(., .), s) \mapsto g(y_{n-1}, g(y_n, ...g(y_0, s)...))$$

La funzione map è banalmente parallelizzabile, essendo le chiamate tra loro indipendenti, mentre questo non è vero non per la funzione $fold$. In molte applicazioni però la funzione g deve essere applicata solo ad una parte degli elementi: se gli elementi dell'array possono essere raggruppati in parti su cui poi applicare $fold$ indipendentemente, anche questa parte della computazione è parallelizzabile [18, pag. 19].

Da questo assunto parte MapReduce: considerando gli algoritmi come una sequenza di computazioni funzionali in due fasi, ovvero trasformazione ed aggregazione per gruppi, a livello implementativo è possibile relegare i dettagli relativi alla distribuzione dei dati ed alla parallelizzazione ad una libreria.

MapReduce differisce dalla computazione funzionale tramite *map* e $fold$ per alcuni dettagli, tesi a semplificare ulteriormente la parallelizzazione.

1.1.2 Formato dei Dati

La prima differenza importante riguarda il formato dei dati: il paradigma MapReduce prevede infatti che i dati, sia in input che in output, vengano visti come coppie di entità: la prima componente viene detta chiave, la seconda componente valore. Entrambe le componenti non sono definite a priori, ma sono libere di assumere qualsiasi forma: l'insieme di definizione è libero e dipendente dalla specifica computazione.

Ogni computazione è caratterizzata, in generale, da tre diversi multi-insiemi di coppie: il multi-insieme di input, il multi-insieme intermedio ed il multi-insieme di output.

Definizione 1.1 (multi-insieme di input - $\mathcal{I}$)**.** *Viene detto* multi-insieme di input *($\mathcal{I}$) il multi-insieme composto dalle coppie* $(k_1, v_1) \in K_1 \times V_1$ *che corrispondono ai dati di input. L'insieme* K_1 *è detto insieme delle chiavi di input e* V_1 *insieme dei valori di input, entrambi dipendenti dalla specifica computazione.*

Definizione 1.2 (multi-insieme intermedio - $\mathcal{S}$)**.** *Viene detto* multi-insieme intermedio *($\mathcal{S}$) il multi-insieme composto dalle coppie* $(k_2, v_2) \in K_2 \times V_2$ *generate durante la computazione.* K_2 *è detto insieme delle chiavi intermedio e* V_2 *insieme dei valori intermedio, entrambi dipendenti dalla specifica computazione.*

Definizione 1.3 (multi-insieme di output - $\mathcal{O}$)**.** *Viene detto* multi-insieme di output *($\mathcal{O}$) il multi-insieme composto dalle coppie* $(k_3, v_3) \in K_3 \times V_3$*, corrispondente all'output della computazione.* K_3 *è detto insieme delle chiavi di output e* V_3 *insieme dei valori di output, entrambi dipendenti dalla specifica computazione.*

A livello di paradigma non esiste alcun vincolo tra gli insiemi di definizione $(K_1, V_1, K_2, V_2, K_3, V_3)$ che possono essere tra loro coincidenti, intersecanti o disgiunti.

1.1.3 Le funzioni *map* e *reduce*

L'altra componente chiave del paradigma è la definizione di due funzioni, dette funzione *map* e funzione *reduce*. Queste funzioni definiscono completamente la computazione ed operano sia sulle chiavi sia sui valori.

Definizione 1.4 (Funzione map). *Fissato il multi-insiemi di input, una funzione map su questo multi-insieme è una funzione tale che*

$$map : (k_1, v_1) \mapsto multiset(k_2, v_2)$$

e si ha che il multi-insieme intermedio risulta

$$\mathcal{S} = \bigcup_{(k_1,v_1) \in \mathcal{I}} (map(k_1, v_1))$$

dove il simbolo di unione denota l'unione di multi-insiemi.

La funzione *map* svolge il ruolo di trasformazione della singola coppia in input, analogo alla sua omonima nella programmazione funzionale; a differenza di questa però l'output in MapReduce può produrre un multi-insieme di elementi, anche vuoto.

Definizione 1.5 (Funzione reduce). *Fissato il multi-insieme intermedio, e sia G_k un sotto-multi-insieme del multi-insieme intermedio formato da coppie con la stessa chiave, ossia*

$$G_k = \{(k_2, v_2) : k_2 = k \wedge (k_2, v_2) \in \mathcal{S}\}$$

una funzione reduce su questo multi-insieme è definita come la funzione che

$$reduce : G_k \mapsto multiset(k_3, v_3)$$

e si ha che il multi-insieme finale risulta

$$\mathcal{O} = \bigcup_{k \in K_2} (reduce(G_k))$$

dove il simbolo di unione denota l'unione di multi-insiemi.

A differenza della funzione $fold$ nella programmazione funzionale, in MapReduce la funzione *reduce* non è vincolata a svolgere una aggregazione. La cardinalità del multi-insieme prodotto da una chiamata a *reduce* può a tutti gli effetti essere maggiore rispetto a quella fornita in input, anche se in molte applicazioni è prodotto al più un valore per ciascuna invocazione di *reduce* [4].

1.1.4 Fasi della Computazione

Dopo aver introdotto le funzioni *map* e *reduce*, è possibile vedere come si svolge, a livello paradigmatico, una singola computazione in MapReduce. A questo livello, è possibile suddividere la computazione in tre fasi sequenziali: map, shuffle e reduce.

- La fase di map lavora direttamente sull'input. La funzione *map* viene invocata su ciascuna coppia del multi-insieme di input, ottenendo ad ogni chiamata un multi-insieme (potenzialmente vuoto) di coppie $(k_2, v_2) \in K_2 \times V_2$.

- La fase di shuffle, implicita e gestita dal sistema, si occupa di raggruppare per chiave tutte le coppie prodotte nella fase precedente. Vengono quindi ottenuti dei multi-insiemi di coppie con la stessa chiave.

- Nella fase di reduce, la funzione *reduce* viene invocata sui multi-insiemi G_k ottenuti nella fase precedente. Per ogni chiamata della funzione *reduce* viene prodotto un multi-insieme di coppie (anch'esso potenzialmente vuoto).

Tutte le coppie prodotte nella fase di reduce, che costituiscono quindi il multi-insieme di output, sono l'output effettivo della computazione.

1.1.5 Un esempio: Word Count

Vediamo un esempio di una computazione specificata tramite il paradigma computazione MapReduce. L'esempio è limitato rispetto alle possibilità del paradigma, ed è appositamente pensato per essere il più semplice possibile, anche a spese dell'efficienza. Il suo scopo è quello di chiarire le definizioni delle funzioni e dei multi-insiemi, così come fornire un esempio di dati visti come coppie chiave-valore.

Esempio 1. [4] Prendiamo in considerazione il problema del calcolo del numero di occorrenze di ogni parola presente in una grande collezione di documenti. Siano

quindi

$$\begin{aligned} K_1 &= \{\text{stringhe } w : w \text{ è il nome di un documento}\} \\ V_1 &= \{\text{stringhe } t : t \text{ è il testo di un documento}\} \\ K_2 &= K_3 = \{\text{stringhe } p : p \text{ è una parola}\} \\ V_2 &= V_3 = \{n \in \mathbb{N}\} \end{aligned}$$

Il multi-insieme di input $\mathcal{I}$ è composto da tutte le coppie $(nome, contenuto)$.

Nella fase di map, viene applicata ad ogni coppia in input la funzione *map*. In questa funzione per ogni parola del documento viene aggiunto nel multi-insieme di output una coppia con chiave la parola e valore 1. Vista in pseudocodice, è

```
function map (k1, v1) begin
    O ← ∅;
    foreach w in document k1 do
        O ← O ∪ {(w, 1)};
    return O;
```

Nella fase di shuffle, implicita, tutte le coppie con la stessa parola come chiave vengono raggruppate in multi-insiemi G_k e fornite in ingresso alla fase successiva.

Infine, nella fase di reduce, a tutti i multi-insiemi prodotti dalla fase precedente viene applicata la funzione *reduce*; in questa funzione i valori di tutte le coppie dell'argomento vengono sommati. Viene quindi prodotta una sola coppia in uscita, contente la parola che svolgeva la funzione di chiave comune per il multi-insieme, e la somma di tutti i valori. In pseudocodice la funzione *reduce* diventa:

```
function reduce (Gk) begin
    O ← ∅;
    c ← 0;
    foreach (k, v2) ∈ Gk do
        c ← c + v2;
    O ← {(k, c)};
    return O;
```

Il multi-insieme di output alla fine della fase di reduce è banalmente composto da coppie con chiavi tutti distinte, una per ogni parola presente nel testo. Il valore di ciascuna coppia risulta essere il numero di occorrenze nell'insieme di documenti della parola associata.

1.2 L'Implementazione Originale

All'interno degli articoli dedicati a MapReduce sono presenti alcune informazioni riguardo l'implementazione della libreria all'interno di Google [4] [5]. Il codice però rimane protetto all'interno dell'azienda.

La memorizzazione dei dati in questa implementazione viene gestita dal filesystem distribuito *Google File System* (GFS) [10]. Ciascun file è suddiviso in blocchi, replicati in diverse posizioni nella topologia del cluster. La replicazione serve da meccanismo di prevenzione contro la perdita di dati in caso di guasti ai nodi.

Per quanto riguarda il calcolo, un nodo denominato master funge da scheduler: si occupa di assegnare le singole esecuzioni delle funzioni map e reduce sui nodi secondari detti worker. Il nodo master è un nodo speciale, e viene mantenuto un backup delle sue strutture dati interne. I nodi worker invece sono identici ed intercambiabili, e possono essere aggiunti con facilità; essi svolgono sia funzioni di calcolo che di memorizzazione: il filesystem è co-locato con i nodi di calcolo. Il filesystem inoltra comunica con la parte del framework che si occupa dell'esecuzione del calcolo, cercando di far eseguire le funzioni su macchine del cluster che posseggono una copia locale.

Il sistema punta ad essere resiliente anche in caso di guasti massicci ai nodi worker, rieseguendo le esecuzioni di map e reduce assegnate ai nodi irraggiungibili. L'output delle esecuzioni è atomico: nel caso che più esecuzioni terminino contemporaneamente, solo una scrive il proprio output.

Questa implementazione è progettata per girare su un cluster di server low-end di oltre 1000 unità, e svolge calcoli quali la realizzazione dell'indice per le ricerche, l'elaborazione dati per il sistema AdSense, la risoluzione di problemi di clustering per Google News, e la produzione del reporting sulle ricerche più frequenti (i.e. Google Zeitgeist). In particolare la reimplementazione dell'algoritmo

di indicizzazione per le ricerche in MapReduce ha ridotto il numero di righe di codice da 3800 a 700, rendendo il codice più facile da mantenere ed estendere [4].

1.3 Motivazioni Pratiche

Come già citato più volte, MapReduce parte come risposta ad una esigenza reale; la definizione del paradigma, insieme all'implementazione originale, trova importanti giustificazioni di carattere ingegneristico: non tutte queste considerazioni hanno un effetto diretto sulle specifiche, ma ne offrono il contesto e ne giustificano il successo.

Le argomentazioni riportate in questa sezione fanno in gran parte riferimento all'ottimo (e più completo) lavoro svolto da J. Lin e C. Dyer [18, pag. 8-13].

1.3.1 Parallelizzazione implicita

L'impulso principale nell'ideazione di MapReduce è stato quello di strutturare un sistema semplice su cui sviluppare applicazioni batch parallele [4].

È noto come lo sviluppo di applicazioni parallele sia un problema cognitivamente complesso: le problematiche relative a fenomeni quali deadlocks e race conditions sono intrinsecamente difficili da individuare, isolare e risolvere.

Sebbene esistano molte tecniche e design patterns che possono parzialmente supplire a questa difficoltà, la programmazione distribuita e parallela rimane un compito complesso. A differenza di altri sistemi di calcolo parallelo, in MapReduce lo sviluppatore non è costretto a farsi carico della sincronizzazione e della distribuzione dei dati.

Per questo motivo l'astrazione ricercata è di tipo funzionale: il fatto di forzare la computazione ad essere costruita essenzialmente dall'implementazione di due funzioni, *map* e *reduce*, permette di garantire al sistema la parallelizzabilità del codice utente prodotto. Le singole istanze *map* e *reduce* sono a livello di definizione indipendenti tra loro, e la ovvia dipendenza tra il completamento delle singole fasi sequenziali è demandato al sistema. Le problematiche di programmazione parallela sono quindi concentrate nella implementazione del framework, che può essere ottimizzato e corretto separatamente.

1.3.2 Scalabilità semplice

Nella letteratura riguardante il calcolo parallelo la scalabilità è sempre un obbiettivo primario [15, pag. 26], sia rispetto alla taglia dell'input che rispetto al numero di processori. Con l'astrazione di tipo funzionale presente in MapReduce, l'idea è quella di prescindere dal numero di processori presenti sia nella progettazione che nell'implementazione dell'algoritmo.

Sebbene questo non garantisca scalabilità lineare, permette però di incrementare il numero di processori senza modificare l'implementazione, riducendo i costi ed i tempi necessari per scalare.

La separazione tra l'algoritmo e la piattaforma assume ancora più rilevanza se contestualizzato con la disponibilità di piattaforme di calcolo cloud on-demand. Grazie a questa tecnologia, è possibile aumentare le dimensioni del cluster in relazione al carico o alla complessità dell'analisi da svolgere mantenendo basso il costo.

Sempre le tecnologie cloud permettono di svolgere la fase di test su un piccolo insieme di macchine per poi accedere alla potenza di calcolo solo in fase di produzione ed esclusivamente per il tempo necessario; questa caratteristica è a tutti gli effetti una chiave del successo MapReduce, in quanto ha permesso l'analisi di grandi moli di dati a basso costo.

1.3.3 Elaborazioni batch data-intensive

All'interno di Google esiste una tradizione nell'uso di un approccio data-driven alla risoluzione dei problemi: basato cioè sul principio che un algoritmo semplice è più preciso di un algoritmo complesso se il primo è eseguito su una quantità di dati maggiore [13]. Lo scopo iniziale di MapReduce è proprio questo. Il tipo di calcoli per i quali è stato progettato erano per la maggior parte caratterizzati da algoritmi semplici su grandi quantità di dati.

Alcuni esempi dei problemi che hanno portato alla definizione di MapReduce sono gli indici inversi, la costruzione dei grafi della pagine web, il set delle query più frequenti ed i riassunti della pagine web visitate [4]. Questi problemi non sono di tipo interattivo, ma presuppongono l'estrazione, da un corpus molto vasto, di dati importanti per un utilizzo successivo: si tratta di computazioni di tipo batch, ossia senza vincoli stretti nelle tempistiche di risposta.

Poiché i dataset di questo tipo sono tipicamente troppo grandi per essere contenuti in memoria, e devono quindi essere memorizzati su disco, il design di MapReduce cerca di favorire l'accesso sequenziale rispetto all'accesso casuale ai dati. Trattandosi di computazioni batch, l'idea è di favorire il troughput a scapito della latenza, avvantaggiandosi della banda aggregata dei dischi di un cluster.

1.3.4 Muovere il calcolo verso il dato

Un altro principio del design di MapReduce consiste nel co-locare dati e calcolo. Se in alcuni altri sistemi di calcolo parallelo esistevano nodi di memorizzazione e nodi di calcolo, nella filosofia di MapReduce i nodi del cluster svolgono entrambe le funzioni [4].

Questo in linea di principio dovrebbe permettere di ridurre drasticamente la quantità di comunicazione all'interno del cluster [4]. Essendo infatti le operazioni di Map eseguite sulla singola coppia e tra di loro indipendenti, possono essere eseguite direttamente sul nodo in cui la coppia è memorizzata. Sia nell'implementazione originale [4] che nell'implementazione Hadoop (Sezione 3.3) si presuppone la presenza di un apposito filesystem distribuito che interagisca con il framework propriamente detto, fornendo le informazioni per sfruttare la località dei dati.

1.3.5 Scaling out, not up

Sebbene il modello possa essere utilizzato anche per parallelizzare il lavoro su una singola macchina [4], una delle idee principali che hanno guidato lo sviluppo di MapReduce è la sua implementabilità su server low-end [1].

Per il calcolo su grandi quantità di dati, tenendo presente molti fattori, è più conveniente dal punto di vista economico utilizzare molti nodi low-end piuttosto che pochi nodi high-end, a causa della non linearità del rapporto prezzo/prestazioni [2].

Il design di MapReduce, in cui si cerca di favorire la località dei dati (abbassando quanto possibile il costo di comunicazione dovuto all'aumentare del numero di nodi), cerca di sfruttare questo trend. Un tipico cluster Hadoop varia tra i 10 ed i 100 nodi, fino a toccare vette di 4500 nodi nel caso di alcuni cluster Yahoo!.[2]

[1] Per alcune raccomandazioni riguardo all'hardware vedere [19]

[2] Per maggiori informazioni su alcuni use-case di MapReduce nella sua implementazione Hadoop vedere http://wiki.apache.org/hadoop/PoweredBy

1.3.6 Guasti Frequenti

Ragionando su cluster, i guasti sono inevitabili: in un cluster di medio/grande dimensione, la probabilità che qualche nodo sia irraggiungibile è molto alta, sia per guasti hardware, sia per guasti di connettività o semplicemente per manutenzione. A maggior ragione questo risulta vero utilizzando macchine di classe server low-end, come introdotto nel paragrafo precedente.

Affinché possa essere efficace, un servizio di calcolo distribuito non può prescindere dalla gestione dei guasti. In particolare, occorre fare in modo che questi non impattino sulla qualità del servizio. Nel caso di computazioni di tipo batch, mentre un ritardo limitato può essere ammissibile, sicuramente non lo è la corruzione o la perdita dei dati.

Nella filosofia di MapReduce, l'irraggiungibilità di un numero limitato di nodi prima della fase di esecuzione è facilmente sopportabile a livello di calcolo, grazie all'astrazione tra l'algoritmo e la piattaforma (come spiegato nella Sottosezione 1.3.1). Lo stesso vale per la disponibilità dei dati: l'irraggiungibilità di specifici nodi può essere superata aggiungendo delle repliche dislocate dei dati a livello filesystem distribuito.

In caso di fallimenti durante l'esecuzione di un calcolo, il sistema deve essere però in grado di portarlo a compimento ugualmente. La strategia suggerita in questo contesto nell'articolo originale è la riesecuzione: nel caso un nodo diventi irraggiungibile, le parti di calcolo assegnate a quel nodo (nella forma di esecuzioni di chiamate a map o reduce) vengono riassegnate ad altri nodi [4].

1.4 Critiche: MapReduce e i Database Distribuiti

Una delle principali critiche al modello computazionale viene da una parte della comunità tradizionalmente legata alle basi di dati. In particolare un articolo pubblicato da Andrew Pavlo et Al. [21] pone una forte critica a MapReduce nella sua implementazione Hadoop, comparato a due *database management system* (DBMS) distribuiti.

Nel loro lavoro Pavlo et al. argomentano come dal punto di vista delle perfomance Hadoop risulti sostanzialmente più lento rispetto a due DBMS distribuiti di un fattore compreso tra 2 e 3 nell'esecuzione di alcuni task. MapReduce viene indicato invece come vantaggioso dal punto di vista della facilità di configurazio-

ne, dell'estendibilità e della tolleranza ai guasti. Dal punto di vista della capacità di memorizzazione invece, Hadoop viene indicato come svantaggioso, dovuto al profilo Low-end delle macchine.[3]

I risultati del suddetto articolo sono a loro volta criticate in un articolo di risposta da parte di Dean e Ghemawat [5]. Le considerazioni che vengono fatte al riguardo sono molteplici. Alcuni tipi di computazione MapReduce sono troppo complessi per poter essere espressi puramente mediante linguaggi di query[4] (i.e. SQL). È indubbio però come estendendo quest'ultimi con l'utilizzo delle *User Defined Functions* (UDF) sia possibile esprimere le stesse computazioni esprimibili in MapReduce. Il supporto per le UDF, anche nei DBMS presi in esame nell'articolo di Pavlo et al. risulta però deficitario [5]. Le computazioni su cui viene svolto il confronto tra DBMS e MapReduce sono esprimibili bene in entrambi i paradigmi: l'argomentazione chiave di Dean e Ghemawat è che MapReduce è in grado di svolgere compiti più complessi rispetto a selezione ed aggregazione, punti forte di SQL.

Il problema principale riguarda quindi una incongruenza tra i loro scopi primari: sebbene alcuni compiti possano essere svolti da entrambi, MapReduce eccelle quando si tratta di svolgere lavori specifici con dati non tabulari [3].

Per quanto riguarda le differenza di prestazioni, Dean e Ghemawat sostengono che la differenza sia imputabile ad una immaturità dell'implementazione piuttosto che intrinseca al paradigma; inoltre i risultati non tengono conto del tempo di caricamento necessario per i DBMS. In particolare, il tempo di caricamento necessario è dell'ordine di 50 volte il tempo di esecuzione di Hadoop, che risulta quindi preferibile per analisi poco frequenti [5].

1.5 Altre Implementazioni

Le motivazioni espresse nella Sezione 1.3 non devono essere viste come assolute: esse giustificano il successo delle scelte progettuali di MapReduce nel contesto di Google ma non limitano l'applicabilità del paradigma. Come espresso anche nell'articolo iniziale di Dean e Ghemawat [4] sono chiaramente possibili diverse implementazioni.

[3] L'argomentazione poggia sul fatto che per ottenere 2PB di capacità siano necessari 1000 nodi Hadoop in confronto di 100 nodi per il DBMS Vertica

[4] Per alcuni esempi vedere [5]

Il paradigma computazionale nel suo nucleo è infatti semplice e portabile, e sono disponibili (o in sviluppo) varie implementazioni con diverse piattaforme come target.

La prima famiglia di implementazioni rimane quella dedicata ai cluster di server low-end, seguendo la traccia dell'implementazione originale proposta da Google; di questa, l'esponente principale è senza dubbio *Hadoop*, che verrà trattato in modo estensivo nel Capitolo 3. Altre implementazioni di questa famiglia sono anche Dryad/DryadLINQ [30], realizzata da Microsoft, e CGL-MapReduce [7], basata su un approccio di tipo streaming ed orientata al calcolo scientifico. Particolarmente interessante per il tipo di problemi che vengono trattati nei prossimi capitoli è l'implementazione *Twister* [6], proposta dall'università dell'Indiana ed orientata al calcolo iterativo di MapReduce.

Tra le altre implementazioni disponibili vale la pena menzionare *Phoenix* [20], orientata verso il calcolo su macchine multiprocessore *Non-Uniform Memory Access*(NUMA), *Mars* [14] concepita per il calcolo su *Graphic Processing Unit* (GPU), e *CellMR* [23] pensata per un cluster asimmetrico misto di CPU general purpose e processori Cell.[5]

In ultimo, è interessante far notare una piccola implementazione di MapReduce, orientata a piccoli cluster, costituita da bash script e UNIX tools [9], che ribadisce l'applicabilità e portabilità del paradigma.

[5] I processori Cell hanno una architettura dedicata specializzata per il calcolo. Un esempio commerciale altamente disponibile di nodo basato su processori Cell è Sony Playstation 3

Capitolo 2

Il Modello

Come già citato, MapReduce risulta sprovvisto di un modello formale universalmente accettato. Nell'articolo in cui viene introdotto [4] non viene dato nessun modello teorico, focalizzando invece l'attenzione sulla componente applicativa.

Un buon modello per MapReduce ha un duplice compito: da una parte catturare l'essenza del paradigma, sia in termini di espressività che di analisi prestazionale, dall'altra astrarre dalla realtà molti dei dettagli implementativi presenti nella definizione iniziale.

La principale caratteristica del paradigma MapReduce, rispetto ad altri paradigmi di calcolo parallelo per i quali esistono modelli, è di essere composto da fasi sequenziali distinte: map, shuffle, e reduce. Durante l'analisi teorica risulta inoltre abbastanza immediato rilevare come una sequenza di queste tre fasi non sia sufficiente ad esprimere algoritmi complessi. A livello di modello si passa quindi a considerare un algoritmo MapReduce come una sequenza ripetuta di queste fasi. Per chiarezza si introduce il termine *round*:

Definizione 2.1 (Round)**.** *Con il termine round si intende una computazione MapReduce, ossia composta (dal punto di vista del paradigma) da una fase di map, una fase di shuffle ed una fase di reduce.*

Un algoritmo in MapReduce sarà quindi visto come una sequenza di round. È importante notare come, a livello paradigmatico, questo concetto non emerga. Il tipo di problemi computazioni per i quali MapReduce è stato sviluppato originariamente potevano essere risolti con algoritmo in un solo round oppure in un numero costante di round (vedi Sottosezione 1.3.3 a pag.15).

L'obbiettivo di questo capitolo è di introdurre infine il modello di riferimento per questa Tesi, denominato *MR(m,M)* sviluppato all'interno dell'Università di Padova in collaborazione con la Brown University [22]. Nella Sezione 2.3 sono riassunti alcuni dei lavori precedenti in questo campo.

2.1 Il Problema della Nomenclatura: Mapper e Reducer

Anche in questo capitolo ritorna il problema della nomenclatura già affrontato in precedenza. Non esiste una terminologia univoca e condivisa che permetta di evitare confusioni tra i concetti relativi al paradigma, ai modelli o alle implementazioni.

La nomenclatura utilizzata nei lavori citati verrà quindi presentata modificata in questa Tesi, tentando di produrre un corpus omogeneo e chiaro.

Una particolare fonte di ambiguità riguarda l'uso dei termini *mapper* e *reducer*. In alcune fonti essi hanno significato analogo ai termini *funzione map* e *funzione reduce*, in altri indicano una specifica macchina, in altri una singola entità parallelizzabile funzionale (i.e. una chiamata alla funzione map o reduce). Poiché questi termini hanno un significato preciso nella terminologia implementativa (ed in particolare nell'implementazione di riferimento, Hadoop) essi sono stati sostituiti nella modellistica da termini più precisi, dipendentemente dal contesto.

2.2 Il Modello di Riferimento: MR(m,M)

Il modello utilizzato all'interno di questa Tesi cerca di catturare l'essenza funzionale di MapReduce, permettendo un uso flessibile del parallelismo.

A differenza di altri modelli, MR(m,M) non prevede di trattare esplicitamente il numero di processori su cui l'algoritmo viene eseguito, semplificando il design degli algoritmi in accordo con lo spirito originale di MapReduce.

2.2.1 Eliminazione della fase di Map

Come già evidenziato in lavori precedenti [17], a livello astratto la fase di map in un algoritmo multi-round può essere soppressa definendo una nuova funzione

reduce per ogni round che aggreghi la funzione *map* del round successivo.

Presa una sequenza di R round MapReduce, per il round j-esimo con $1 \leq j \leq R-1$ la nuova funzione *reduce* corrisponde alla funzione *reduce* originale al round j al cui output è applicata, coppia per coppia, la funzione *map* del round $j+1$. È importante notare come la nuova funzione sia a tutti gli effetti una funzione *reduce* valida secondo il paradigma, ma su multi-insiemi diversi: l'insieme di output del round j dell'algoritmo definito con la nuova funzione corrisponde all'insieme intermedio del round $j+1$ dell'algoritmo definito con la funzione originale.

Rimangono da considerare i casi limite corrispondenti al primo ed all'ultimo round. La fase di map del primo round può essere facilmente realizzata con una fase reduce in cui la funzione *map* sia applicata a ciascun elemento in ingresso. L'ultimo round può invece essere realizzato applicando semplicemente la funzione *reduce*originale.

In questo modo è possibile considerare una sequenza di round di fasi map, shuffle e reduce come una sequenza di fasi shuffle e reduce. Questo approccio è quello seguito nella realizzazione del modello MR(m,M).

2.2.2 Definizione di Algoritmo

Introduciamo la simbologia specifica del modello definendo la nozione di algoritmo in MR(m, M).

Definizione 2.2 (MR-Algoritmo). *Si dice MR-Algortimo una sequenza di round tali che l'r-esimo round con $r \geq 1$ trasforma il multi-insieme di coppie chiave-valore W_r in due multi-insiemi*

- W_{r+1} *che funge da input per il round successivo;*
- O_r *un (multi-)sottoinsieme dell'output finale, eventualmente vuoto.*

L'universo delle chiavi delle coppie in W_r è indicato con U_r.

La computazione al round r-esimo è definita dalla sola funzione reduce (vedi Sezione 2.2.1) indicata con ρ_r, che viene applicata indipendentemente ad ogni multi-insieme $W_{r,k} \subset W_r$ composto da tutte le coppie con la stessa chiave $k \in U_r$.

L'input di un MR-Algoritmo è formato dal multi-insieme W_1, mentre il suo output è formato da $\bigcup_{r\geq 1} O_r$ dove con $\bigcup$ è indicata l'unione tra multi-insiemi.

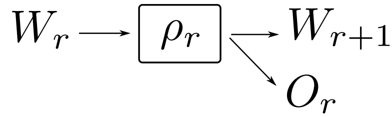


Figura 2.1 – Schema di un r-esimo round in MR(m,M). La computazione è definita in termini della sola funzione reduce.

La struttura di un round in MR(m,M) è schematizzata in Figura 2.1. Il multi-insieme O_r che contribuisce direttamente all'output rappresenta una estensione del paradigma. Nel paradigma MapReduce infatti non è prevista una distinzione esplicita dell'output, sebbene questa possa essere facilmente definita modificando la definizione delle funzioni. Questa estensione però modella accuratamente una situazione reale frequente: una parte dell'output di un round viene separata e conservata separatamente (i.e. su un altro file) rispetto al resto dell'output del round. Questo tipo di *side-effect* è riportato come una estensione di MapReduce anche nell'articolo di presentazione di MapReduce tra i casi d'uso presenti in Google [4].

2.2.3 Vincoli

Il modello è definito in funzione di due parametri interi, m e M, che rappresentano dei vincoli sulla memoria; si impone infatti che la memoria locale utilizzata per ogni invocazione della funzione *reduce* ρ_r sia lineare rispetto al parametro m. La memoria occupata da tutte le invocazioni in un singolo round si impone invece sia lineare rispetto a M, così come la memoria utilizzata per l'output finale in tutti i round. Formalmente si ha quindi che:

Vincolo 1 (Vincoli di memoria)**.** *Sia n la taglia dell'input, si indica con $m_{r,k}$ lo spazio necessario per calcolare la funzione reduce $\rho_r(W_{r,k})$ in una macchina RAM con parole di memoria lunghe $O(\log_n)$ bit, definito in modo che*

- *sia incluso lo spazio dell'input e lo spazio di lavoro (i.e. $m_{r,k} \geq |W_{r,k}|$)*
- *sia escluso lo spazio dell'output, sia che contribuisca ad O_r che a W_{r+1}*

Nel modello MR(m,M) si impone che

$$m_{r,k} \in O(m) \qquad \forall r \geq 1,\ \forall k \in U_r \tag{2.1}$$

$$\sum_{k \in U_r} m_{r,k} \in O(M) \qquad \forall r \geq 1 \tag{2.2}$$

$$\sum_{r \geq 1} |O_r| \in O(M) \tag{2.3}$$

Il vincolo espresso nella formula 2.1 riguarda la memoria locale di una singola esecuzione, mentre la formula 2.2 esprime un vincolo sulla memoria aggregata del cluster per ogni round. La terza formula (2.3) impone invece un vincolo sull'output globale dell'algoritmo.

Non viene imposto nessun vincolo temporale esplicito per la funzione *reduce* nei vari round. La scelta in MR(m,M) è quella di limitarne semplicemente la complessità affinché il tempo di esecuzione di una singola funzione non diventi dominante. Formalmente questo vincolo diviene:

Vincolo 2 (Complessità Polinomiale). *La complessità in termini di modello RAM della funzione reduce deve essere polinomiale in n, dove n è la taglia dell'input.*

2.2.4 Modello di costo

La principale assunzione di MR(m,M) in questo campo riguarda la fase di shuffle. Si considera questa fase molto costosa in termini di risorse e di tempo, in quanto in essa è concentrata tutta la comunicazione tra i nodi. Poiché questa fase è inevitabile e fuori dal controllo dello specifico algoritmo, l'obbiettivo è minimizzare il numero di fasi di shuffle (e quindi di round) nell'ipotesi che questo minimizzi il tempo globale necessario ad eseguire l'algoritmo. Questo si riflette nella definizione della complessità per un MR-algoritmo:

Definizione 2.3 (Complessità). *La complessità di un MR-algoritmo è il numero di round che esso svolge al caso pessimo.*

L'idea alla base del modello è quella di esporre un *tradeoff* tra i parametri riguardanti la memoria (i.e. M e m) ed il numero di round necessari per completare l'algoritmo.

2.3 Lavori precedenti

In letteratura sono presenti alcuni modelli per MapReduce, con caratteristiche differenti. Qui sono elencati e brevemente confrontati con il modello di riferimento, MR(m,M).

Un modello è stato proposto da Karloff et al. [16]. Detta n la taglia dell'input ed ϵ una costante in $(0, 1)$, gli autori impongono in questo modello che ogni chiamata alla funzione *reduce* elabori un multi-insieme di taglia $O(n^{1-\epsilon})$ mentre la memoria globale dev'essere di taglia $O(n^{2-2\epsilon})$. Il costo delle computazioni all'interno delle funzioni è nascosto, mentre la complessità è limitata ad essere polinomiale in n. Viene inoltre assunto che l'infrastruttura consista in $\Theta(n^{1-\epsilon})$ macchine con memoria locale $\Theta(n^{1-\epsilon})$.

Il modello MR(m,M) è più generale al modello di Karloff. Imposti infatti i parametri m e M come $m \in O(n^{1-\epsilon})$ e $M \in O(n^{2-2\epsilon})$ il modello di Karloff si riconduce ad un caso particolare di MR(m,M). In più quest'ultimo evita di esprimere le caratteristiche del cluster nel modello, rendendolo più simile ai principi del paradigma.

Un secondo modello è introdotto da Goodrich et al. [11]. La sua caratteristica principale è quella di utilizzare un modello di costo complesso che tiene in considerazione anche la comunicazione. Questo modello di costo è simile a quello presente nel modello BSP [28]. Se confrontato con il modello di Goodrich, MR(m,M) introduce il vincolo di memoria globale, che in quel modello non era espresso. L'analisi in MR però si focalizza solo sul numero di round, a differenza della complessa funzione di costo del modello di Goodrich che tiene in conto la complessità delle specifiche funzioni, la latenza e la banda della piattaforma.

Esiste infine un terzo modello orientato allo streaming, introdotto da Feldman et al. [8]. Il questo modello, detto MUD, cerca di ricongiungere lo spirito di MapReduce con la tradizione relativa ai Data Stream. Le funzioni in MUD processano le coppie chiave-valore in un solo passaggio utilizzando memoria poli-logaritmica rispetto alla taglia dell'input. Questo modello ha uno scopo diverso rispetto a MR(m,M) e ne differisce in modo sostanziale.

Il termine Reducer

Come introdotto in Sezione A, è interessante notare a livello di nomenclatura come in ognuno di questi modelli il termine *reducer* abbia un'accezione diversa. In particolare, nel modello di Goodrich, un Reducer viene trattato come una macchina fisica, con memoria, che conserva lo stato tra una iterazione e la successiva. Nel lavoro di Karloff invece un Reducer viene definito come una funzione, ma utilizzato anche come istanza della stessa.

Capitolo 3

Apache Hadoop

> The name my kid gave a stuffed yellow elephant. Short, realtively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid's term.
>
> Dough Cutting, sull'origine del nome Hadoop [29]

L'implementazione più diffusa per numero di installazioni di MapReduce è senza dubbio Apache Hadoop. Scritto per la maggior parte in Java, esso rappresenta al giorno d'oggi l'unica implementazione del paradigma disponibile al pubblico con molti casi d'uso concreti; per questo motivo è stata scelta quale piattaforma su cui provare a validare il modello MR(m,M).

La letteratura tecnica su Hadoop è abbastanza corposa, ed esistono varie possibili fonti di documentazione disponibili, anche se spesso sono incomplete o parziali. In questo capitolo si cercherà di descrivere sinteticamente la struttura di Hadoop al fine di delinearne le caratteristiche principali. Particolare attenzione sarà riservata quindi ad alcune caratteristiche che sono state utilizzate nella parte sperimentale.

Nell'Appendice A è disponibile una lista di consultazione rapida con i principali termini relativi ad Hadoop.

3.1 Introduzione

3.1.1 Breve storia di Hadoop

Hadoop nasce all'interno di un sotto-progetto di Apache Lucene: Nutch, un motore di ricerca Open Source. Il suo autore è il creatore del progetto Lucene stesso, Dough Cutting.

La prima versione di Nutch, sviluppata nel 2002, aveva però problemi di scalabilità: sebbene il crawler ed il generatore di indici funzionassero bene per piccole quantità di dati, non riuscivano a gestire i dati relativi all'intera struttura del Web. Quando nel 2003 fu pubblicato l'articolo sul *Google File System* (GFS) [10], il team di sviluppo di Nutch realizzò come una soluzione simile potesse risolvere parte dei problemi che stavano incontrando. Nel 2004 iniziò quindi una re-implementazione Open Source di GFS, chiamata inizialmente *Nutch Distributed File System* (NDFS); questa implementazione diventerà poi il filesystem di Hadoop. Quando in seguito Dean e Ghemawat pubblicarono il primo articolo su MapReduce [4] sembrò naturale proseguire su questa strada, creando una versione Open Source anche di quest'ultimo, da affiancare a NDFS.

Fu ben presto chiaro che l'implementazione per Nutch di MapReduce era applicabile fuori dai confini del motore di ricerca; per questo motivo nel 2006 il codice relativo fu spostato in un nuovo sotto-progetto di Lucene: Hadoop.

Contemporaneamente Yahoo! stava ristrutturando il sistema di generazione degli indici per il suo motore di ricerca: il sistema in sviluppo condivideva alcuni dei concetti alla base di MapReduce, ma era meno maturo rispetto ad Hadoop. Dough Cutting fu quindi assunto da Yahoo!, che contemporaneamente fornì un team di sviluppo dedicato e le risorse necessarie a rendere Hadoop sufficientemente scalabile per affrontare la complessità del World Wide Web. Nel 2008 Yahoo! dichiarerà ufficialmente che il suo indice per le ricerche in produzione è generato da un cluster Hadoop composto da 10.000 core.

Nel Febbraio 2008, Hadoop venne separato da Lucene, formando un proprio progetto indipendente all'interno della fondazione Apache, e venne adottato in produzione da altre imprese commerciali quali Last.fm, Facebook ed il New York Times [29, p.9].

Nell'Aprile 2008, Hadoop diventò il sistema più veloce per ordinare un TeraByte di dati grezzi: 209 secondi, usando un cluster da 910 nodi. Attualmente

(2011), detiene il record per la maggior quantità di dati ordinata in un minuto usando un algoritmo di ordinamento *general purpose* [12].

Yahoo! ha contribuito o sviluppato circa l'80% del codice di Hadoop [24].

3.1.2 Versioni disponibili

Hadoop viene sviluppato all'interno del progetto Apache, ma ne esistono anche alcune distribuzioni alternative.

La fondazione Apache sviluppa hadoop in 3 sottoprogetti: hadoop-common, hadoop-hdfs e hadoop-mapreduce. Allo stato attuale i tre progetti condividono la base di codice ed il numero di versione. Durante la scrittura della Tesi è stata rilasciata una versione alpha (0.23) che risulta particolarmente interessante in quanto integra alcune importanti novità strutturali. In particolare viene proposta una nuova implementazione del codice relativo ai demoni MapReduce chiamata NextGen MapReduce o YARN. Questa implementazione aggiunge nuova flessibilità al framework, sopperendo a molte delle limitazioni emerse durante l'analisi sperimentale. Inoltre la versione 0.23 fornisce anche nuovi strumenti di monitoraggio integrato che potrebbero essere sfruttati con successo per espandere il lavoro di questa Tesi; la mancanza di documentazione e lo stato "alpha" ne sconsigliano però attualmente l'utilizzo.

La principale distribuzione alternativa è fornita da Cloudera, una società che si occupa di fornire supporto commerciale per Hadoop. Cloudera fornisce Hadoop pacchettizzato, aumentandone la facilità di installazione e manutenzione tramite una serie di script. È da notare però che l'attuale versione distribuita (0.20.203-cdh3), sebbene basata sulla versione stabile distribuita da Apache, ha applicate alcune patch che la rendono parzialmente incompatibile con la versione originale.

La versione utilizzata nella parte sperimentale della Tesi è la versione stabile 0.20.203 vanilla (i.e. senza patch) prelevata direttamente dai repository Apache.

3.1.3 L'ecosistema Hadoop

Il progetto Hadoop ha poi una discreta quantità di progetti di supporto, utilizzati per fornire funzionalità aggiuntive o livelli di astrazione superiori rispetto a MapReduce. Nessuno di questi progetti è stato utilizzato durante la Tesi, ma

vengono citati in quanto forniscono possibilità di estensione per questo lavoro. Si segnalano in particolare:

- *Pig*, un linguaggio di alto livello completo di interprete interattivo che viene tradotto in termini MapReduce ed eseguito su Hadoop;
- *Hbase*, database noSQL distribuito basato su Hadoop e HDFS, simile a BigTable di Google;
- *Mohaut*, libreria Hadoop per l'intelligenza artificiale ed il datamining;
- *Chukwa*, strumento per l'analisi di log distribuita utilizzando Hadoop e HBase. Può essere utilizzata per analizzare i log di Hadoop;
- *Hive*, sistema per il data-warehousing su Hadoop. Permette di esprimere le query in termini SQL-like o MapReduce.

3.2 Introduzione ad Hadoop

Nella terminologia di Hadoop il concetto di round si traduce con il termine *job*. Come spesso avviene tra implementazione e modello, esiste una piccola distanza tra i due concetti, che diverrà più chiara con il proseguire del Capitolo. Per il momento, questa uguaglianza renderà più comprensibile la descrizione.

Hadoop è formato da due parti principali: la parte che si occupa della schedulazione ed esecuzione dei calcoli, denominata MapReduce[1], ed una parte che si occupa della gestione del filesystem distribuito, chiamata *Hadoop Distributed File System*(HDFS).

Questa seconda componente è necessaria per eseguire MapReduce: HDFS e MapReduce sono progettati per lavorare in coppia. La parte HDFS comunica alla parte MapReduce la posizione dei dati nel cluster, al fine di ottimizzare la computazione. MapReduce può comunque sfruttare altre sorgenti di dati, quali Amazon S3, HBase o altri database distribuiti, ma al momento necessita comunque di HDFS per svolgere alcuni compiti di supporto (ad es. la distribuzione del codice utente da eseguire).

[1] per chiarezza, nei casi in cui possano sorgere confusioni con il paradigma generale, ci riferirà alla componente di Hadoop come *Hadoop MapReduce*.

3.2.1 Tipi di nodi

Hadoop, come l'implementazione MapReduce di Google, è progettata per girare su un cluster di server low-end. Tra gli obbiettivi di questa scelta progettuale c'è quello di facilitare la sostituzione dei nodi in caso di guasti, come discusso nel Capitolo 1.

In Hadoop non tutti i nodi sono intercambiabili; l'architettura generale infatti prevede attualmente alcuni servizi centralizzati, come lo spazio dei nomi e lo scheduling dei job. Esistono quindi due tipi di nodi:

- nodi *master*, dove vengono eseguiti i demoni di coordinamento per HDFS e MapReduce. La scalabilità del cluster è legata alle risorse disponibili su queste macchine, in termini di CPU, memoria RAM e spazio sul disco. Nel caso di cluster di grandi dimensioni, questi demoni sono eseguiti su delle macchine dedicate e sottoposte a politiche di backup e/o *High Availability*.

- nodi *worker*, dove vengono co-locati memorizzazione e calcolo. In queste macchine sono eseguiti i due demoni di servizio per MapReduce e HDFS.

In caso di cluster di dimensione limitata i due servizi di coordinamento possono essere eseguiti sulla stessa macchina, ed in caso di cluster molto piccoli (intorno alla decina di macchine) questa macchina può funzionare anche da worker.

3.3 HDFS: il FileSystem Distribuito

Il filesystem distribuito di Hadoop, HDFS, è un filesystem particolare, specializzato, il cui scopo primario è di contenere i dati di input ed output per MapReduce. HDFS è strutturato sul modello di un filesystem POSIX, ma sacrifica l'aderenza allo standard per migliorare le prestazioni se utilizzato in coppia con Hadoop MapReduce. In particolare le caratteristiche salienti sono:

- Supporto di file di grandi dimensioni: un tipico file su HDFS ha dimensione compresa tra le centinaia di MegaByte e qualche TeraByte. Non esiste un limite esplicito alla dimensione dei file;

- Accesso Sequenziale: supporta un paradigma write-one, read-many-times. Il troughput è ottimizzato a spese della latenza;

- Utilizzo di nodi *commodity*: non prevede l'uso di dispositivi RAID. La tolleranza ai guasti viene fornita tramite replicazione sui nodi.

È da notare quindi che HDFS risulta inadatto per applicazioni che richiedano l'accesso ai dati con bassa latenza, o la scrittura di un singolo file da parte di più programmi. Nello stesso modo risulta problematica la gestione di un gran numero di piccoli file.

La principale caratteristica che rende invece adatto HDFS a supportare l'esecuzione di MapReduce è data dalla gestione della replicazione, che verrà affrontata in Sezione 3.3.2.

Nella parte rimanente di questa sezione si cercherà fornire le informazioni essenziali riguardo alla struttura ed al comportamento di HDFS. Ulteriori dettagli possono essere recuperati su altri articoli specializzati [24] [25] [26].

3.3.1 Infrastruttura

Il filesystem è costituito da alcuni demoni principali, datanode e namenode, e da altri demoni secondari che svolgono servizi di supporto. Le entità che interagiscono con HDFS in lettura e scrittura prendono invece semplicemente il nome di *client*. La struttura di un cluster HDFS è illustrata in Figura 3.1.

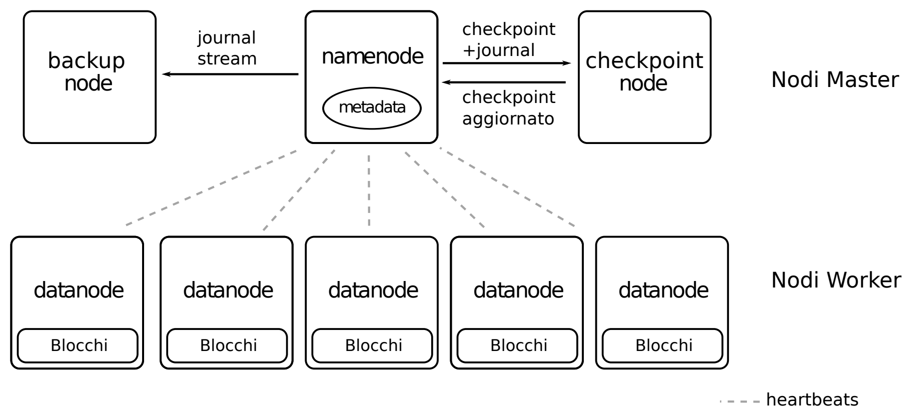


Figura 3.1 – Struttura di HDFS, in cui sono indicati i vari demoni. Il namenode mantiene in RAM lo spazio dei nomi, mentre i datanode si occupano di memorizzare i blocchi sotto forma di file nel filesystem locale. Backup node replica in metadati, mentre checkpoint node unisce journal e checkpoint ad intervalli regolari. [24]

datanode

Il primo tipo di demone, *datanode*, è il demone eseguito sui worker. Come suggerisce il nome, esso è il demone di memorizzazione di Hadoop, e mantiene i blocchi di dati. Un blocco ha una dimensione configurabile, normalmente compresa tra i 64Mb ed i 128Mb. I singoli blocchi sono mappati sotto forma di file regolari nel filesystem locale del datanode. In caso di blocchi parzialmente utilizzati, solo lo spazio necessario viene occupato. A livello locale un datanode può utilizzare più di un disco a questo scopo. Poiché la tolleranza ai guasti è garantita da HDFS direttamente, nelle specifiche si consiglia di utilizzare i dischi in modalità *Just a bunch of disks*(JBOD).

namenode

Il secondo demone, *namenode*, è eseguito in singola copia all'interno del cluster su un nodo master. Esso funge da spazio dei nomi e tabella di allocazione per il filesystem; in aggiunta coordina la memorizzazione, garantendo che il fattore di replicazione sia mantenuto. Per ogni blocco viene memorizzata una lista di datanode che ne possiedono una copia, ed i metadati relativi, come il file di appartenenza. I metadati sono mantenuti in RAM, in modo da velocizzarne l'accesso; per migliorare l'affidabilità, namenode mantiene anche una versione iniziale statica dei metadati su disco detta *checkpoint*, e un *journal*, con le modifiche differenziali. L'unione tra journal e checkpoint viene effettuato solo al riavvio, aggiornando il checkpoint.

I datanode comunicano con il namenode tramite *heartbeats*, di default uno ogni 3 secondi. Gli heartbeats hanno un duplice scopo: da una parte di segnalare l'attività e lo stato "live" dei datanode e di riportane le statistiche (spazio totale, spazio disponibile etc.), dall'altra di trasportare nelle risposte le istruzioni del namenode riguardanti le operazioni da svolgere, come ad esempio la rimozione di blocchi o la replicazione su altri nodi.

La scelta di separare dati e metadati permette di modificare velocemente quest'ultimi, garantendo al contempo di sfruttare durante l'accesso ai dati la banda combinata di tutti i dischi del cluster.

Altri demoni

Oltre ai demoni principali, in HDFS possono essere presenti altri demoni, con ruoli differenti. Il *checkpoint node* (detto anche *secondary namenode*), normalmente eseguito su un nodo master separato (in quanto ha le stesse richieste di memoria del namenode), si occupa di recuperare journal e checkpoint dal namenode e di effettuarne l'unione. La nuova versione viene quindi reinviata al namenode. È utilizzato per cluster che rimangano in funzione per lunghi periodi: questa operazione viene normalmente svolta una volta al giorno. Recentemente è stato introdotto il *backup node*, che mantiene una copia read-only sincronizzata in tempo reale del namenode. Può essere utilizzato per creare i checkpoint al posto del checkpoint node.

3.3.2 Il Ruolo della Replicazione

La replicazione ha un ruolo importante in HDFS. Per prima cosa funge da sistema di prevenzione contro la perdita di dati in caso di guasti. Rispetto ad altre tecniche più complesse che garantiscono la tolleranza (ad esempio block striping o erasure codes) non richiede calcoli aggiuntivi per ricostruire eventuali blocchi persi [25]. Nel caso un datanode non invii heartbeats per un periodo prolungato, il namenode ripristina il livello di replicazione schedulando una nuova copia su un altro datanode.

In aggiunta la replicazione permette l'accesso parallelo e distribuito ai dati durante situazioni di alto carico. A differenza di altri filesystem di questo tipo, HDFS rende disponibile tramite API la locazione dei blocchi, permettendo ai client di accedere direttamente ai datanode per la lettura. Questa caratteristica è ampiamente sfruttata da MapReduce durante la fase di map per co-locare dati e calcolo, riducendo il consumo di banda.

Il fattore di replicazione può essere impostato sia a livello globale che manualmente per ogni singolo file. Di default è impostato al valore 3, che sembra essere la scelta conveniente per la maggior parte delle applicazioni, anche su cluster di grandi dimensioni [25]. Poiché la replicazione è impostabile dai client, a livello di API, HDFS dichiara lo spazio disponibile senza tenere conto della replicazione.

Le repliche sono distribuite in modo da minimizzare la possibile perdita in caso di guasti alla connettività. HDFS può essere informato della struttura del cluster a livello di rack: se questo avviene, le copie vengono distribuite sia in-rack che extra-rack, cercando un tradeoff tra la possibilità di guasti e la banda tra i rack. Invece HDFS non distribuisce le copie in funzione dello spazio disponibile sul singolo datanode: il datanode più prossimo al client sarà sempre destinatario di una copia del blocco. Questo può portare ad un rallentamento della lettura limitandone il parallelismo possibile. Per ovviare a questo problema si può effettuare un ri-bilanciamento del carico, utilizzando l'apposito script fornito con Hadoop. [2]

[2] Lo script nella versione 0.20.203 è $HADOOP_HOME/bin/start-balancer.sh

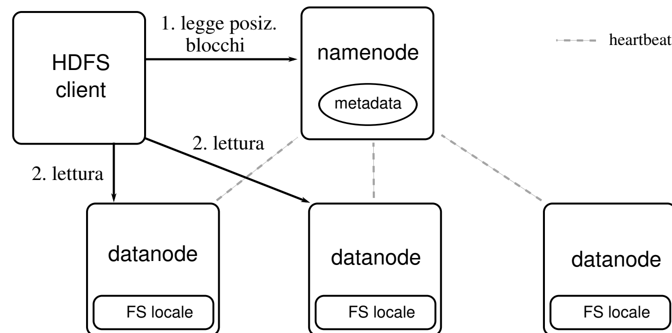


Figura 3.2 – Schema lettura da HDFS [29]

3.3.3 Anatomia delle operazioni principali

Lettura

Uno schema di una operazione di lettura è presentato nella Figura 3.2. Per eseguite una operazione di lettura, un client HDFS contatta il namenode, che ritorna i datanode che possiedono una copia dei primi blocchi. Le operazioni di lettura sono effettuate direttamente sui datanode, interrogandoli singolarmente. I datanode vengono forniti ordinati in base alla distanza dal client: se il client è eseguito su un nodo del cluster (come ad esempio vedremo succederà in una esecuzione di MapReduce su un worker) e possiede una copia del blocco, leggerà la sua copia locale.

Man mano che la lettura del file procede, il client richiede al namenode la locazione dei blocchi successivi. L'apertura delle connessioni viene gestita in modo trasparente dal framework, così come la lettura dei vari blocchi: in uscita si osserva un flusso continuo di dati.

Da notare che, poiché il namenode si limita fornire le locazioni dei blocchi, un elevato numero di client concorrenti può accedere ai dati.

Scrittura

Uno schema delle operazioni di scrittura è presentato in Figura 3.3. Per effettuare una scrittura, un client HDFS comunica la creazione di un file al namenode, il quale controlla la non-esistenza del file ed i permessi di accesso. Se la scrittura viene autorizzata, il namenode crea un record sui metadati per il file. Il namenode alloca quindi i blocchi, fornendo al client una lista di possibili datanode su cui

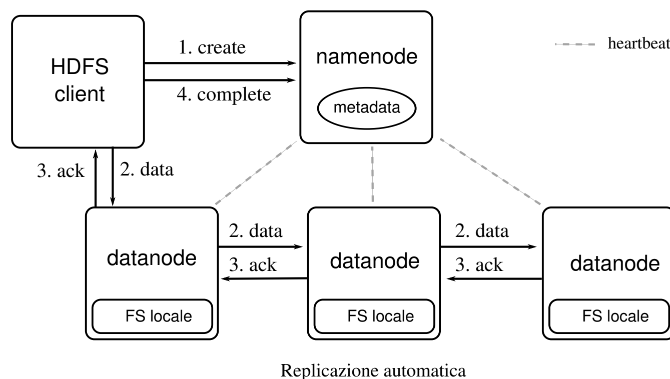


Figura 3.3 – Schema scrittura su HDFS [29]

posizionarli; la taglia di questa lista è pari al livello di replicazione desiderato. La lista viene quindi utilizzata come una *pipeline* lungo la quale inviare i dati di ciascun blocco. Il primo datanode memorizza i dati ricevuti dal client, li salva sul filesystem locale (nel file locale associato al blocco HDFS) e re-invia i dati al secondo datanode. L'operazione viene ripetuta fino alla fine della lista. Le conferme di scrittura (*ack*) percorrono contemporaneamente la pipeline in senso contrario.

Nel caso che un datanode fallisca durante la scrittura, il framework reagisce in modo trasparente rispetto al client. La pipeline viene interrotta al livello del nodo mancante. Il blocco parzialmente scritto cambia poi identità sui nodi attivi: in questo modo se il nodo mancante dovesse tornare online, il blocco parziale verrebbe eliminato in quanto non più allocato sul namenode. Questa modifica viene comunicata al namenode via heartbeats. I pacchetti non confermati sull'ultimo datanode prima del guasto vengono quindi re-inviati ai nodi a valle del nodo mancante e la pipeline riaperta senza il nodo in questione. Il namenode quindi individuerà via hearbeats il blocco come sotto-replicato, e schedulerà una nuova copia su un datanode. Il client nel frattempo continua la scrittura senza interruzioni. Nel caso di fallimenti multipli il comportamento è simile. La scrittura ha successo se il numero di repliche effettuate è pari ad un valore minimo, configurabile a livello di file system.

Quando ha finito di inviare i dati nella pipeline, il client contatta il namenode comunicando la fine della scrittura. Il namenode aspetta a notificarne il successo fino a quando i datanode hanno comunicato via heartbeats che tutti i blocchi di cui il file è composto hanno raggiunto il numero minimo di repliche.

3.4 MapReduce: l'Infrastruttura di Calcolo

La parte del framework che si occupa del calcolo distribuito è costituita sia dai demoni, che si occupano dell'effettiva esecuzione, sia che da un'ampia libreria, utilizzata per scrivere il codice eseguibile.

Come abbiamo introdotto precedentemente, Hadoop traduce il concetto di round con il termine job, a meno di alcune differenze. Per Hadoop un job è essenzialmente un oggetto di configurazione che, una volta preso in carico dai demoni MapReduce, viene schedulato e tradotto in una esecuzione equivalente ad un round ma diversamente strutturata.

Anche l'infrastruttura di MapReduce, così come HDFS, è scritta in Java, ma integra la possibilità di eseguire job scritti in altri linguaggi tramite due modalità aggiuntive denominate *streaming* e *pipes*. Queste due modalità prevedono l'implementazione delle funzioni *map* e *reduce* in eseguibili esterni, comunicando con essi tramite rispettivamente lo standard output e le librerie wrapper. Nel proseguire di questo Capitolo, per semplicità di trattazione, verrà discussa solo l'implementazione in Java dei job, in quanto utilizzata nella parte sperimentale.

3.4.1 Infrastruttura

A livello di infrastruttura, la parte di Hadoop dedicata al calcolo è costituita da due demoni, detti *jobtracker* e *tasktracker* e dal programma utente, che viene denominato *driver*.

jobtracker

Il *jobtracker* viene eseguito su un nodo master e ha funzione di coordinamento nelle esecuzioni MapReduce. Esso assegna in primo luogo le computazioni ai nodi, comunicando con HDFS per sfruttare la co-locazione tra calcolo e dati. Nel caso vi siano più job concorrenti, il jobtracker procede a schedulare in modo opportuno le computazioni. Inoltre questo demone monitora lo stato ed i progressi dei vari *tasktracker*, comunicandoli al driver che ha inviato il job. In caso di problemi (ad esempio nodi irraggiunbili, errori di calcolo o mancanza di progressi) prende le azioni opportune, rieseguendo parti di calcolo o dichiarandone il fallimento.

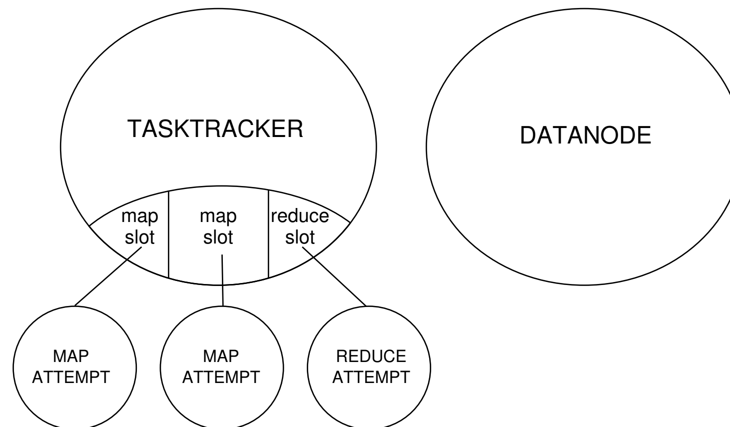


Figura 3.4 – Struttura di un nodo worker in Hadoop. Ogni cerchio rappresenta una JVM separata. Sono presenti i due demoni, tasktracker per MapReduce e datanode per HDFS, e le istanze di calcolo. Il nodo in questione ha 2 map slot e 1 reduce slot.

tasktracker

In un cluster Hadoop vengono eseguite molte copie del demone tasktracker, una per ogni nodo worker. La funzione di questo demone è quella di eseguire i calcoli veri e propri. Nella configurazione di ogni tasktracker è specificato il numero di processi di calcolo che può eseguire. Questi processi potenziali sono detti *task slot*, o più semplicemente *slot*. Nella versione di Hadoop corrente essi sono specifici per tipo di calcolo: si distinguono quindi in *map slot* e *reduce slot*. Ciascuna istanza di calcolo, detta *task attempt* viene eseguita in una JVM separata rispetto al demone, per garantire separazione e tolleranza ad eventuali errori nel codice del job. In Figura 3.4 è schematizzato la struttura di un worker node, dove sono messi in evidenza le varie JVM.

Similarmente a quanto accade in HDFS, i tasktracker comunicano con il jobtracker per mezzo di *heartbeats* periodici, nelle cui conferme il jobtracker assegna l'esecuzione dei calcoli.

driver

Con il termine *driver* in Hadoop si intende qualsiasi programma utente che si occupa di inviare dei job ad Hadoop. Un driver nella sua accezione base consiste in un eseguibile Java contenuto in un archivio Java (file .jar) eseguito tramite il

comando *hadoop*. Il driver può fisicamente essere eseguito anche su una macchina esterna al cluster, a patto che possa comunicare con tutte le macchine del cluster. Nel caso più semplice il driver si limita ad effettuare il caricamento delle risorse del job su HDFS, comunicare al jobtracker di eseguire il job e terminare. Nel caso di algoritmi *multiround*, il driver ha invece il compito di eseguire i job nell'ordine prestabilito e monitorarne l'esecuzione.

3.4.2 Job, Task e Task Attempt

Prima di proseguire nella presentazione di MapReduce, occorre chiarire ad alto livello il suo funzionamento, introducendo nel contempo la terminologia riguardante la suddivisione e schedulazione di un job.

A livello driver, un job è visto essenzialmente come una configurazione. Questa contiene in primo luogo i nomi delle classi Mapper e Reducer (vedi 3.4.3), ed i percorsi di input ed output. A livello job possono inoltre essere impostate molte proprietà che influenzano il comportamento del framework, ridefinendo i valori contenuti nella configurazione globale di Hadoop, ed anche parametri specifici destinati ad essere letti nel codice di Mapper e Reducer.

Quando viene inviato al jobtracker ed eseguito, il job viene decomposto in *task*. Ogni task rappresenta una entità schedulabile astratta, composta da un insieme di dati e dall'operazione che vi verrà svolta. Esistono quindi due tipi di task: *map task* e *reduce task*. Definita la lista di task, il jobtracker schedula la loro esecuzione sui tasktracker. Come introdotto in fase di presentazione della struttura di MapReduce, una particolare esecuzione di un task su un trasktracker viene detta *task attempt* o semplicemente *attempt*. L'attempt, come si può dedurre dal nome, rappresenta un tentativo di esecuzione di un task. Nel caso che un attempt fallisca, il jobtracker schedula un altro attempt per lo stesso task su un tasktracker differente. Un attempt può fallire perché:

- Viene lanciata una RuntimeException;
- La JVM che lo ospita termina inaspettatamente;
- Non riporta progressi per un tempo troppo lungo (di default 10 minuti).

Se il numero di attempt falliti per task supera una certa soglia, il task fallisce e, di default, fallisce l'intero job. Al contrario, un task ha successo quando almeno un attempt termina con successo.

All'interno del cluster possono essere presenti più attempt per lo stesso task. Esiste infatti una funzionalità detta *esecuzione speculativa*: in caso un attempt proceda più lentamente del previsto, il jobtracker può assegnare un altro attempt concorrente per lo stesso task. Questa funzionalità è attiva di default.

Gli attempt possono anche essere terminati: questo solitamente avviene nel caso che un attempt concorrente termini con successo oppure in caso che il job fallisca. Gli attempt interrotti sono marcati come "*killed*" ed il loro numero non contribuisce al fallimento di un task.

3.4.3 Dalle funzioni agli oggetti

Nell'implementazione Hadoop le funzioni del paradigma vengono implementate tramite oggetti. Come introdotto nel capitolo precedente, i termini *Mapper* e *Reducer* assumono in Hadoop un significato molto preciso: sono gli oggetti Java che implementano rispettivamente la funzione *map* e la funzione *reduce*.

Oltre a Mapper e Reducer, in Hadoop esistono altre tipologie di oggetti importanti che contribuiscono alla definizione di un job. Le loro classi sono dichiarate nella configurazione di un job e permettono di ottimizzare o modificare la semantica della fase di shuffle.

Poiché la struttura e le relazioni sono complesse, si è preferito presentarle in modo separato, concentrandosi qui sulla descrizione degli oggetti e la nomenclatura. La descrizione dettagliata del ruolo di questi oggetti in un job è presentata invece nella Sottosezione 3.4.7.

Mapper e Reducer

Con il termine Mapper si intende un oggetto (in senso Java) che implementa il metodo map, mentre con il termine Reducer si intende un oggetto che implementa il metodo reduce. Questi metodi contengono naturalmente l'implementazione delle funzioni *map* e *reduce*, ma, uscendo dal paradigma, è possibile fare in modo che svolgano side-effect o agiscano sullo stato dell'oggetto. Il framework infatti non istanzia un nuovo oggetto per ogni chiamata ai metodi ma bensì un oggetto per ogni task attempt, mantenendo una granularità più grossa.

La collezione di coppie chiave-valore che vengono elaborati da un singolo Mapper è detta *input split* o semplicemente *split*. Queste coppie provengono sempre e

soltanto da un singolo file di input. La dimensione massima di uno split è configurabile, ma solitamente corrisponde a quella di un blocco HDFS. Il metodo map è invocato dal framework prendendo in input un singolo record alla volta.

Le coppie in ingresso ad un singolo Reducer prendono invece il nome di *partition*, o in Italiano, *partizione*. Una partizione è costituita da più multi-insiemi di coppie con la stessa chiave, che in Hadoop sono detti *groups* (in Italiano *gruppi*). La divisione dei gruppi in partizioni viene effettuata da un oggetto, il *Partitioner*, personalizzabile a livello di job. Il framework provvede a invocare il metodo reduce una volta per ogni gruppo presente nella partizione, fornendo un iteratore sulle coppie del gruppo.

In ultimo, è importante notare che un Mapper ed un Reducer su Hadoop possono elaborare solo uno specifico formato chiave-valore. Questo formato non deve essere lo stesso: come da paradigma, i multi-insiemi di input, intermedio e di output possono avere chiavi e valori da universi differenti. Essi però devono essere univocamente determinati in fase di configurazione di un job.

Combiner

In Hadoop con il termine Combiner si intente un oggetto opzionale simile al Reducer: opera su gruppi con una funzione *reduce*. La differenza tra Combiner e Reducer è relativa al momento in cui viene utilizzato.

Il Combiner infatti viene invocato in un momento corrispondente alla fase di shuffle del paradigma, con lo scopo di ridurre la quantità di dati da scambiare. L'output di un Combiner viene quindi successivamente partizionato ed elaborato nella fase di reduce.

Hadoop non fornisce nessuna garanzia rispetto al numero di volte che una coppia viene elaborata da un Combiner: può essere elaborata più volte così come nessuna. Affinché la semantica di un job sia coerente, l'applicabilità di un Combiner è limitata ai casi in cui la funzione *reduce* (o la prima parte della funzione *reduce*) è una funzione di aggregazione associativa e commutativa.

Partitioner, Grouping Comparator e Sorting Comparator

L'oggetto Partioner si occupa di dividere le coppie nelle partizioni. A livello implementativo fornisce semplicemente un metodo che, dati una chiave ed un valore, restituisce il numero della partizione. L'implementazione di default, detta

HashPartitioner, noto il numero di reduce task (vedi 3.4.4), distribuisce le coppie utilizzando una funzione hash della sola chiave. Questo distribuisce il carico di lavoro in modo uniforme solo se le chiavi sono uniformemente distribuite ed i valori della stessa dimensione: in caso contrario la taglia (in termini di quantità di dati) delle partizioni potrebbe risultare sbilanciata. [3]

Mentre la divisione in partizioni viene effettuata usando il Partioner, la divisione in gruppi viene eseguita ordinando le coppie all'interno di una partizione: le coppie con la stessa chiave si troveranno adiacenti. Grouping Comparator (detto anche OutputValueGroupingComparator) permette di modificare il criterio con cui vengono ordinate le chiavi in questa fase, e quindi influire sulla formazione dei gruppi. Il Sorting Comparator (detto anche OutputKeyComparator) viene invece utilizzato per definire l'ordine degli elementi all'interno di uno specifico gruppo. Se questi oggetti non vengono esplicitamente dichiarati, l'implementazione standard utilizza il metodo compareTo implementato nelle chiavi.

3.4.4 Parallelismo e Numero di Task

Il parallelismo in Hadoop è gestito in modo diverso rispetto al paradigma. Da una parte il massimo parallelismo (in termini di unità di calcolo parallele) corrisponde al numero di slot disponibili. Dall'altra parte, affinché venga sfruttato tutto il parallelismo è necessario che il numero di task presenti sia almeno pari a questo numero. Dal punto di vista della quantità di task di cui un job è composto, vi è una radicale differenza tra map task e reduce task.

Il numero di map task dipende dai file di input del job. Per definizione vi è un map task per ogni input split, la cui dimensione è definita in fase di configurazione di Hadoop. Solitamente essa è pari alla dimensione di un blocco HDFS. Come descritto in 3.4.3, uno split è composto da record provenienti da uno stesso file: vi è quindi almeno uno split per ogni file. Perciò si ha che

$$\#\ \text{map task} = \#\ \text{split} = \sum_{\text{file in input}} \left\lceil \frac{\text{dimensione del file}}{\text{dimensione split}} \right\rceil$$

Il numero di reduce tasks invece è definito per ogni job da parte dell'utente tramite la proprietà `mapred.reduce.tasks` che di default ha valore 1. Esistono

[3] Se ciò accade, una possibile soluzione è creare un Partioner basato su una distribuzione campione presente nella libreria Hadoop. Per maggiori informazioni vedere[29, p.238].

delle linee guida per impostare questa proprietà: detto $n_{reduceslot}$ il numero di reduce slot disponibili sul cluster, la letteratura tecnica consiglia un numero di task pari a $k \cdot (n_{slot} - \epsilon)$, in modo da effettuare i task in k ondate successive, tollerando ϵ fallimenti. Quindi:

$$\#\text{ reduce task} = \texttt{mapred.reduce.tasks} \approx k \cdot (n_{reduce\ slot} - \epsilon)$$

Valori di k maggiori di 1 permettono una miglior distribuzione del carico, a spese di maggior tempo di setup [29, p. 195]. Nel caso il numero di gruppi effettivamente presenti nella fase di shuffle sia minore del numero di reduce task, questi vengono eseguiti ma terminano immediatamente.

Mentre non è possibile eliminare la componente *map* di un job Hadoop (essa può essere resa una semplice funzione identità), è permesso configurare il numero di reduce task a 0. Quando questo avviene, Hadoop considera il job come composto solo dalla fase di map, al termine della quale l'output verrà scritto in output direttamente dai Mapper.

3.4.5 Formato dei Dati

Una delle critiche al paradigma MapReduce citate nel Capitolo 1 è quella di dipendere da inefficienti formati testuali come sorgente di dati [21]. Sebbene i formati testuali rappresentino un importante caso d'uso, Hadoop MapReduce, così come l'implementazione Google, supporta vari formati per i dati sul file system distribuito.

In Hadoop la lettura e scrittura sono demandate a due classi della libreria: *InputFormat* e *OutputFormat*. Queste classi possono essere estese dall'utente per supportare qualsiasi formato, sia esso di input o di output. Nella libreria di Hadoop sono incluse sottoclassi per i formati più comuni.

Formato binario su HDFS: Sequence File

Per gli scopi di questa Tesi è importante il ruolo del formato *sequence file*. Con questo termine viene indicata una struttura dati a livello HDFS pensata per contenere oggetti Java serializzati in forma di coppie.

Una importante proprietà di un formato di ingresso per MapReduce è la divisibilità (*splitability* in Inglese), intesa come la garanzia di accessibilità per blocchi

in caso il file si estenda su più di un blocco. Questo permette ai map task attempt di poter operare su un singolo blocco, sfruttando la località.

I sequence file hanno l'importante caratteristica di essere divisibili (*splitable*) e contemporaneamente supportare la compressione. Quest'ultima può essere effettuata in due modalità: per record o a blocchi, spiegate nel seguito.

La struttura di un sequence file non compresso e di uno compresso per record sono simili. Essi sono costituiti da (vedi Figura 3.5) :

- un header, contenente metadati quali i nomi delle classi degli oggetti chiave e valore, l'eventuale algoritmo e modalità usati per la compressione, e la codifica per il marcatore di sincronizzazione;

- i record; all'inizio di ogni record sono presenti due campi (da 4 byte ciascuno) con indicata la lunghezza in bit dei campi chiave e valore del record stesso. Essi sono seguiti dai relativi campi contenenti gli oggetti serializzati. Nel caso la compressione per record sia abilitata, il campo valore è memorizzato compresso;

- il marcatore di sincronizzazione, posizionato ad intervalli nel file, è utilizzato per permettere durante la lettura di trovare una posizione di inizio record da ogni posizione all'interno del file, senza che sia necessario scorrerlo fino all'inizio. A livello di specifica, l'ammontare dello spazio occupato dai marcatori non può superare l'1% dello spazio del file.

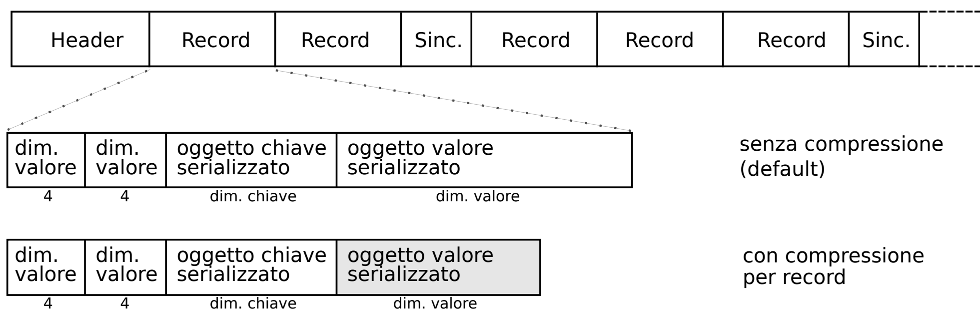


Figura 3.5 – Struttura di un sequence file e di un suo record, sia nel caso non sia attivata alcuna compressione, sia nel caso sia attivata la compressione per record. Sono evidenziati in grigio i campi compressi.

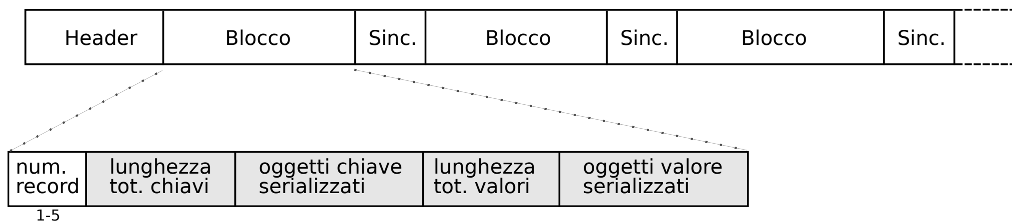


Figura 3.6 – Struttura di un sequence file in caso sia attivata la compressione a blocchi. Sono evidenziati in grigio i campi compressi.

Nel caso venga invece attivata la compressione a blocchi (vedi Figura 3.6), i record vengono raggruppati in blocchi di una dimensione fissata in fase di creazione del file. Per ogni blocco di record, sono presenti:

- un header di blocco, dove sono memorizzati il numero di record presenti, può occupare da 1 a 5 byte;
- un campo compresso contenente la lunghezza totale del successivo campo chiavi;
- un campo chiavi, compresso, con gli oggetti chiave serializzati;
- un campo compresso contenente la lunghezza totale del successivo campo valori;
- un campo valori, anch'esso compresso, con gli oggetti chiave serializzati.

A differenza della compressione per record, questa modalità trae vantaggio dalle similitudini tra i record, e risulta pertanto più compatta.

I sequence file sono progettati come formato binario naturale per memorizzare i dati che debbano essere forniti a più job MapReduce, o in caso di job concatenati: sono quindi la scelta naturale per algoritmi su più round.

Trasmissione

Hadoop adotta per la trasmissione dei dati nella fase di shuffle un proprio framework di serializzazione, pensato per ridurre l'overhead dovuto alla trasmissione dei metadati. Questo framework prende il nome di *Writables*. Rispetto all'implementazione standard della serializzazione di Java (Java Object Serialization),

Primitiva Java	Implementazione Writable	Dim. Serializzata (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Tabella 3.1 – I tipi primitivi di Java, la loro implementazione Writables e la dimensione serializzata in byte

Writables risulta più compatto e veloce, ma obbliga il ricevente a conoscere il formato e la struttura dei dati in ingresso [29, p. 102].

In Tabella 3.1 sono riportate le dimensione serializzate dei tipi primitivi di Java, il loro wrapper di default e la dimensione serializzata in byte. All'interno della libreria di Hadoop sono presenti molte altre implementazioni (Oggetti generici, valori null, stringhe, Collections etc.). In molti casi però è utile scrivere delle proprie implementazioni Writables per chiavi e valori utilizzando i metodi analoghi alle implementazioni delle primitive. Questo è l'approccio seguito durante la fase sperimentale, come descritto nella Sottosezione 4.2.1.

3.4.6 Input e Output di un job

MapReduce prevede che, a livello di job, l'input e l'output da HDFS vengano definiti sotto forma di path.

L'input path è vincolato ad esistere, ma può essere vuoto. Di default tutti i file contenuti in questo path, il cui nome non inizia con il carattere '.', sono considerati come input per il job e processati di conseguenza.

Per salvaguardare i dati prodotti, l'output path invece non può essere già presente su HDFS: se è presente un job fallisce in fase preliminare. All'interno di questo path l'output è scritto organizzato per task. Ogni singolo reduce task (o map task, nel caso di un job solo map) produce un file separato, il cui nome non

può essere controllato. È possibile selezionare il nome di output solo utilizzando delle apposite classi della libreria di Hadoop al posto del normale output del job. Questa tecnica può essere utilizzata per ottenere in Hadoop un comportamento analogo a O_r nel modello (vedi Sezione 2.2.2).

3.4.7 Anatomia di un job

L'esecuzione di un job in MapReduce è una complessa sequenza di operazioni. La divisione presentata a livello di paradigma in fase di map, fase di shuffle e fase di reduce non corrisponde in modo semplice a come un job viene eseguito. In particolare la fase di map e la fase di reduce non corrispondono all'esecuzione dei map attempt e dei reduce attempt. Le due tipologie di attempt sono infatti eseguite in contemporanea, sebbene raggiungano il picco di attività in momenti differenti. La fase di shuffle è invece il risultato di operazioni svolte sia da map attempt sia da reduce attempt.

In questa sezione è descritto, ad un buon livello di dettaglio, l'esecuzione di un job in MapReduce, come avviene nella versione 0.20.203 [29].

Job setup

Prima che l'esecuzione del calcolo inizi, un job deve essere distribuito sul cluster ed inviato al jobtracker, in modo che possa essere assegnato ai nodi di calcolo. Questo processo è schematizzato in Figura 3.7.

La prima fase dell'invio di un job MapReduce riguarda il driver. Per prima cosa il driver contatta il jobtracker, ricevendo un identificativo per il job (*jobid*). Procede poi a contattare HDFS per calcolare gli *input split* per i file in ingresso. In ultimo copia su HDFS le risorse necessarie per eseguire il job, tra cui il file JAR contenente le classi utilizzate non presenti nella libreria, la configurazione e gli input split calcolati. Questi dati sono memorizzati su HDFS con un fattore di replicazione molto elevato (di default 10) in modo che nel cluster siano presenti molte copie locali. Al termine di queste operazioni, il driver comunica al cluster l'invio effettivo del job, che è quindi pronto per l'esecuzione.

A questo punto il jobtracker si occupa di calcolare la sequenza di task da effettuare. Per fare ciò recupera da HDFS la lista degli split e la loro posizione. Il numero di map task e reduce task sono quindi calcolati come spiegato nella Sottosezione 3.4.4. In questa fase ad ogni task viene assegnato un identificativo (*taskid*).

Usando le risposte (*acknowledgment, ack*) agli heartbeats, il jobtracker assegna ad ogni tasktracker un task attempt da eseguire dalla lista dei task in attesa, dando priorità ai map task. Mentre i reduce task vengono assegnati in ordine progressivo, i map task vengono assegnati ai tasktracker in ordine di preferenza

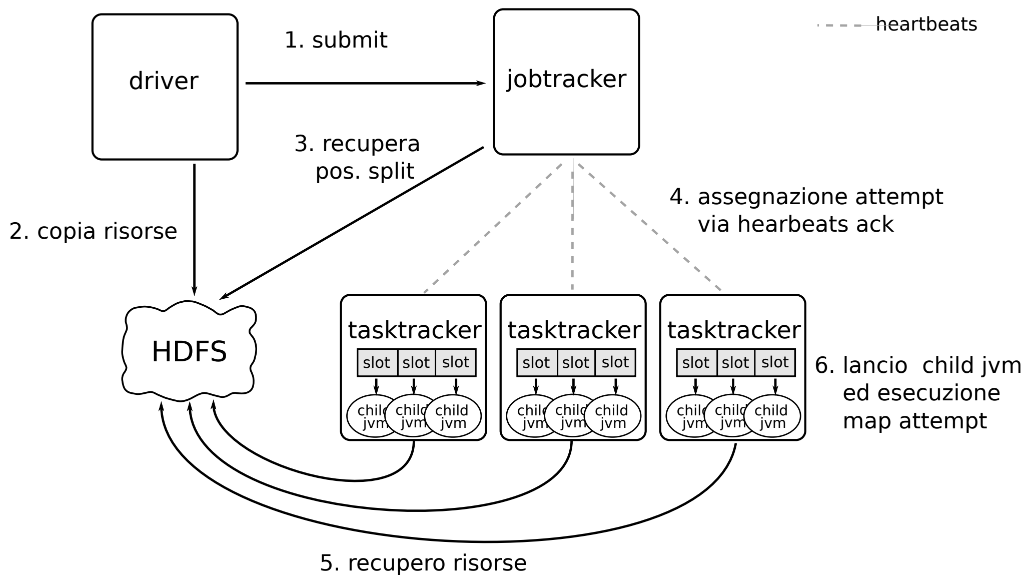


Figura 3.7 – Job setup in Hadoop MapReduce.

inversamente proporzionale alla distanza dallo split che sono destinati ad elaborare. In particolare vengono preferiti i nodi che possiedono una copia dello split nel datanode co-locato al tasktracker.

Quando i tasktracker ricevono un task attempt da eseguire, procedono recuperando da HDFS le risorse associate che erano state caricate dal driver. Per ogni task attempt i tasktracker creano una nuova JVM figlia che esegue il codice relativo all'attempt.

Map Task Attempt

Come anticipato, un map attempt si occupa non solo dell'esecuzione del map propriamente detto, ma esegue anche una parte dello shuffle. Queste operazioni sono schematizzate in Figura 3.8.

Una volta avviato, l'attempt istanzia l'oggetto Mapper, recupera l'input split assegnato e lo elabora tramite chiamate successive al metodo map. L'output di ciascuna chiamata viene scritto su un buffer circolare in memoria. Quando questo buffer supera una soglia, un thread procede a svuotarlo. Usando una istanza del Partioner, le coppie vengono suddivise nelle partizioni di destinazione e, all'interno di ogni partizione, vengono ordinate e scritte su un file locale detto *spill*. Se è definito un Combiner, ogni partizione creata dal thread viene elaborata

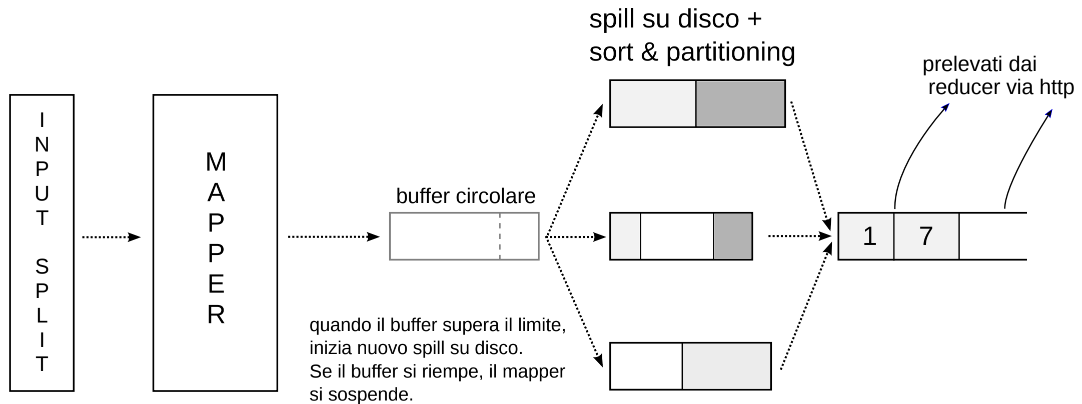


Figura 3.8 – Schema delle operazioni eseguite da un map task attempt senza Combiner. Se il Combiner è presente, viene invocato sulle partizioni al termine di ciascun merge degli spill.

tramite di esso prima di essere scritta su disco.

Se durante l'esecuzione il buffer circolare si riempe, il thread che esegue il metodo map si ferma fino a quando il buffer non viene svuotato. Quando questo avviene, viene creato un nuovo spill su un file separato.

Se sono presenti più spill quando tutte le coppie dello split sono state elaborate, questi vengono uniti tramite mergesort in sequenza. Anche qui, se è definito un Combiner, questo viene utilizzato prima di scrivere il risultato su un nuovo spill.

Dopo che tutti gli spill sono stati uniti in un file unico, l'attempt termina e comunica al tasktracker le partizioni per le quali esistono dati locali. Queste informazioni vengono poi passate al jobtracker. Il tasktracker si occupa di rendere disponibili i dati tramite protocollo HTTP affinché vengano prelevati dai reduce task. Da notare che l'output di un map attempt non viene eliminato dopo essere stati prelevato, per prevenirne la perdita in caso di fallimento di un reduce attempt. Verrà eliminato solo a job completato.

Reduce Task Attempt

Anche il reduce attempt esegue operazioni complesse, non sovrapponibili alla fase di reduce (vedi Figura 3.9).

La prima operazione eseguita consiste nel recuperare i dati della partizione assegnata tramite HTTP, utilizzando thread paralleli. Poiché non tutti i map task terminano contemporaneamente, i reduce task attempt vengono eseguiti in

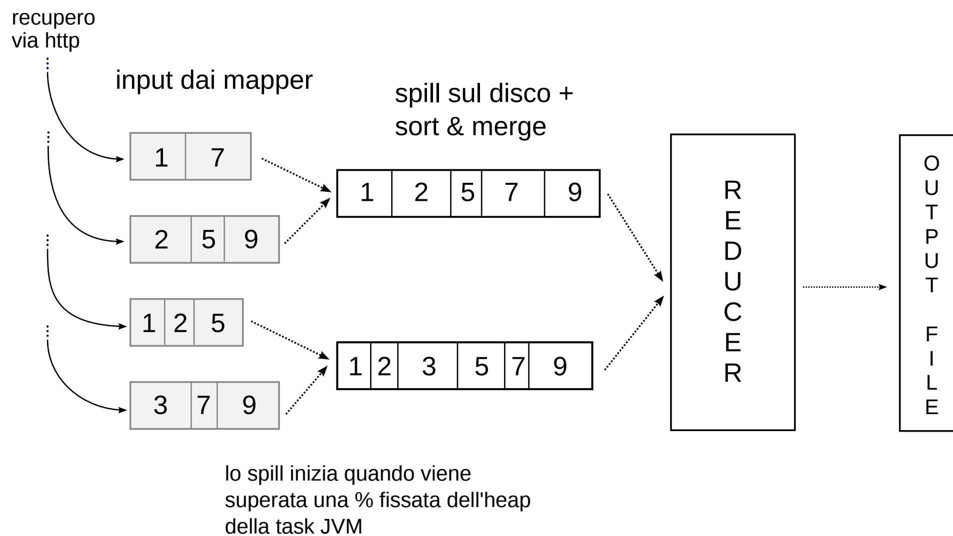


Figura 3.9 – Schema delle operazioni eseguite da un reduce task con output su HDFS.

sovrapposizione a questi, recuperando i dati man mano che diventano disponibili. Le informazioni riguardo alla posizione dei dati viene recuperata dal jobtracker tramite le risposte agli heartbeats.

Le coppie recuperate dai map task attempt vengono immagazzinate in memoria, fino a quando non superano una percentuale di occupazione della memoria heap a disposizione del reduce attempt. In questo caso vengono scritte su disco, in un apposito file locale. Nel caso che siano disponibili dati provenienti dai diversi map, durante questa operazione questi vengono anche uniti, sempre tramite mergesort. Nella configurazione di default la percentuale di memoria heap dedicata a questa operazione è 0%, quindi tutti i dati in input all'attempt vengono scritti immediatamente su disco.

Quando tutte le coppie della partizione sono state recuperate, il reduce task attempt procede con una sequenza finale di mergesort. L'output dell'ultima unione viene inviato all'oggetto Reducer, fornendo al metodo reduce la prima chiave di ciascun gruppo ed un iteratore alla sequenza di valori associata.

Se è utilizzato HDFS, l'output della funzione reduce, man mano che viene generato, è scritto direttamente sul file system distribuito. Quando tutte le coppie della partizione sono state elaborate, il reduce attempt termina con successo.

3.5 Differenze Rispetto a Modello e Paradigma

Le differenze tra il modello MR(m,M) ed il paradigma sono molteplici, ma sottili. In questa sezione si cerca di sottolinearne alcune, che si ritengono essere particolarmente importanti.

Parallelismo

Una importante differenza di Hadoop rispetto a paradigma e modello riguarda il parallelismo durante la fase di reduce. Esso è influenzato dalle diverse modalità di shuffle, con l'introduzione del concetto di partizione, oltre che dalla scelta di lasciare definire esplicitamente il numero di reduce task. Si ha che:

- il numero di slot rappresenta il massimo parallelismo reale del sistema; esso è una caratteristica del cluster, ed aumenta se si scala orizzontalmente. Può diminuire inaspettatamente durante una esecuzione a causa di guasti;
- il numero di reduce task rappresenta il massimo parallelismo sfruttabile dall'esecuzione; se è minore del numero di slot disponibili, quelli in soprannumero sono eseguiti ma terminano immediatamente;
- il numero di gruppi rappresenta il massimo parallelismo possibile del problema. Esso in generale non è noto a priori, ma dipende dai dati. In MR(m,M) corrisponde $|U_r|$;
- In numero di partizioni è (a meno di implementazioni personalizzate del Partitioner) uguale al numero di reduce task configurato. Esso rappresenta il numero di reduce task che ricevono effettivamente dati.

Riassumendo, è possibile esprimere il parallelismo in fase di reduce come segue:

$$\text{parallelismo reduce} = \min(n_{reduceslot}, n_{gruppi}, n_{partizioni}, n_{reduce\ task})$$

Algoritmi su più round

Dalla struttura delle API si nota come Hadoop sia focalizzato nell'esecuzione di singoli job, in maniera analoga al paradigma, in cui non emerge il concetto di round. Non esiste quindi un concetto Hadoop sovrapponibile a quello di MR-Algoritmo. Come verrà descritto in fase implementativa nel Capitolo 4, il

problema è stato risolto complicando il codice del driver inserendovi una parte della logica. Questo ha però comportato ulteriori difficoltà di sviluppo, poiché ha reso più complesso controllare la correttezza del codice. Per riuscire ad effettuare il debug di questo tipo di driver è stato sviluppato un piccolo framework di astrazione descritto in Appendice C.1.

Funzioni e Stato

Rispetto al paradigma, Hadoop implementa oggetti rispetto a funzioni: permette quindi di mantenere uno stato. Questo, unito al diverso livello di granularità (i.e. un Reducer esegue più funzioni reduce, una per ogni gruppo nella partizione) ed alla possibilità di controllare nel dettaglio l'assegnazione delle chiavi, permette di formulare computazioni in modo sostanzialmente diverso. Rimane da chiarire come questa differenza si ripercuota sulle possibilità espressive.

Map, Shuffle e Reduce

Rispetto a MR(m,M), Hadoop permette l'esecuzione di job solo map mentre non permette l'esecuzione di job solo reduce, come previsto dal modello. La fase di map è ineliminabile: può essere solo resa inefficace eseguendo una funzione identità.

Inoltre in Hadoop le fasi di map, shuffle e reduce non sono ben definibili. La divisione in task ed il modello di resilienza sono costruiti in modo che i compiti di shuffle siano suddivisi tra map task attempt e reduce task attempt.

Mapper e Reducer

Da notare infine come il modello di esecuzione e di divisione delle chiavi in Hadoop sia in contrasto con la definizione di Mapper e Reducer presente in una parte della letteratura modellistica. Anche la definizione di memoria locale in MR(m,M) ne risulta in qualche modo influenzata: se è vero che i dati elaborati da una chiamata alla funzione reduce ρ_r per un MR-algoritmo valido è $O(m)$, i dati in ingresso ad un Reducer sono invece dipendenti dal numero di reduce task dichiarati. Facendo riferimento alle linee guida (vedi 3.4.4), questo numero è proporzionale al numero di slot, quindi in qualche modo ad M/m.

Capitolo 4

Moltiplicazione Matriciale Densa

Per validare il modello MR(m,M) consideriamo l'algoritmo per la moltiplicazione matriciale densa presentato all'interno dell'articolo [22]. Questo algoritmo propone un evidente tradeoff tra memoria locale m, memoria locale M e complessità in round. In questo capitolo viene presentato prima l'algoritmo in termini formali e pseudocodice, con la relativa complessità, poi le scelte implementative utilizzate in fase di validazione. In particolare nella Sottosezione 4.2.2 sono descritte in modo dettagliato tutte le versioni prodotte nel corso della Tesi.

4.1 L'algoritmo in MR(m,M)

L'algoritmo calcola la moltiplicazione matriciale tra due matrici A,B dense, tali cioè da avere un numero di elementi non nulli $\in \Theta(n)$. Detto $\mathcal{M}_{\sqrt{n}\times\sqrt{n}}$ l'insieme delle matrici quadrate di taglia $\sqrt{n}\times\sqrt{n}$, l'input e l'output dell'algoritmo risultato essere

$$\begin{aligned} \mathcal{I} &: A, B \in \mathcal{M}_{\sqrt{n}\times\sqrt{n}} \\ \mathcal{O} &: C = A \cdot B \end{aligned}$$

Gli elementi delle matrici sono considerati nel semianello $(S, \oplus, \odot)$: in questo contesto gli algoritmi per la moltiplicazione matriciale veloce come Strassen non sono utilizzabili, a causa della mancanza dell'inverso per la somma $\oplus$.

Gli elementi delle matrici A,B,C sono indicati rispettivamente con i simboli $a_{s,t}$, $b_{s,t}$ e $c_{s,t}$. Si suppone inoltre che le matrici in input siano fornite come un insieme di elementi codificato inizialmente in coppie chiave-valore nella forma $((s,t),(s,t,v_{s,t}))$ dove s è la riga dell'elemento, t la colonna, $v_{s,t}$ il valore.

Ciascuna matrice viene suddivisa in $\sqrt{\frac{n}{m}}$ blocchi di taglia $\sqrt{m} \times \sqrt{m}$. Questi blocchi sono indicati come $A_{i,j}, B_{i,j}$ e $C_{i,j}$ con $0 \leq i, j \leq \sqrt{\frac{n}{m}} - 1$.

$$A_{i,j}, B_{i,j}, C_{i,j} \in \mathcal{M}_{\sqrt{m} \times \sqrt{m}}$$

Per semplicità, le parti di algoritmo sono presentate in pseudo-codice considerando direttamente questi blocchi. Le modifiche per trasformare le chiavi (s,t) sono banali:

$$a_{s,t} \in A_{i,j} \Leftrightarrow \begin{cases} i = \lfloor s/m \rfloor \\ j = \lfloor t/m \rfloor \end{cases}$$

Le assegnazioni delle chiavi a partire da r e s all'interno dell'algoritmo possono quindi essere ricavate di conseguenza da quelle presentate in pseudocodice.

Divisione in Gruppi di Prodotti

Dalla divisione in blocchi, si ha che ogni singolo blocco di C è il risultato di una somma di prodotti tra blocchi di A e B:

$$C_{i,j} = \sum_{h=0}^{\frac{n}{m}-1} A_{i,h} \cdot B_{h,j}$$

La matrice C nel suo complesso può essere quindi calcolata tramite $\left(\frac{n}{m}\right)^{\frac{3}{2}}$ prodotti del tipo $A_{i,h} \cdot B_{h,j}$. Ognuno di questi prodotti viene indicato come $\mathcal{P}_{i,h,j}$. Questi prodotti vengono suddivisi in $\sqrt{\frac{n}{m}}$ gruppi, definiti come

$$\mathcal{G}_l = \left\{ \mathcal{P}_{i,h,j}, \forall i, j \in [0, \sqrt{n/m} - 1] : (i + h + j) = l \mod \sqrt{n/m} \right\}$$

con $0 \leq l \leq \sqrt{n/m} - 1$. Per come sono definiti i gruppi, ogni blocco, sia esso appartenente alla matrice A o alla matrice B, compare una sola volta per ogni gruppo $\mathcal{G}_l$. Nello stesso modo, per ogni gruppo, un solo prodotto contribuisce al blocco $C_{i,j}, \forall i, j : 0 \leq i, j \leq \sqrt{n/m}$

Esecuzione dei prodotti

Sia

$$K = \min\left(\frac{M}{n}, \sqrt{\frac{n}{m}}\right)$$

in ogni round della fase di moltiplicazione si eseguono tutti i prodotti contenuti in K gruppi successivi. Poiché in ogni gruppo viene eseguito un solo prodotto

che contribuisce al blocco di $C_{i,j}$, in ogni round vengono eseguiti K contribuiti per ogni blocco di C. Le matrici A e B sono replicate K volte nel primo round (Pseudocodice 1) e distribuite di round in round in modo che ogni chiamata alla funzione reduce ρ_r esegua una moltiplicazione appartenente a $\bigcup_{l=(r-1)K}^{rK} \mathcal{G}_l$ e ne sommi il risultato al contributo ottenuto nei round precedenti per i gruppi con lo stesso indice k, tale che $k = l \mod K$ (Pseudocodice 2).

```
input  : (i,j) chiave,
         M_{i,j} ∈ {A_{i,j}, ∀i, j} ∪ {B_{i,j}, ∀i, j}
         K fattore replicazione
output: Operandi A_{i,h} e B_{h,j} per il prossimo round

funzione ρ_1 begin
    for all M_{i,j} do
        // Genera la replicazione, assegnando la chiave
           opportuna
        for l ← 0 to K − 1 do
            if M_{i,j} ∈ A then
                h ← i − j − l  mod √(n/m)
                emit((i, h, l), M_{i,j})
            else if M_{i,j} ∈ B then
                h ← j − i − l  mod √(n/m)
                emit((h, j, l), M_{i,j})
```

Pseudocodice 1 – Algoritmo per la funzione *reduce* ρ_1 (al primo round). Si tratta di una funzione *map* simulata. I dati in input dipendono dall'input globale dell'algoritmo.

input : (i,j,k) chiave,
$A_{i,h}$ $B_{h,j}$ blocchi associati alla chiave (i,j,k)
$C_{i,j}$ contributo al blocco i,j di C con indice k (solo se $r>2$)
r indice di round
K fattore replicazione

output: Operandi $A_{i,h}$ e $B_{h,j}$ per il prossimo round,
contributo a $C_{i,j}$ con indice k

```
funzione ρ_r begin
    if r=2 then
        C_{i,j} ← 0 ∈ M_{√m×√m}
    // Esecuzione della moltiplicazione
    C_{i,j} ← C_{i,j} + A_{i,h} · B_{h,j}
    if r + 1 < (n/m)^{3/2} then
        // Se non è l'ultimo round della moltiplicazione,
           produce gli operandi per il prossimo round.
        emit((i, j, k), C_{i,j})
        j' ← h − i − (r + 1) · K  mod √(n/m)
        i' ← (h − j − (r + 1) · K  mod √(n/m)
        emit((i, j', k), A_{i,h})
        emit(i', j, k), B_{h,j})
    else
        // Esecuzione funzione map della prima somma
        for c_{r,c} ∈ C_{i,j} do
            emit((r, c, ⌊k/m⌋), c_{r,c})
```

Pseudocodice 2 – Algoritmo per la funzione *reduce* ρ_r durante la fase di moltiplicazione (ossia $1 < r < \frac{\sqrt{n}}{K\sqrt{m}}$)

Somma parallela

Terminata la fase di moltiplicazione occorre sommare i K contributi per ogni blocco di C. Questo viene eseguito tramite somme parallele lungo un albero di arietà m. Ogni chiamata alla funzione reduce si occuperà di sommare tutti gli elementi di m contribuiti che contribuiscono ad ogni singolo elemento di C $c_{i,j}$ (Pseudocodice 3).

```
input  : (i,j,k) chiave,
         L insieme valori associati alla chiave (i, j, k)
         m arietà dell'albero di somma
output: s somma degli elementi

funzione ρ_r begin
    s ← 0
    for  all c ∈ L do
        s ← s + c
    emit((i, j, ⌊k/m⌋), s)
```

Pseudocodice 3 – Algoritmo per la funzione *reduce* ρ_r per la somma parallela

Proposizione 4.1. *[22] L'algoritmo presentato per la moltiplicazione matriciale è un MR-algoritmo valido di complessità*

$$\Theta\left(\frac{n^{\frac{3}{2}}}{M\sqrt{m}} + \log_m n\right)$$

Dimostrazione. Occorre verificare che, in ogni round, siano soddisfatti i vincoli presentati nella Sottosezione 2.2.3 a pagina 24. Poiché nell'algoritmo presentato $O_r = \emptyset, \forall r$ e tutte le complessità RAM sono banalmente polinomiali, rimangono da verificare i vincoli di memoria locale e globale.

Al primo round, trattandosi di un *map* simulato, il numero di elementi associati ad una chiave dipende dalla disposizioni delle chiavi nell'input; essendo le chiavi di input nella forma (r, s) dove r e s sono indici di elemento, e k indice di operatore, ogni chiamata alla funzione ρ_1 elabora al più due elementi, usando spazio di lavoro costante. Gli elementi di input sono pari a $2 \cdot n$, ed essendo per ipotesi $M \geq n$, anche il vincolo di memoria globale risulta soddisfatto.

Durante la moltiplicazione, ogni chiamata a ρ_r calcola 1 prodotto tra blocchi di dimensione $\sqrt{m} \times \sqrt{m}$ e la somma con il contribuito parziale a $C_{i,j}$. Lo spazio necessario per memorizzare l'input risulta quindi pari a $3 \cdot m$. Lo spazio di lavoro necessario è pari al più ad m. Si ha quindi

$$m_{r,(i,j,k)} \leq 3m + m \in O(m)$$

Da come sono distribuite le chiavi, l'insieme U_r ha taglia $K \cdot n/m$. Perciò:

$$\begin{aligned}
\sum_{(i,j,k)\in U_r} m_{r,(i,j,k)} &\leq (K \cdot \frac{n}{m}) \cdot (4m) \leq \\
&\leq 4 \cdot \frac{M}{n} \cdot \frac{n}{m} \cdot m = \\
&= 4 \cdot M \in O(M)
\end{aligned}$$

Durante la somma parallela, ogni chiamata alla funzione reduce calcola al più m somme di elementi ed il numero di chiavi in U_r è sempre minore o uguale $K \cdot n/m$, da cui latesi.

La complessità in MR(m,M) è definita come il numero di round al caso pessimo. Per l'algoritmo presentato, dati i valori dei parametri il numero di round è fissato. Detto R il numero di round, si ha quindi

$$\begin{aligned}
R &= 1 + \frac{\sqrt{\frac{n}{m}}}{K} + \lceil \log_m(n) \rceil \leq \\
&\leq 2 + \sqrt{\frac{n}{m}} \cdot \frac{n}{M} + \log_m(n) = \\
&= 2 + \frac{n^{3/2}}{M\sqrt{m}} + \log_m(n) \in \Theta\left(\frac{n^{\frac{3}{2}}}{M\sqrt{m}} + \log_m n\right)
\end{aligned}$$

□

Si dimostra che l'algoritmo presentato è ottimo [22]. Da notare che la complessità presentata nella Proposizione 4.1 è raggiungibile solo implementando la somma parallela per elementi e non per blocchi. Visto lo Pseudocodice 4, si ha infatti la seguente proposizione:

Proposizione 4.2. *Considerati di $\frac{Kn}{m}$ blocchi matriciali $C_{i,j,k} \in \mathcal{M}_{\sqrt{m}\times\sqrt{m}}$ rappresentati in coppie chiave-valore nella forma*

$$((i, j, \lfloor k/a \rfloor), C_{i,j,k}) \quad \text{con } 0 \leq i, j \leq \sqrt{n/m}, 0 \leq k \leq K$$

la somma parallela per blocchi tramite un albero di arietà a in MR(m,M) richiede $\lceil \log_a n \rceil$ *round, dove* $a \in O(1)$ *rispetto a m.*

Dimostrazione. La somma termina banalmente dopo $log_a(n)$ round. In ogni round, una chiamata alla funzione reduce ρ_r riceve $a \cdot m$ elementi. Perciò:

$$m_{r,(i,j,k)} = a \cdot m$$

Ora per il vincolo di memoria locale (vedi 2.2.3) occorre che $m_{r,(i,j,k)} \in O(1)\ \forall r$. Questo è possibile se e solo se $a \in O(1)$. □

input : (i,j,k) chiave,
L insieme blocchi $\in \mathcal{M}_{\sqrt{m} \times \sqrt{m}}$ che contribuiscono alla sottomatrice $C_{i,j}$ associati alla chiave (i, j, k)
a arietà dell'albero di somma
output: s somma degli elementi

funzione ρ_r **begin**
  $S \leftarrow 0(\in \mathcal{M}_{\sqrt{m} \times \sqrt{m}})$
  for all $C \in L$ **do**
    $S \leftarrow S + c$
  `emit(`$(i, j, k/a), S$`)`

Pseudocodice 4 – Algoritmo per la funzione *reduce* ρ_r per la somma parallela per blocchi. Per soddisfare i vincoli di MR(m,M), l'arietà a deve essere $O(1)$ rispetto a m. - vedi Proposizione 4.2

4.2 Implementazione

L'algoritmo in oggetto può essere implementato in vari modi in Hadoop. Per prima cosa, occorre fare delle scelte di tipo più strettamente tecnologico e non correlate direttamente al modello, quali i formati delle chiavi, i formati dei file di input/output, l'algoritmo di moltiplicazione matriciale nei Reducer e la struttura dei driver.

Per verificare la validità del modello, sono state poi realizzate diverse possibili implementazioni, che, pur condividendo l'algoritmo astratto, lo implementano in modo diverso. Queste implementazioni sono state suddivise in livelli progressivi, a seconda dell'aderenza al modello teorico. Per ogni livello concettuale sono state poi sperimentate due diverse strategie in relazione al formato dei dati, ed alle modalità di moltiplicazione. Le implementazioni oggetto di studio sono descritte nel dettaglio al termine di questaSezione.

4.2.1 Scelte comuni

Formato dei Chiavi e Valori

Chiavi e valori sono stati implementati utilizzando degli oggetti Writables personalizzati (vedi Sottosezione 3.4.5).

L'implementazione per la chiave (i, j, k) è stata realizzata tramite un oggetto WritableComparable contenente tre campi interi, serializzati in modo analogo a IntWritable (Tabella 3.1, pag. 49). L'ordinamento predefinito ed il metodo .hashCode() sono stati implementati utilizzando tutti e tre i campi, per garantirne la compatibilità con le implementazioni standard di Partioner e GroupComparator.

Per quanto riguarda i dati, sono state realizzate due implementazioni Writable: MatrixItem e MatrixBlock. La prima, MatrixItem, rappresenta un singolo valore di una matrice, in accordo con quanto proposto nell'articolo di riferimento [22]. Essa utilizza tre campi interi a rappresentare riga, colonna e valore, ed un campo byte per rappresentare l'operatore di appartenenza (i.e. A, B o C). I vari campi sono serializzati in modo analogo a IntWritable e ByteWritable. La seconda implementazione, MatrixBlock, rappresenta invece un intero blocco di m elementi. Esso utilizza internamente un oggetto BlockRealMatrix della libreria *Apache Commons Math* [1] per rappresentare la matrice, ed un campo byte per rappresentare l'operatore di appartenenza. BlockRealMatrix utilizza al suo inter-

no campi di tipo double, ma in fase di serializzazione essi sono interpretati come interi e serializzati in modo analogo a IntWritable. Il campo operatore è invece serializzato come nell'implementazione precedente.

Implementazione	Campi	Dim. Serializzata (bytes)
IntTriple	int i, int j, int k	12
MatrixItem	int r, int c, int v, byte op	13
MatrixBlock	BlockRealMatrix b, byte op	$4m$ +1

Tabella 4.1 – Le implementazioni Writables utilizzate

Formato di Input/Output per Job

Una caratteristica dell'algoritmo sotto esame è la necessità di riutilizzare in input l'output di un round precedente. Poiché la portabilità non è richiesta per questi dati, la scelta naturale è stata di utilizzare come formato di input ed output per i vari job il formato Sequence File, descritto sempre in Sottosezione 3.4.5. In particolare, esso è stato utilizzato nella sua variante non compressa, usando gli oggetti chiave e valore descritti nella sottosezione precedente.

Algoritmo di Moltiplicazione

La moltiplicazione all'interno dei Reducer è stata implementata in modo diverso a seconda del formato dei dati in input. Nelle versioni che utilizzano MatrixItem la moltiplicazione è effettuata tramite l'algoritmo banale, con l'accortezza di scorrere il secondo operatore per colonne in modo da migliorare l'efficienza sulla cache.

Nelle versioni che utilizzano MatrixBlock invece viene sfruttato il metodo multiply() di BlockRealMatrix, che implementa la moltiplicazione matriciale a blocchi di dimensione tale da essere memorizzati in cache [1].

Driver, parametri ed interazioni con HDFS

La strategia algoritmica sotto esame richiede una precisa sequenza di round. Per tradurre questo in Hadoop, la scelta è stata di implementare dei driver complessi, che si occupino di configurare ed inviare i job corrispondenti. A ogni singola

implementazione utilizzata nella fase sperimentale corrisponde quindi un driver particolare.

In Hadoop inoltre l'output tra un job ed il successivo deve essere scritto su HDFS su un path (vedi 3.4.6, pag. 49). Il modello non prevede come questi dati debbano essere trattati. Essi pongono una evidente limitazione ai valori di n che possono essere trattati dal cluster. Infatti, al caso pessimo considerando le versioni presentate in 4.2.2, si ha che

$$\text{byte su disco} \geq 2B_n\alpha + 3KB_n + \sum_{r=0}^{\frac{\sqrt{n}}{(\sqrt{m}K)}} (3KB_n\alpha) + \sum_{r=0}^{\log_m(K)} (\frac{K}{m^r}B_n\alpha)$$

dove B_n è il numero di byte necessario per rappresentare una matrice di taglia n comprese le chiavi, α è il fattore di replicazione di HDFS. Il contributo evidenziato come 'copie locali' comprende le 3 matrici scritte su disco alla fine della fase di map (negli spill) ed all'inizio della fase di reduce, che condividono lo spazio su disco con le repliche HDFS. Le costanti presenti nelle formule sono molto elevate: B_n in particolare vale $25n$ se si utilizzano MatrixItem, e $n(5 + 13/m)$ se si utilizzano MatrixBlock. Per questo motivo è stato necessario implementare una strategia di cancellazione dei dati intermedi prodotti durante l'esecuzione: i driver si occupano, tramite operazioni su HDFS, di richiedere la cancellazione dei dati utilizzati in input dopo ogni singolo job.[1]

I driver sono quindi costituiti principalmente da due cicli (uno per la moltiplicazione ed uno per la somma) all'interno dei quali vengono configurati ed eseguiti i job. I driver rimangono sospesi fino a quando il jobtracker non riporta l'esecuzione terminata. Al termine dell'esecuzione essi comunicano con HDFS richiedendo l'eliminazione dei dati prodotti dal job precedente. Per semplificare il codice di driver, inoltre, essi rinominano il path di uscita (questa operazione avviene sui metadati e quindi non dovrebbe avere alcun impatto a questo livello).

I parametri necessari nell'esecuzione di Mapper e Reducer (quali ad esempio n,M,m) non sono passati sotto forma di chiave-valore, in quanto questo avrebbe comportato una decisa complicazione del codice. Hadoop fornisce infatti la possibilità di inserire questi parametri nella configurazione di un job, per essere poi recuperati in fase di creazione degli oggetti.

[1] Le formule esplicite con questa strategia di cancellazione sono invece proposte e discusse nel dettaglio nella Sezione 5.4.

Un esempio di Driver è riportato in Appendice C, nell'estratto di codice 3 a pagina 120.

4.2.2 Versioni: Strategie e Livelli

Come introdotto in apertura di Sezione, per vedere il comportamento dell'algoritmo in relazione a quanto previsto dal modello sono state sviluppate sei versioni, indicate con lettere progressive (a, b, c, d, e, e f). Queste possono essere divise in tre livelli di aderenza al modello. Il primo livello (L1) è fatto in modo da perseguire la massima aderenza rispetto al modello MR(m,M). Le versioni a questo livello sono una implementazione pedissequa dell'algoritmo descritto nella Sezione 4.1: tutti calcoli vengono eseguiti solo da Reducer. Da notare che, poiché Hadoop non consente job composti da solo Reducer, i Mapper sono comunque presenti, ma si limitano ad eseguire una funzione identità. Al secondo livello (L2) invece i Mapper svolgono una parte del calcolo attiva, diversa dipendentemente dal tipo di strategia, come descritto in seguito. Infine al terzo livello (L3) il calcolo viene svolto cercando di sfruttare le potenzialità offerte da Combiner e Partitioner.

Sono adottate complessivamente due strategie per gestire la replicazione. La prima (denominata S1, comprende le versioni a,b,c,d) consiste nello generare tutta la replicazione necessaria al primo round, e scrivere le matrici A e B replicate in output a ciascun round, trasformando le chiavi in modo da eseguire le moltiplicazioni successive. La seconda strategia (S2) invece consiste nel generare la replicazione ad ogni round a partire dalle matrici di input. Questa strategia è resa possibile solo permettendo l'uso di Mapper attivi (livello $>$ 2, versioni e,f) e punta a ridurre la quantità di dati scritta su HDFS a spese di un maggior lavoro.

Le versioni c e d sono utilizzate come termine di paragone rispetto alle versioni di livello 1 a e b: svolgono le stesse operazioni, solo che, invece di accorpare la fase di map con la fase di reduce precedente come da modello, la esplicitano.

Per i primi due livelli, per ogni strategia sono state implementate delle versioni sia che usano MatrixItem (a,c,e) come veicolo per i dati, sia che usano MatrixBlock (b,d,f). Poiché ogni Job deve dichiarare preventivamente le classi utilizzate per chiavi e valori, la stessa strategia richiede una diversa implementazione del driver (e diversi Mapper/Reducer) nei due formati. Come descritto in seguito, il formato MatrixItem risulta inefficiente: per questo motivo non ne viene presentata una versione al livello 3.

Le implementazioni dei driver sono schematizzate in Figura 4.1, mentre le caratteristiche delle versioni sono confrontate in Tabella 4.2. Nel dettaglio:

Livello 1

- Ver. a - Replicazione generata nel primo round con una fase di map simulata tramite Reducer, simile a quella presentata nello Pseudocodice 1. Alla fine di ogni round intermedio, gli operandi vengono emessi con le chiavi modificate, per assegnarli alla moltiplicazione del round successivo. Nell'ultimo round della moltiplicazione, invece, gli operandi non vengono emessi, e viene effettuato il primo step della somma parallela (come in Pseudocodice 2). Vengono utilizzati MatrixItem per tutto l'algoritmo.

- Ver. b - Al primo round i MatrixItem vengono raggruppati per blocchi di dimensione m (dividendo i, j della chiave iniziale per m) in una fase di map simulato. Nel round successivo il Reducer crea i MatrixBlock, e li replica come in Pseudocodice 1. Successivamente si comporta come la ver. a, con l'aggiunta che nell'ultimo round della moltiplicazione, oltre ad effettuare il primo step della somma, scompone anche MatrixBlock in MatrixItem.

Livello 2

- Ver. c - Versione di a con map espliciti. Nel primo round la replicazione viene generata direttamente dalla funzione *map*. Nei round successivi i Reducer non assegnano le chiavi modificate agli operatori, ma queste vengono invece assegnate dai Mapper del round successivo. L'ultimo round della moltiplicazione è identico ai precedenti, ed il primo step della somma viene effettuato da una successiva fase di map nel primo round della somma parallela. Con lo stesso principio, la somma parallela non modifica le chiavi nel Reducer ma nel Mapper successivo.

- Ver. d - Versione di b con map espliciti. Nel primo round la creazione per blocchi avviene grazie ai Mapper, ed i Reducer si limitano a creare i MatrixBlock. Nel secondo round i Mapper creano la replicazione. Nei round successivi i Reducer non assegnano le chiavi modificate agli operatori, ma queste vengono invece assegnate dai Mapper del round successivo. La trasformazione dei MatrixBlock in MatrixItem avviene con un round solo

map invece che nell'ultimo round della moltiplicazione. Per il resto, coincide con c.

- Ver. e - Utilizza i Mapper per evitare di scrivere su HDFS gli operatori replicati. Durante la moltiplicazione, i Mapper leggono gli operatori ed i contributi parziali. Gli operatori vengono replicati ed assegnati in base all'indice di round, mentre i contributi parziali sono emessi senza modifiche. I Reducer eseguono le moltiplicazioni ed emettono solamente i contributi parziali. La somma parallela è eseguita come in c. Utilizza MatrixItem.

- Ver. f - Versione di e con MatrixBlock. Nel primo round, i MatrixItem vengono raggruppati in MatrixBlock usando Mapper e Reducer. La moltiplicazione avviene in modo analogo a e. Prima della somma parallela, i MatrixBlock vengono scomposti tramite un round solo map. La somma viene eseguita come in e.

Versione	Livello	Descrizione	Strategia	Classe Valori	Classe Driver	Complessità in round al caso pessimo
a	L1	Solo Reducer	S1	MatrixItem	L1_TheoreticalPipeItemDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil + 1$
b			S1	MatrixBlock	L1_TheoreticalPipeBlockDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil + 2$
c	L2	L1 con Mapper	S1	MatrixItem	L2_MapPipeItemDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil$
d			S1	MatrixBlock	L2_MapPipeBlockDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil + 2$
e	L2	Replic. generata ogni round	S2	MatrixItem	L2_SimpleItemDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil$
f			S2	MatriBlock	L2_SimpleBlockDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil + 2$

Tabella 4.2 – Riassunto delle principali caratteristiche delle implementazioni, suddivise per livelli

Figura 4.1 – Struttura delle implementazioni, raggruppate per livelli (a sinistra) e per strategie di replicazione (a destra). Ogni rettangolo rappresenta un job, con nella parte sinistra la fase di map e nella parte destra la fase di reduce. Le fasi di map indicate con id eseguono la funzione identità. Le fasi di reduce barrate sono omesse (job solo map). Nelle frecce tra i job sono indicati come sono rappresentati i valori.

Capitolo 5

Verifica Sperimentale

In questo capitolo è raccolto il materiale sviluppato durante la parte sperimentale di questa Tesi, in termini di risultati e considerazioni.

Lo scopo iniziale infatti consisteva nell'osservare se la complessità in round espressa durante la presentazione dell'algoritmo nella Sezione 4.1 rappresentasse un indice del tempo di esecuzione parallelo. In particolare l'obbiettivo era studiare, fissato n, il tempo di esecuzione al variare di m e M, identificando i costi nascosti dell'esecuzione reale su Hadoop delle varie versioni, e confrontando il comportamento di varie scelte implementative.

Durante questo processo sono però emersi molti altri aspetti interessanti, che costituiscono un importante risultato del processo. In particolare l'apparato sperimentale si è rilevato sottodimensionato rispetto alle esigenze di computazione richieste: questo ha impedito di scalare tutte le versioni per matrici di taglia elevata, ma ha evidenziato e quindi fatto individuare i legami presenti tra le risorse del cluster ed i parametri m e M del modello.

I tempi di esecuzione riscontrati si sono rilevati molto elevati, precludendo per motivi di tempo uno studio accurato di molti dei possibili andamenti. Anche a causa di alcuni guasti avvenuti all'apparato sperimentale, solo una parte di parametri sono stati effettivamente sperimentati, ma questi sono comunque sufficienti a derivare degli indirizzi futuri.

In Sezione 5.1 è presentato l'apparato sperimentale, in termini di caratteristiche Hardware e configurazione, mentre in Sezione 5.1.3 è presentata la tecnica utilizzata per produrre i dati in ingresso, ed in Sezione 5.3 le misure effettuate sul cluster. In Sezione 5.4 sono presentati alcuni vincoli reali tra i parametri

del modello e le risorse del cluster, seguite in Sezione 5.5 da delle nuove versioni ottimizzate per migliorare la scalabilità rispetto a questi vincoli. I risultati sperimentali ottenuti sono presentati Sezione 5.6. Infine in Sezione 5.7 sono raccolte alcune considerazioni sui risultati e suggerimenti per futuri esperimenti.

5.1 Apparato Sperimentale

5.1.1 Hardware

Per gli esperimenti è stato utilizzato un cluster interno al Dipartimento. Dal punto di vista hardware, il cluster utilizzato è composta da 16 nodi identici le cui caratteristiche sono riassunte in Tabella 5.1. Rispetto alle caratteristiche consigliate per Hadoop [19], i nodi risultano sottodimensionati rispetto ad alcuni parametri, quali il numero di processori e lo spazio su disco disponibile, pur rientrando nei requisiti minimi.

I nodi sono collegati tra loro utilizzando una rete a 10 Gbps dedicata, più una rete 1 Gbps utilizzata per il monitoraggio e per un filesystem condiviso tra le macchine, usato per il codice della libreria Hadoop ed i file di configurazione globali.

Il cluster è utilizzato anche per altri scopi oltre alla ricerca su MapReduce, sebbene durante gli esperimenti è stato possibile averne disponibilità esclusiva in termini di tempo macchina. Questo ha però richiesto del lavoro supplementare per renderlo compatibile con i sistemi preesistenti, come illustrato in Appendice B. Inoltre la presenza di questi sistemi preesistenti ha limitato in modo importante lo spazio su disco disponibile per Hadoop.

5.1.2 Configurazione di Hadoop

Dallo studio dettagliato effettuato è emerso come Hadoop sia un sistema molto complesso: il numero di parametri di configurazione è enorme, e permette di regolare molti dei comportamenti interni al sistema. In questa prima fase di sperimentazione, dati i limiti temporali imposti ad un lavoro di tesi, si è scelto di mantenere la configurazione abbastanza in linea rispetto allo standard presente in letteratura tecnica per quanto riguarda i parametri obbligatori, lasciando la maggior parte dei parametri opzionali al valore di default.

Nodi		
Hostname	Componente	
eridano[10-26]	CPU	nehalem i7 950 @ 3.07 GHz (LGA 1366)
	RAM	3 * 4 GB @ 1600MHz tri-channel
	Motherboard	ASUS P6T SE
	Scheda di Rete	10G-PCIE-8B-S (10Gbps)
	Dischi	5 dischi
	S.O.	GNU\Linux Debian 6.0
		kernel 2.6.32 (x86_64)
		java 1.6.0_24 (sun)
		Hadoop 0.20.203 (stable)

Hardware di Rete	
Switch	Fujitsu XG2600 (10Gbps)

Tabella 5.1 – Hardware e Software utilizzato per l'esecuzione degli esperimenti

Infine, vista la taglia medio-piccola del cluster si è preferita una assegnazione aggressiva delle risorse CPU, dedicando un nodo alla sola funzione di master, eseguendo sia jobtracker che namenode.

Numero di slot e spazio di Heap

Seguendo le linee guida presenti in letteratura tecnica, il numero di slot consigliato per macchina è in relazione al numero di core disponibili. La relazione consigliata ha la forma

$$n_{slot} = 2 \cdot (n_{core} - 1)$$

dove un core viene riservato all'esecuzione dei demoni datanode e jobtracker, ed il numero di slot viene poi suddiviso equamente in map slot e reduce slot. Questa configurazione è specificatamente ottimizzata per avere job concorrenti, in cui map task e reduce task hanno il picco di esecuzione sovrapposto. Nel nostro caso, per massimizzare il parallelismo MapReduce disponibile, è stata fatta una scelta

più aggressiva, assegnando direttamente due slot per core. In totale risultano quindi 8 slot per macchina: 4 map slot e 4 reduce slot.

Occorre poi assegnare la memoria RAM sotto forma di memoria heap, tra slot, demoni e sistema operativo. Dati i 16 GB di memoria disponibili per macchina, questi sono stati suddivisi come segue:

- 2000 MB per demone
- 1200 MB per slot
- 2000 MB per il sistema operativo

È stato lasciato un amplio margine per il sistema operativo, data la presenza sul cluster di parecchi servizi di supporto preesistenti non eliminabili.

Configurazione dei dischi

Per ogni nodo, sono state ricavate 5 partizioni (una per disco), senza l'uso di tecnologia RAID, da utilizzare per HDFS e per lo spazio locale utilizzato per i dati intermedi nei round. Nella configurazione sperimentata le partizioni hanno la stessa dimensione e sono condivise per entrambi gli usi. Come verrà messo in evidenza in Sezione 5.4, questo aspetto della configurazione è risultato determinante per definire le capacità di elaborazione di un cluster in termini di valori massimi di M e n. Condividere lo spazio tra dati intermedi e HDFS provoca una complicazione nei limiti imposti dalla memoria su disco (in quanto spazio locale e spazio HDFS sono condivisi), ma permette di massimizzare il troughput utilizzando tutti i dischi sia in fase terminale di ogni job che in fase intermedia.

5.1.3 Resilienza e replicazione

Le versioni implementate sono state provate su tre varianti della configurazione base di Hadoop in relazione ai due meccanismi di resilienza di Hadoop: la replicazione HDFS e l'esecuzione speculativa.

- Default: configurazione di Hadoop base, fornita con la distribuzione. L'esecuzione speculativa è abilitata, mentre la replicazione HDFS assume il valore di default (ossia 3);
- "No resilienza": sia esecuzione speculativa che replicazione sono disabilitate;

Configurazione per Nodo Worker	
Slot e Heap	
map slot	4
reduce slot	4
heap demoni	2000 MB
heap slot	1200 MB
Utilizzo del disco	
spazio locale globale	183.32 GB
spazio HDFS	$183.32/\alpha$ GB
spazio locale per disco	32 GB
Stato globale	
Numero nodi worker	15 [a]
Totale map slot	60
Totale reduce slot	60
Totale spazio HDFS	$2.69/\alpha$ TB

[a] in assenza di guasti

Tabella 5.2 – Il Cluster visto da Hadoop. Con α è indicato il fattore di replicazione di HDFS

- "No replicazione": configurazione intermedia tra le precedenti, in cui l'esecuzione speculativa è abilitata, mentre la replicazione è impostata ad 1. Non fornisce quindi resilienza contro i guasti, ma garantisce tempi di esecuzione più uniformi. È utilizzata come configurazione di riferimento per massimizzare i parametri n ed M elaborabili dal cluster.

5.2 Generazione dei Dati

Un aspetto importante della fase sperimentale riguarda chiaramente i dati utilizzati come input. Non disponendo di dati reali, si per generare gli operandi è stata utilizzata una tecnica mutuata da quella utilizzata per generare i dati nel Benchmark "Terasort" fornito con Hadoop. Questa tecnica sfrutta Hadoop in modo non convenzionale, uscendo dal paradigma ma utilizzando il parallelismo offerto dal cluster durante la fase di map per generare dati.

Alla base di questa tecnica vi è l'utilizzo di un InputFormat personalizzato: esso crea input split fittizi (non associati a nessuna sorgente di dati reali) in numero uguale ai map slot disponibili sul cluster, in modo da generare i dati usando la massima parallelizzazione possibile. L'InputFormat fa in modo che ad ogni split sia associato un numero progressivo p. Il codice associato allo split (anch'esso parte dell'implementazione personalizzata di InputFormat) fa poi in modo da ritornare una unica coppia chiave-valore, dove la chiave corrisponde all'indice p dello split ed il valore alla stringa vuota. Sul cluster vengono quindi eseguiti tanti Mapper quanti sono i map slot, ed ogni Mapper procede a generare la p-esima parte dell'input. Infine, detto $r = n \mod p$, il Mapper associato all'ultimo split procede a generare gli ultimi r record.

Questa strategia di generazione ha il beneficio di creare una distribuzione bilanciata sul cluster: ogni nodo possiede una copia locale di una parte degli operatori anche in assenza di duplicazione HDFS. Non è quindi necessario bilanciare il cluster (vedi Sottosezione 3.3.2). Una ulteriore ottimizzazione potrebbe essere raggruppare i file generati per entrambi gli operatori in un unico sequence file, per ottimizzare il numero di split. Questa strategia è stata adottata nella fase di pre-processing delle versioni ad elevata scalabilità trattate in Sezione 5.5.

Le matrici sono create in forma di MatrixItem, in modo da dare un formato comune simile a quanto descritto nell'articolo di riferimento. Questo permette

inoltre di riutilizzare gli stessi dati per diversi valori di m. Le chiavi sono generate in forma (i, j, k) dove i e j sono gli indici dell'elemento all'interno della matrice e k l'indice di operando (che assume valore 1 per A, 2 per B). Questo permette di bilanciare il lavoro durante la fase di map simulate della versioni a e b. I valori degli elementi delle matrici sono generati casualmente nello spazio degli interi a 32 bit.

5.3 Misure

La realizzazione delle misure è risultata una delle maggiori difficoltà incontrate durante la fase sperimentale. Hadoop produce una grande quantità di misure in modo naturale, divise in due categorie: metriche e jobhistory.

Le metriche sono prodotte a scopo di monitoraggio globale dai vari attori a basso livello (JVM, thread, demoni) ad intervalli regolari. Sebbene si sia tentato all'inizio di accedere a questo livello di dettaglio, risulta molto complesso correlare questi dati con gli altri dati rilevati. Il numero di metriche raccolte, pur in un cluster di medio-piccola dimensione, sono troppo dettagliate per essere analizzate, e richiedono uno strumento apposito. [1] Nella versione di Hadoop attuale inoltre le metriche sono scarsamente documentate.

Con il termine jobhistory si intendono invece i dati forniti da Hadoop alla fine dell'esecuzione di job, memorizzati nella apposita directory su jobtracker. Un primo tipo di questi dati sono i contatori: si tratta di misure aggregate per job, quali il numero di coppie (dette anche *record* in questo contesto), il numero di gruppi MapReduce, e le quantità di byte in uscita dai Mapper. Sempre in jobhistory vengono anche esposti degli eventi, corrispondenti ai tempi di avvio e terminazione dei singoli attempt e di alcune fasi intermedie per i reduce attempt. Infine sono poi anche forniti i tempi di invio, inizio elaborazione e terminazione del job in oggetto.

Si è ritenuto utile, in questa fase, aggiungere anche un livello di monitoraggio globale che potesse tenere traccia di tutte le interazioni presenti nel cluster. È stato quindi implementato uno script di monitoraggio a livello sistema operativo che aggreghi ad intervalli fissi (5 secondi) tre misure globali:

[1] Nel caso si volessero esplorare le metriche, lo strumento più promettente è Chuckwa, che integra già questo tipo di funzionalità - vedi Sottosezione 3.1.3.

- l'utilizzo della CPU, misurato in intervalli da 0.01 s [2];
- la comunicazione in uscita, misurata in byte;
- la comunicazione in entrata, misurata in byte.

Queste informazioni sono prelevate direttamente dal kernel (via /etc/proc) e scritte su un file testuale per ogni host.

Infine, poiché Hadoop non riconosce il concetto di algoritmo multi-round, all'interno del codice sono state tenute delle tracce temporali per gli algoritmi terminati con successo, con la versione, i relativi valori dei parametri e la configurazione di riferimento.

Correlare questi dati si è rivelato un compito estremamente complesso, anche evitando di prendere in considerazione le metriche. Infatti si è interessati a controllare ed esplorare le relazioni multi-parametro a vari livelli di aggregazione; inoltre non è noto esattamente il fenomeno che si vuole misurare: data la lunghezza degli esperimenti, essi non possono essere riproposti per analizzare fenomeni diversi. È poi necessario mettere in relazione dati temporali, aggregando i vari job a livello algoritmo. Infine la quantità di dati, essendo relativi ad un sistema complesso e distribuito, è molto elevata, rendendo sconsigliabile una elaborazione manuale.

Non avendo trovato alcun software disponibile adatto allo scopo, è stato sviluppato un pacchetto software che potesse aiutare nel correlare ed interpretare i dati raccolti, tramite l'utilizzo di una base di dati ed alcuni parser. La struttura di questo pacchetto è descritta brevemente in AppendiceC.

Infine esiste un problema di definizione di durata per una trasposizione di MR-algoritmo: essa può essere misurata sia come somma dei tempi di esecuzione dei job (memorizzato come contatore), sia come come il tempo misurato dal driver. In fase sperimentale si è visto che le due misure sono relativamente simili a meno di un errore di misura intorno a ± 20 s. Questa differenza è imputabile a diversi fattori: il jobtracker comunica al driver il termine del job prima di aver eliminato le risorse condivise da HDFS (caricate dal driver in fase di setup - vedi Sottosezione 3.4.7), ma registra nel tempo di esecuzione del job anche questo intervallo di tempo. Al contrario, il tempo misurato dal driver comprende anche

[2] La misura viene fornita in *jiffies*, che corrispondono alla risoluzione dell'orologio software. Con il kernel in uso, la risoluzione è quella indicata

il tempo necessario ad eseguire le operazioni su HDFS. Per gli scopi di questa Tesi, ragionando a livello multi-round, risulta più coerente considerare il tempo utente misurato a livello driver.

5.4 Limitazioni riscontrate ai valori dei parametri

All'inizio della sperimentazione è risultato evidente come non tutti i valori dei parametri n, m e M fossero sperimentabili, pur essendo entro ad un intervallo intuitivamente sensato. Nella definizione del piano di esperimenti è stato quindi necessario determinare i valori massimi per i quali una esecuzione potesse terminare con successo date le risorse disponibili su cluster. Ci si è quindi occupati di determinare le relazioni dettagliate tra questi parametri e le risorse utilizzate, cercando poi di determinarne i valori.

Spazio di Heap e m

In modo evidente, esiste una relazione tra la memoria consumata in esecuzione dai Reducer ed M. Di conseguenza, la dimensione m massima elaborabile è in relazione con lo spazio di heap configurato per gli slot. Poiché durante l'esecuzione vengono memorizzate ed elaborate 3 matrici di interi (rappresentati in Java da 4 byte) di taglia m, detto H lo spazio di heap in byte si ha che

$$m \leq \frac{H}{(4 \cdot 3)}$$

In termini di potenza di due, il valore massimo sperimentabile con la configurazione del cluster dove $H = 1200MB$ si attesta a $m = 2^{24}$. In realtà esso è più basso ($m = 2^{22}$), a causa della rappresentazione come BlockRealMatrix che utilizza valori double nella fase di calcolo. Questo risultato è confermato per via sperimentale: i job eseguiti con valori di m superiori terminano a causa di un fallimento della JVM relativo all'esaurimento dello spazio di heap.

Spazio aggregato minimo su disco

Il parametro M si è visto essere limitato dallo spazio aggregato sui dischi del cluster. Anche effettuando le cancellazioni dei dati intermedi (vedi 4.2.1) si è riscontrato infatti che lo spazio è la risorsa limitante riguardo alle dimensioni di n e M elaborabili, anche se in modo non immediato. In caso di sovrastima di M il comportamento di un job è pessimo: esso non termina con errore ma tende a bloccare il cluster. È quindi necessario determinare i valori delle costanti che determinano la quantità di dati necessarie al corretto svolgimento dei job per

ciascuna versione, prima considerando le varie strategie, poi le specifiche differenze e lo spazio occupato dall'input.

Lo spazio su disco è utilizzato sia da HDFS, sia per memorizzare i dati intermedi in uscita dai map attempt ed in entrata dei reduce attempt (spill - vedi 3.4.7). Nella configurazione di default di Hadoop in particolare tutti i dati recuperati dai reduce attempt sono scritti su disco prima di iniziare la fase di ordinamento, e non sono eliminati fino a che il job non è completato.

Nella configurazione utilizzata, lo spazio utilizzato per gli spill e lo spazio utilizzato per HDFS sono sovrapposti - vedi 5.1.2. Nel caso non lo fossero, queste due componenti vanno considerate separatamente.

Definito S_{max} lo spazio massimo occupato dalla singola implementazione ed S_{local} lo spazio presente su una macchina, esiste quindi una limitazione che può essere espressa come

$$S_{max} < \sum S_{locale} = S_{locale} \cdot n_{datanode} \tag{5.1}$$

dove S_{max} comprende tutti i dati presenti in HDFS durante l'esecuzione ed i dati intermedi presenti sui nodi, e si considerano tasktracker e datanode co-locati, lo spazio HDFS sovrapposto allo spazio per i dati intermedi e si tralascia l'overhead di rappresentazione nei sequence file dovuto ai marcatori sync. Procediamo quindi a stimare S_{max} per le varie versioni, a partire dalle rappresentazioni Writables.

Definiamo B_n lo spazio in byte necessario per memorizzare in Hadoop una matrice $\sqrt{n} \times \sqrt{n}$ comprese le chiavi. Ovviamente questo valore dipende dalle classi Writables utilizzate. Nel nostro caso si ha quindi

$$B_n^i = 25n$$

$$B_n^b = n\left(5 + \frac{13}{m}\right)$$

dove B_n^i è il valore di B_n nel caso si usino MatrixItem, mentre B_n^b è il valore nel caso di usino MatrixBlock. Si può facilmente notare che per ogni valore di m, $B_n^i > B_n^b$, poiché la rappresentazione per item deve tenere traccia della posizione del singolo elemento nel blocco di taglia $\sqrt{m} \times \sqrt{m}$.

Procediamo ora a calcolare lo spazio richiesto da un job per le varie strategie di replicazione, senza contare lo spazio occupato dall'input globale. Consideriamo ora la strategia S1: lo spazio necessario durante un round moltiplicativo diverso dal primo è composto dalle tre matrici (A,B e le somme parziali C) replicate

K volte presenti su HDFS in ingresso al job, all'uscita dei map attempt ed in ingresso dei reduce attempt. Si ha quindi che lo spazio occupato durante un round moltiplicativo intermedio è:

$$6KB_n(\alpha+1) = \underbrace{3KB_n\alpha}_{input\ su\ HDFS} + \underbrace{3KB_n}_{uscita\ map} + \underbrace{3KB_n}_{ingresso\ reduce} + \underbrace{3KB_n\alpha}_{output\ su\ HDFS} \tag{5.2}$$

dove α è il fattore di replicazione di HDFS. Nel caso del primo round moltiplicativo, lo spazio richiesto è minore, in quando non sono presenti le somme parziali. Questo vale anche per l'ultimo round moltiplicativo, essendo prodotte su HDFS alla fine del job solo le somme parziali. Nel caso venga effettuato un solo job moltiplicativo i requisti di spazio sono quindi minori, crescono nel caso di due round e si attestano al valore sopra indicato nel caso avvengano più di due round. Questi contributi possono essere calcolati in modo totalmente analogo. Nel complesso, per la strategia S1 si ha quindi

$$f(B_n) = \begin{cases} 3KB_n(\alpha+4/3) & \text{se } \frac{\sqrt{n}}{\sqrt{mK}} = 1 \\ 4KB_n(\alpha+2/3) & \text{se } \frac{\sqrt{n}}{\sqrt{mK}} = 2 \\ 6KB_n(\alpha+1) & \text{se } \frac{\sqrt{n}}{\sqrt{mK}} > 2 \end{cases} \tag{5.3}$$

Nel caso invece sia utilizzata la strategia di replicazione S2, prendendo in considerazione un round intermedio, le matrici di input A e B vengono lette e replicate dai Mappers ma non vengono scritto al termine dei job, contribuendo solo alla fine dei map attempt e reduce attempt, e l'unico contributo replicato K volte scritto su HDFS corrisponde alle somme parziali. Si ottiene quindi

$$2KB_n(\alpha+3) = \underbrace{KB_n\alpha}_{input\ su\ HDFS} + \underbrace{3KB_n}_{uscita\ map} + \underbrace{3KB_n}_{ingresso\ reduce} + \underbrace{KB_n\alpha}_{output\ su\ HDFS} \tag{5.4}$$

dove, in analogia con la formula per S1, non viene conteggiato lo spazio richiesto per l'input iniziale, che verrà conteggiato separatamente nelle formule finali per le versioni. In caso venga effettuato un solo round il contributo dell'input nella formula è nullo, in quanto già conteggiato, ed il numero di byte necessari per la moltiplicazione diventa $KB_n\alpha + 6KB_n$. Nel complesso quindi diventa

$$g(B_n) = \begin{cases} KB_n\alpha + 6KB_n & \text{se } \frac{\sqrt{n}}{\sqrt{mK}} = 1 \\ 2KB_n(\alpha+3) & \text{se } \frac{\sqrt{n}}{\sqrt{mK}} > 1 \end{cases} \tag{5.5}$$

Consideriamo infine la somma parallela. Nel primo round in input vengono lette K somme parziali da HDFS, scritte come dati intermedi ed elaborate nei

reduce attempt, emettendo in uscita K/m somme parziali. Nel primo round si ha che lo spazio richiesto è quindi:

$$2KB_n + KB_n(\alpha + 1/m) = \underbrace{KB_n\alpha}_{input\ su\ HDFS} + \underbrace{KB_n}_{uscita\ map} + \underbrace{KB_n}_{ingresso\ reduce} + \underbrace{K/m \cdot B_n\alpha}_{output\ su\ HDFS} \tag{5.6}$$

Nel caso fossero poi necessari più round di somma, lo spazio richiesto è banalmente minore.

Andiamo ora a considerare le versioni implementate. Per tutte le versioni disponibili, l'input è sempre rappresentato sotto forma di MatrixItem, e non viene eliminato durante l'esecuzione del driver. Ad esso quindi corrisponde un contributo fisso $2B_n^i$. Per le versioni a, c ed e si ha semplicemente che

$$S_{max}^a = S_{max}^c = 2KB_n^i + f(B_n^i) \tag{5.7}$$

$$S_{max}^e = 2KB_n^i + g(B_n^i) \tag{5.8}$$

dato che in entrambi i requisiti in spazio della somma parallela sono minori rispetto a quelli dei round moltiplicativi. Nel caso delle versioni che utilizzano invece MatrixBlock, gli operatori vengono convertiti tramite un job che ha requisiti di spazio bassi ($4B_n^i + 2KB_n^b\alpha$ nella versione b e $4B_n^i + 2B_n^b\alpha$ per versioni c e f), che sono banalmente soddisfatti. L'input, convertito in blocchi, viene mantenuto dalla versione f per tutta la durata della fase moltiplicativa. Nelle versione b e d viene invece trattato come dato temporaneo. Lo spazio richiesto per la conversione in item alla fine è banalmente incluso nei requisti della somma per item, che deve essere considerata esplicitamente. Si ottiene quindi:

$$S_{max}^b = S_{max}^d = 2B_m^i\alpha + \max(f(B_n^b); 2KB_n^i + KB_n^i\alpha(1 + 1/m)) \tag{5.9}$$

$$S_{max}^f = 2B_n^i\alpha + \max(2B_n^b\alpha + g(B_n^b); 2KB_n^i + KB_n^i\alpha(1 + 1/m)) \tag{5.10}$$

Essendo funzioni multi-parametro complesse, per limiti di tempo non si è ritenuto opportuno determinarne una caratterizzazione completa. I valori di S_{max} sono stati calcolati per alcuni valori di M e n con l'ausilio di un foglio elettronico, allo scopo di determinare i valori dei parametri accettabili su cui effettuare gli esperimenti. Sono stati considerati eseguibili solo le versioni con limite inferiore ai 2400 GB, stimando al 10% l'overhead dovuto ai marcatori sync dei sequence file (vedi 3.4.5). In Tabella 5.3 sono riportati alcuni valori di esempio per tutte le versioni.

m	S^a_{max}	S^b_{max}	S^c_{max}	S^d_{max}	S^e_{max}	S^f_{max}	$S^{d'}_{max}$	$S^{f'}_{max}$
2^2	2800	842	2800	842	2000	700	809	561
2^4	2800	663	2800	663	2000	663	570	395
2^6	2800	653	2800	653	2000	653	510	354
2^8	2800	651	2800	651	2000	651	495	343
2^{10}	2800	650	2800	650	2000	650	491	341
2^{12}	2800	650	2800	650	2000	650	490	340
2^{14}	2800	650	2800	650	2000	650	490	340
2^{16}	2800	650	2800	650	2000	650	490	340
2^{18}	2800	650	2800	650	2000	650	490	340
2^{20}	2800	650	2800	650	2000	650	490	340
2^{22}	2800	650	2800	650	2000	650	277	340
2^{24}	1800	650	1800	650	1800	650	290	290

Tabella 5.3 – Tabella di esempio con i valori di S_{max} espressi in GB per le varie implementazioni fissati $n = 2^{30}$ e $M = 2^{33}$ e replicazione sul HDFS $\alpha = 1$. Con d' ed f' sono indicate le versioni ottimizzate introdotte alle fine della sezione.

Spazio locale minimo su disco

Anche riguardo alla spazio disponibile su una singola macchina, sono stati individuati anche dei requisiti non banali necessari per la corretta esecuzione di una implementazione.

Il primo riguarda lo spazio locale necessario per memorizzare i dati in ingresso nella fase di reduce, ed è legato al numero di partizioni Hadoop. Ogni reduce attempt infatti prova a recuperare una partizione copiandola sulla parte di disco locale. Affinché il reduce attempt abbia successo è necessario che lo spazio locale su un singolo mount point della macchina sia abbastanza capiente. In Hadoop, questa parte di disco contiene in ogni caso i dati in output dai map attempt locali e, nella configurazione consigliata, anche una frazione dello spazio utilizzato HDFS. Considerando i map attempt equamente distribuiti, si ha

$$S_{partizione} + \frac{S_{map\ output\ tot} + S_{hdfs}}{n_{nodi} \cdot n_{mount\ point}} < \max(S_{mount\ point}) \tag{5.11}$$

Durante l'ultima fase della sperimentazione, è stata poi anche identificata una ulteriore limitazione sui map attempt: durante la fase di unione ed ordinamento alla fine di un map attempt, indipendentemente dalla configurazione, i dati prodotti da ogni attempt vengono uniti in un unico file. Lo spazio su disco locale su un singolo mount point deve quindi essere sufficiente a contenerlo. Nelle implementazioni considerate, supposti i map attempt equamente distribuiti, questa limitazione si può esprimere come

$$\frac{S_{interm}}{n_{map\ task}} + \frac{S_{hdfs}}{n_{nodi} \cdot n_{mount\ point}} < \max(S_{mount\ point}) \tag{5.12}$$

dove il secondo contributo è dovuto alla sovrapposizione tra spazio locale e spazio HDFS. Si noti che i due requisiti coincidono se il numero di reduce slot coincide con il numero di task; aumentando invece il numero di task, la taglia di una partizione può diminuire a piacere, rilassando il vincolo espresso in formula 5.11.

Considerazioni sui vincoli

Tutti i vincoli espressi in questa sezione, sebbene abbiano una valenza importante nel determinare i limiti fisici imposti dall'hardware, sono fortemente dipendenti dai dettagli dell'implementazione Hadoop; di conseguenza, sono fortemente dipendenti dalla versione utilizzata. Anche per questo motivo non si è ritenuto opportuno esplicitarli nel dettaglio in questa fase di studio, tenendo presente

che è alle porte una nuova versione di Hadoop, con una completa riscrittura dei demoni e della fase di shuffle (vedi Sottosezione 3.1.2 a pag. 31)

5.5 Versioni altamente scalabili

Alla luce di quanto individuato in Sezione 5.4, si è voluto creare due versioni che ottimizzassero il valore di S_{max} minimizzandolo. Queste due versioni sono dette *ad alta scalabilità* poiché, date le risorse del cluster, permettono di raggiungere i valori di n e M più elevati.

Si è visto che il principale contributo a S_{max} è dovuto all'implementazione MatrixItem. Per questo motivo si sono prese come base le due versioni a blocchi migliori d ed f, creando due versioni ottimizzate dette d' e f'. La minimizzazione di S_{max} è stata effettuata da una parte tramite una fase di pre-processing (calcolata a parte) che riduce l'input in MatrixBlock, dall'altra sostituendo la somma per elementi con una somma direttamente per blocchi simile a quella presentata in Pseudocodice 4 a pagina 63.

Queste versioni sono state identificate come di livello 3, dato che utilizzano per la somma un algoritmo teoricamente inaccettabile rispetto ai vincoli del modello.

Si ha quindi

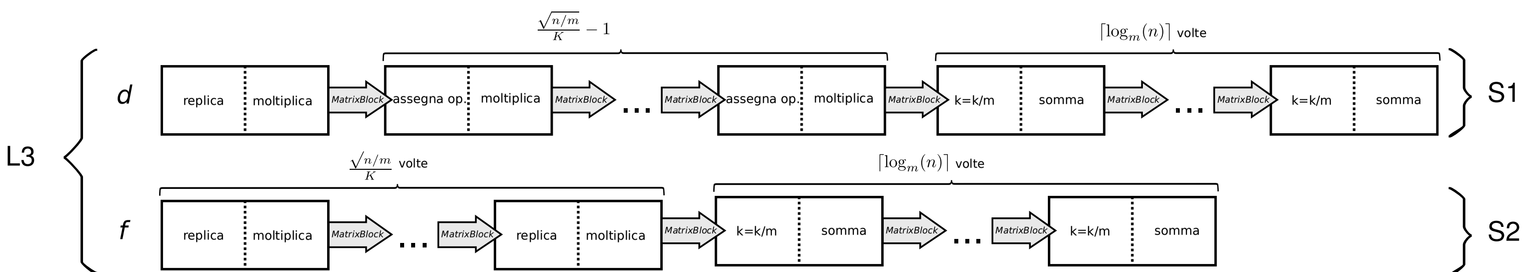
- d', analogo a d a meno dei round di conversione e della somma parallela, che viene effettuata per blocchi;

- f', analogo a f a meno dei round di conversione e della somma parallela, che viene effettuata per blocchi;

Per cui si ha che

$$S_{max}^{d'} = 2B_m^b\alpha + f(B_n^b) \tag{5.13}$$

$$S_{max}^{f'} = 2B_n^b\alpha + g(B_n^b) \tag{5.14}$$



Versione	Livello	Descrizione	Strategia	Classe Valori	Classe Driver	Complessità in round al caso pessimo
d'	3	Versioni massima scalab.	S1	MatrixBlock	L3_MapPipeBlockDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil$
f'			S2	MatriBlock	L3_SimpleBlockDMM	$\frac{n^{3/2}}{M\sqrt{m}} + \lceil \log_m(n) \rceil$

Figura 5.1 – Struttura delle implementazioni ottimizzate, raggruppate per livelli (a sinistra) e per strategie di replicazione (a destra). A seguire, la tabella riassuntiva della caratteristiche e la complessità in numero di round

5.6 Risultati

Come spiegato in introduzione a questo capitolo, il numero di parametri da considerare durante la sperimentazione si è rivelato molto elevato. Infatti, oltre ai parametri n, M e m da cui naturalmente dipende l'algoritmo, molti dei parametri di configurazione di Hadoop potrebbero avere un impatto importante sull'andamento dei tempi di esecuzione. Durante la sperimentazione si sono poi verificati dei guasti hardware, che hanno ostacolato lo svolgimento di una indagine completa sui valori dei parametri principali.

Poiché i tempi di esecuzione si sono rilevati importanti, specie per m piccoli, e dato l'elevato numero di versioni, è stato preferito privilegiare un intervallo il più possibile elevato di valori, a scapito della precisione sul singolo valore. Questo fatto è giustificato in quanto lo scopo è identificare gli andamenti. I dati raccolti infatti vanno interpretati come dati preliminari: essi corrispondono ad una singola esecuzione delle varie versioni, e possono perciò risentire di fenomeni transitori, tipici di un sistema ad elevata complessità. Come risulterà durante la presentazione dei risultati, i dati risultato abbastanza consistenti in termini di andamenti, ed i risultati coincidono sostanzialmente nelle varie serie, quando i parametri collimano.

In fase preliminare è stata individuata come taglia adeguata dell'input da dove iniziare a provare le versioni $n = 2^{24}$, che permette di controllare l'andamento del tempo per tutto l'intervallo di m e M senza incorrere nelle limitazioni descritte in Sezione 5.4. Per m piccoli è stato preferito campionare evitando i valori 2^6 e 2^{10} a causa dell'elevato tempo di esecuzione. Questa serie è poi stata ripetuta in configurazione 'no resilienza' per osservare l'impatto della replicazione a livello HDFS e dell'esecuzione speculativa sui tempi di esecuzione. Si noti che questa taglia di n può essere effettuata anche sequenzialmente su una delle singole macchine del cluster dedicandovi l'intera memoria RAM. Utilizzando Java ed un algoritmo analogo a quello utilizzato nei Reducer, il tempo di esecuzione sequenziale è risultato dell'ordine dei secondi. La maggior parte del tempo riscontrato sarà quindi essenzialmente dovuto all'overhead necessario per l'esecuzione di MapReduce, e rappresenta una sorta di limite inferiore nelle prestazioni. I risultati di questa esecuzione sono presentati nella Sottosezione 5.6.2.

Come spiegato nella Sezione 5.4, si sono riscontrati dei problemi di scalabilità in termini di n a causa delle limitate risorse disponibili. Questi problemi sono

concentrati nelle versioni che utilizzano MatrixItem, che soffrono una cospicua penalizzazione dovuta alla grande quantità di dati necessaria per rappresentare le chiavi. Si è quindi tentato di sperimentare delle matrici di taglia più elevata, più vicina ad un caso d'uso reale, utilizzando le versioni ad alta scalabilità, presentate in Sezione 5.5 per m vicini al valore massimo possibile dalle risorse del cluster, come suggerito dalla serie precedente. Durante la sperimentazione però tre nodi del cluster si sono guastati, e non è stato possibile sostituirli, limitando ulteriormente lo spettro dei parametri. I risultati di questi esperimenti ottenuti a cluster ridotto sono riportati in Sezione 5.6.3.

In Tabella 5.4 sono riportate le serie effettuate.

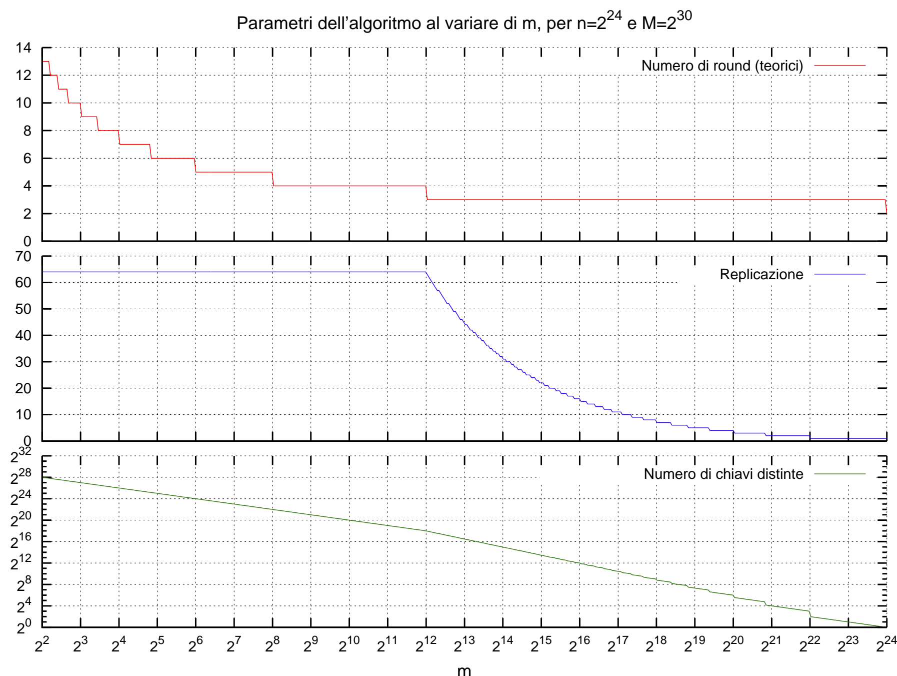
Serie	n	M	m										
			2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
1	2^{24}	2^{30}	✓		✓		✓	✓	✓	✓	✓	✓	✓
2	2^{30}	2^{32}									✓	✓	
3	2^{32}	2^{32}									✓	✓	
4	2^{32}	2^{35}									✓	✓	

Tabella 5.4 – Tabella riassuntiva degli esperimenti eseguiti. Alcune combinazioni non sono state eseguite a causa dell'eccessivo numero di job di cui erano composte o dei limiti di scalabilità espressi in Sezione 5.4

5.6.1 Il numero di round, il numero di chiavi distinte ed il fattore di replicazione

Durante l'analisi ed interpretazione dei dati è risultato evidente come le prestazioni dell'algoritmo sono influenzate da tre variabili principali, dipendenti da m,M en:

- il fattore di replicazione (K), che descrive la quantità di dati elaborata in un round. Come illustrato nel Capitolo 4, esso assume il valore $K = \min(M/n, \sqrt{n/m})$;
- il numero di round, corrisponde al numero di job a meno di un contributo additivo costante (vedi Tabella 4.2). Assume il valore $R = \sqrt{n}/(K\sqrt{m}) + \lceil \log_m(n) \rceil$;



	m								
	2^4	2^8	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
K	2^6	2^6	2^6	2^5	2^4	2^3	2^2	2^1	2^0
R	8	5	4	3	3	3	3	3	2
$\|U_r\|$	2^{26}	2^{22}	2^{18}	2^{15}	2^{12}	2^9	2^6	2^3	1

Figura 5.2 – L'andamento dei principali fattori che influenzano le prestazioni, al variare di m per $n = 2^{24}$ e $M = 2^{30}$.

- il numero di chiavi distinte durante la moltiplicazione, ossia il numero di gruppi MapReduce. Come descritto in Sottosezione **??**, esso corrisponde al massimo parallelismo possibile. Esso assume il valore $|U_r| = n/m \cdot K$.

Esse rappresentano per molti aspetti la chiave nell'analizzare i tempi di esecuzione, come mostrato nelle prossime sottosezioni. Per ciascuna serie sperimentale, il loro andamento e lo specifico valore in corrispondenza dei valori di m sperimentati sono illustrati per la serie completa in Figura 5.2,

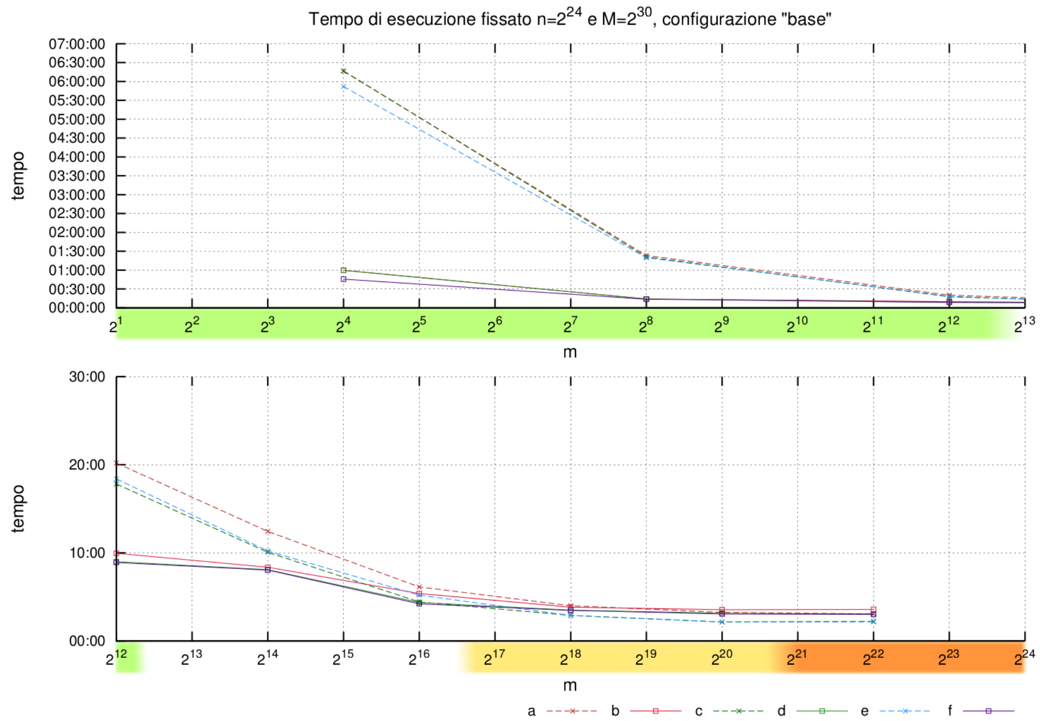


Figura 5.3 – Tempi di esecuzione globale al variare di m, per $n = 2^{24}$ e $M = 2^{30}$. In rosso è evidenziata la zona in cui il cluster è sottoutilizzato, in giallo la zona in cui vengono eseguiti due round, in verde la zona per K costante.

5.6.2 Serie 1 ($n = 2^{24}$,$M = 2^{30}$)

Tempo di esecuzione rispetto a m

Come si può notare dal grafico dei tempi di esecuzione, riportato in Figura 5.3, il tempo necessario per completare la moltiplicazione matriciale è fortemente dipendente da m.

Per m piccoli i tempi di esecuzione delle versioni che utilizzano MatrixBlock (versioni b,d,f) risultano molto minori rispetto alle altre versioni. Come messo in evidenza precedentemente, anche per il più piccolo valore di m sperimentato ($m = 2^4$) il numero di bit utilizzato per rappresentare un blocco di taglia m è

maggiore di circa un fattore 4 per MatrixItem:

$$\begin{aligned} \text{MatrixItem:} \quad & 25m & = 400 \text{ bit} \\ \text{MatrixBlock:} \quad & 12 + (5m + 1) & = 93 \text{ bit} \end{aligned}$$

Poiché per questi valori di m il fattore di replicazione è elevato, la differenza in termini di quantità di dati scambiati rimane importante come valore assoluto. Permane inoltre un fattore m di differenza nel numero di chiavi presenti durante la fase di shuffle, a vantaggio delle versioni con MatrixBlock. Utilizzando MatrixBlock infatti il numero di chiavi è pari al più a $3|U_r| = 3nK/m$, contro $3nK$ per MatrixItem. Questo comporta un maggior lavoro durante la fase di ordinamento presente nello shuffle per le versioni con MatrixItem. Entrambi questi fattori hanno una forte ripercussione sul tempo medio per job durante la moltiplicazione, come riportato in Figura 5.4.

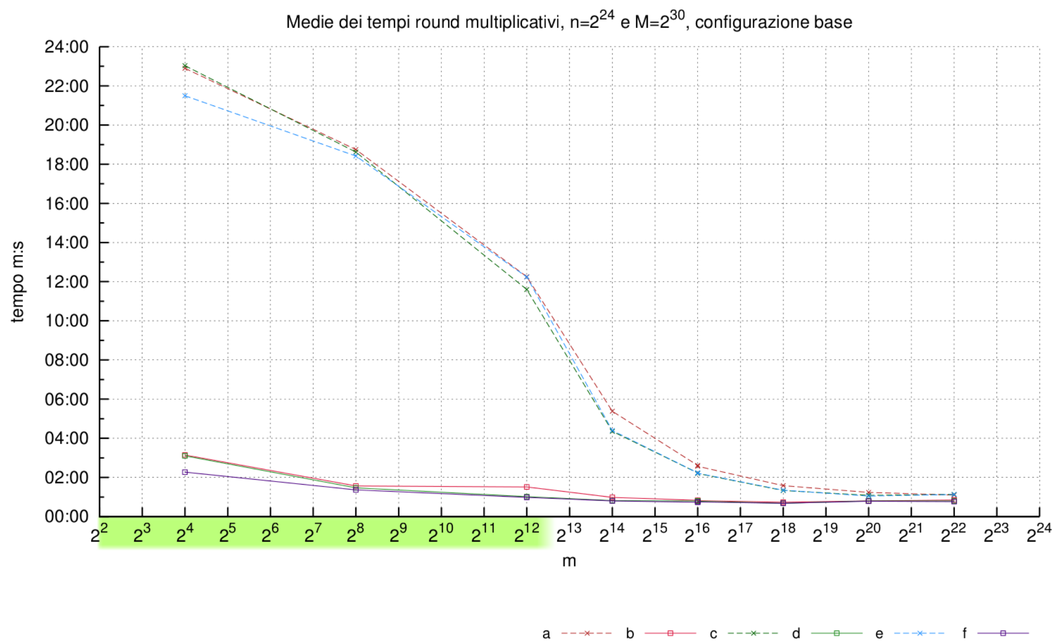


Figura 5.4 – Tempi di esecuzione medio per job al variare di m, per $n = 2^{24}$ e $M = 2^{30}$. Sono evidenziati in verde i valori di m per cui l'indice di replicazione K è costante ($K = 64$).

Al decrescere di m cresce inoltre il numero di round. L'insieme di questi due fattori spiega la differenza in termini di tempo riscontrata nei dati sperimentali.

Osservando il tempo medio per round (Figura 5.4) si nota come il tempo medio aumenti all'aumentare di m pur con fattore di replicazione K costante. La principale spiegazione è dovuto all'aumentare del numero di chiavi distinte ($|U_r|$). Come spiegato in apertura di questo capitolo, questo numero corrisponde al numero di gruppi MapReduce: questo ha un impatto sui tempi dovuto al maggior lavoro necessario in fase di shuffle, ed all'overhead dovuto al maggior numero di chiamate al metodo reduce. Infine all'aumentare della media contribuisce il fatto che l'aumentare del numero di round diminuisce l'impatto del primo job moltiplicativo, che deve elaborare un numero minore di coppie (non sono presenti somme parziali). Dai dati sperimentali emerge infatti che il primo round moltiplicativo impiega infatti circa due terzi del tempo rispetto agli altri job moltiplicativi. Si noti che nel caso in esame ($n = 2^{24}$, $M = 2^{30}$) per $m > 12$ viene effettuato un solo job moltiplicativo.

All'altro estremo del grafico ($m = 2^{22}$), due versioni che utilizzano MatrixItem (c,e) hanno il tempo di esecuzione minore. Questo è dovuto essenzialmente al minor numero di round globali delle implementazioni, come illustrato in Tabella 4.2. Infatti pur mantenendo il tempo medio per round più elevato a causa del maggior numero di dati scambiati, questo non è il contributo dominate a causa del basso fattore di replicazione e numero di round. L'assenza del fattore costante additivo per queste due versioni porta infatti ad avere due soli job, contro i 3 o 4 job necessari per le altre versioni.

Per tutti i valori di m, i tempi di esecuzione dei round "solo map" sono estremamente contenuti: il massimo tempo di esecuzione per un round "solo map" si riscontra per $m = 2^4$, con un tempo di esecuzione dell'ordine del minuto. Da sottolineare che il compito svolto dai round "solo map" è la conversione delle K somme parziali da MatrixBlock a MatrixItem, cui corrisponde un elevato numero di coppie in uscita.

Infine nel caso degenere ($m = 2^{24}$) lo spazio di heap allocato per map slot non risulta sufficiente per eseguire la moltiplicazione matriciale, che fallisce per mancanza di spazio in tutte le sue versioni.

Peso delle varie fasi

Come spiegato in Sezione 3.5, risulta difficile riconoscere in modo esplicito le varie fasi (map, shuffle e reduce) in Hadoop. È però possibile provare a stimare la fase di reduce, facendo corrispondere la fase di reduce per ciascun attempt con il tempo compreso tra la fine dell'ordinamento e la fine del job. Come introdotto in Sezione 5.3 Hadoop fornisce un evento di fine ordinamento. Poiché il numero di attempt in condizioni normali è pari al numero di slot, tutti gli attempt sono eseguiti in parallelo. I reduce attempt inoltre iniziano ad ordinare i dati solo dopo che tutti i map task sono completati: supponendo il carico distribuito in modo uniforme grazie all'HashPartitioner, è quindi giustificato aspettarsi che tutti i reduce attempt terminino di ordinare i dati circa contemporaneamente. Questa considerazione trova supporto nei dati sperimentali: per tutti i job considerati, tutti gli eventi di fine sort sono compresi in un intervallo di circa 10 secondi. Fanno eccezione alcune esecuzioni, in cui un singolo attempt è fuori da questo intervallo, probabilmente a causa dell'esecuzione speculativa.

Partendo da questo assunto, è possibile provare a stimare l'impatto della fase di reduce in ciascun job definendo una funzione I_{reduce}:

$$I_{reduce}(j) = \frac{\sum_{a \in S_{ra}} (t_{endsort}(a) - t_{endjob}(j))}{|S_{ra}|(t_{endjob}(j) - t_{startjob}(j))}$$

dove S_{ra} è l'insieme dei reduce attempt terminati con successo, $t_{endsort}(a)$ il timestamp relativo agli eventi di fine ordinamento per l'attempt a, $t_{endjob}(j)$ e $t_{startjob}(j)$ sono i timestamp di inizio e fine del job j. La funzione I_job è quindi una stima dell'impatto percentuale del tempo di reduce sul tempo totale di un job.

Per ciascuna esecuzione delle varie versioni è stata poi calcolata la media di I_{job} per i round moltiplicativi, ed è riportata sotto forma di grafico in Figura 5.5.

Osservando i valori per le varie versioni, si può notare come i contributi siano per la maggior parte dei valori inferiori al 50%, crescenti al crescere di m. Il comportamento delle versioni che utilizzano MatrixItem per m piccoli può essere spiegato con il crescere del numero di chiamate alla funzione reduce, e la necessità di ricompattare le matrici per eseguire le moltiplicazioni. Non si nota una sostanziale differenza tra le implementazioni che eseguono calcoli nella fase di map (e,f) suggerendo come il tempo sia dominato dalla comunicazione e da altre latenze, essendo il lavoro eseguito in fase di reduce propriamente detta molto

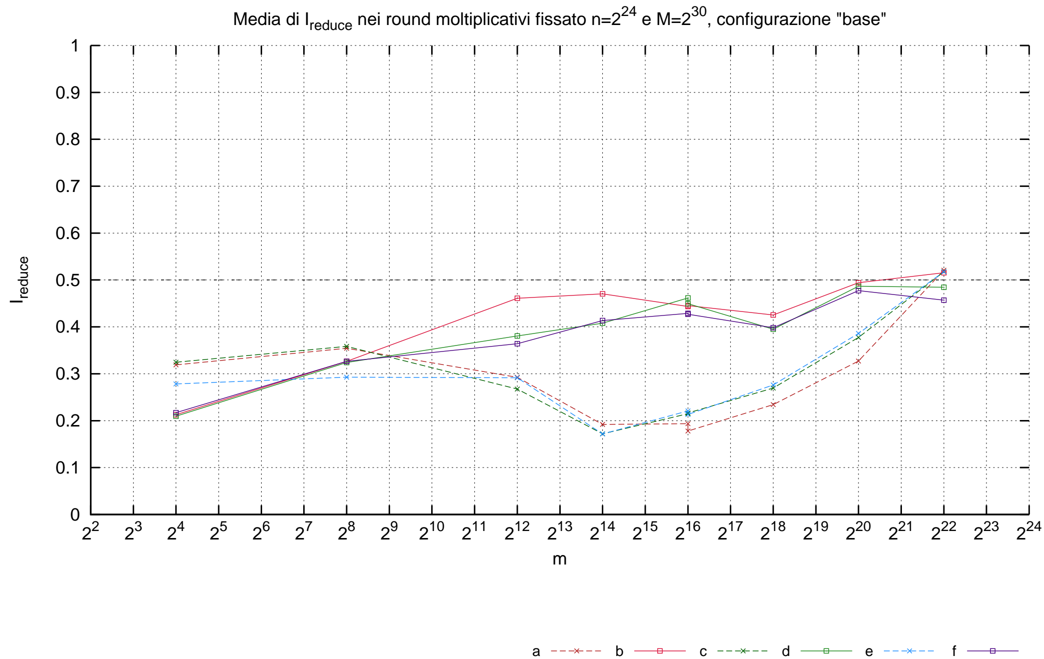


Figura 5.5 – Stima del contributo della fase di reduce al tempo totale tramite la funzione I_{reduce} nei round moltiplicativi

basso a causa della dimensione molto limitata di n. Come ci si aspettava, I_{reduce} non si comporta in modo ordinato: il concetto di fase di reduce in Hadoop si conferma poco determinato e difficile da isolare a livello job.

Effetto della replicazione

Questa serie è stata rieseguita sul cluster in configurazione "no resilienza". Il tempo di esecuzione riscontrato è riportato in Figura 5.6, mentre il tempo medio per round è riportato in Figura 5.7.

Per quanto riguarda l'andamento generale al variare di m, esso è analogo a quello riscontrato nella configurazione base. Per valori di K piccoli, in corrispondenza della parte destra del grafico, i tempi sono equivalenti a quelli della configurazione base. A differenza di quest'ultima però il caso degenere $m = 2^{24}$ termina con successo, con tempi lievemente superiori al caso con $m = 2^{22}$. In alcuni casi si riscontra un tempo leggermente maggiore: questo è probabilmente dovuto a qualche rallentamento nei meccanismi interni del cluster ed alla mancanza di

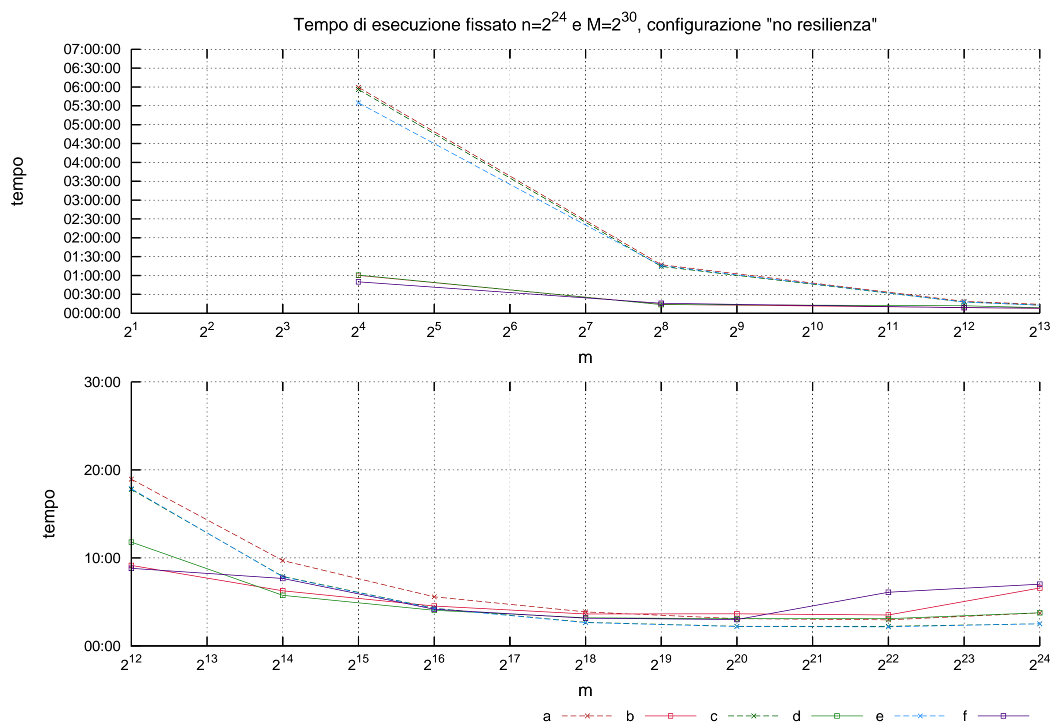


Figura 5.6 – Tempi di esecuzione globale al variare di m, per $n = 2^{24}$ e $M = 2^{30}$, in configurazione "no resilienza".

esecuzione speculativa, che garantisce un tempo minimo in questi casi.

Nello stesso modo, osservando il tempo medio per round, le anomalie di cui sopra sono confermate nel caso $m = 2^{12}$ per la versione d e per lo stesso motivo.

5.6.3 Matrici di grandi dimensioni

Visti i problemi riscontrati con le rappresentazioni a blocchi quando sul cluster sono presenti molti dati, sia in termini di S_{max} che in termini di tempo, si è preferito evitare di sperimentare le versioni puramente ad item sul cluster. Si è quindi tentato di eseguire le versioni a blocchi (b, d, f) nella serie 2, e le versioni ad alta scalabilità (d',f') nella serie 3 e 4, utilizzando per entrambe la configurazione "no duplicazione", ossia con fattore di replicazione per HDFS $\alpha = 1$, per rilassare i vincoli su S_{max}.

Durante la sperimentazione però si sono verificati dei guasti al cluster, perdendo 3 nodi; non è stato possibile, nei tempi ragionevoli per questo lavoro, sostituirli.

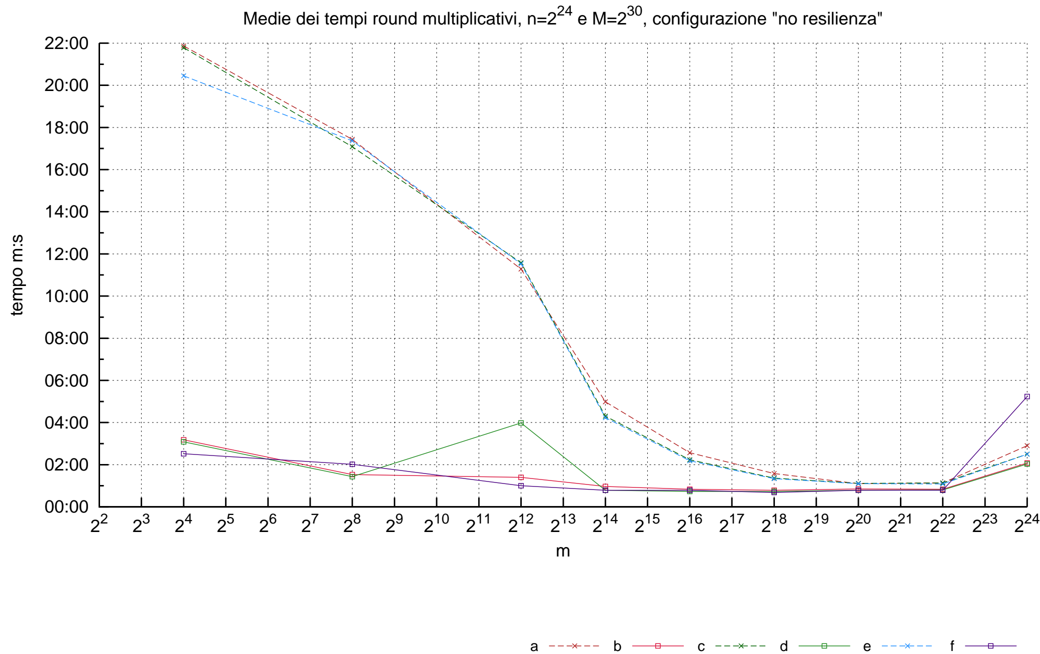


Figura 5.7 – Tempi di esecuzione medio per job al variare di m, per $n = 2^{24}$ e $M = 2^{30}$ in configurazione "no resilienza'.

Questo ha comportato una perdita di potenza di calcolo e soprattutto di spazio di memorizzazione che ha ulteriormente limitato le prove per grandi valori di n.

I risultati ottenuti con le risorse limitate sono riportati in Tabella 5.5. Non tutte le versioni sono state tentate per tutti i valori, poiché i guasti avvenuti hanno interrotto alcune serie che non sono state ritentate per limiti temporali.

Si conferma la tendenza dei tempi a diminuire al crescere di m per gli unici dati completi. Interessante invece notare come per la versione d', a parità di m, al crescere di M aumenta pure il tempo di esecuzione, in contrasto con quanto previsto dal modello. Questo fatto è dovuto all'avvicinarsi del limite di spazio locale per i map attempt identificato in Sezione 5.4: l'output dei Mapper non sempre viene scritto con successo, ed alcuni di essi falliscono con l'errore "no space left on device". Situazione analoga si rivela anche in fase di reduce, dove non tutti i reduce attempt riescono ad allocare spazio sufficiente. La computazione arriva a termine, ma rallentata a causa delle necessarie ripetizioni. Si noti che, in teoria, questo non dovrebbe succedere, poiché i dati dovrebbero essere distribuiti

in modo uniforme nei mount point.

Per quanto riguarda invece le versioni provate nella serie 2, esse vanno in stallo a causa del vincolo di memoria locale, ed in particolare a causa della somma per elementi. Durante la somma, poiché è utilizzata la rappresentazione a MatrixItem, alla fine della fase di map tutti i mount point su tutti i dischi risultano parzialmente occupati in modo tale che non vi è spazio per recuperare nessuna partizione MapReduce. Il job quindi rimane sospeso tra la fase map e la fase di reduce, non riuscendo ad allocare le risorse necessarie per completare.

Nella serie 4 invece appare un fenomeno di cui non si è riusciti a capire l'origine: la versione f' fallisce con l'errore "Java Heap exception" per tutti i map attempt, nonostante non vi sia alcun accumulo nei Mapper. A livello intuitivo, questo è dovuto a qualche buffer all'intero del codice di gestione dei map attempt, ma per limiti di tempo non è stato possibile indagare.

Serie	n	M	versione	m	
				2^{20}	2^{22}
2	2^{30}	2^{32}	b	S	S
			d	S	S
			f	S	S
3	2^{32}	2^{32}	d'	7:11:57	4:45:43
			f'		4:27:43
4	2^{32}	2^{35}	d'		5:03:16
			f'	MH	MH

Tabella 5.5 – Risultati parziali ed errori riscontrati nell'eseguire le versioni per le serie con n grande. Gli errori sono dovuti ai limiti delle risorse hardware del cluster.

5.7 Considerazioni sui risultati

L'apparato sperimentale, anche prima dei guasti, è apparso limitato rispetto al compito assegnato. In particolare, lo spazio su disco è stato il fattore limitante. Questo ha però permesso di evidenziare alcuni interessanti vincoli espliciti rispetto ai parametri, che, una volta approfonditi e verificati, possono essere utili in relazione al modello.

Il fatto che la dimensione dei dischi non fosse appropriata non era evidente a priori. L'algoritmo per la moltiplicazione densa infatti si rivela particolarmente aggressivo, generando un forte tradeoff tra parallelismo e quantità dei dati. L'overhead dovuto alle chiavi, in ogni caso, rimane determinante, realizzando per matrici piccole un notevole rallentamento rispetto al caso sequenziale.

La scelta determinante per quanto riguarda i limiti di scalabilità è senza dubbio la condivisione di spazio HDFS e spazio per i dati intermedi (o spazio locale). La scelta portata avanti in questa Tesi, in linea con la letteratura tecnica, ha complicato e reso difficile isolare i vincoli sui parametri.

Somma per blocchi

Interessante in termini di rapporto con il modello è la somma per blocchi: in essa infatti ogni Reducer elabora al più m matrici di taglia m, pur con la dimensione di heap limitata, in contrasto con quanto detto in 5.4. In realtà solo lo spazio di lavoro è limitato da m, e la somma può essere effettuata utilizzando spazio di lavoro costante in m, recuperando una matrice alla volta.

Problemi aperti

A causa dei problemi al cluster ed allo scarso tempo disponibile, il numero di esperimenti effettuati lascia molti problemi aperti. In particolare sarebbe interessante vedere la reazione dei tempi di esecuzione all'aumentare del numero di nodi: in molti casi infatti il parallelismo possibile ($|U_r|$) dell'algoritmo è molto inferiore a quello realmente utilizzato dal cluster.

Un altro problema aperto riguarda il modello di esecuzione di Hadoop: diminuendo il numero di slot per nodo è possibile utilizzare m di taglia maggiore, diminuendo l'overhead relativo alle chiavi. Anche questo fenomeno potrebbe essere interessante riguardo ai tempi di esecuzione. Nello stesso modo, sarebbe

interessante vedere l'andamento del tempo al variare del numero di task dichiarati. Come accennato in precedenza, aumentare il numero di task potrebbe rilassare il vincolo sullo spazio locale introdotto in 5.4, diminuendo la taglia di una partizione.

Per migliorare poi l'utilizzo del disco, sarebbe interessante vedere l'impatto della compressione sia dei dati in uscita dai Mapper sia dei sequence file su HDFS.

Considerando infine i problemi segnalati in Sottosezione 5.6.3 riguardo ai vincoli di memoria locale, si osservi che all'aumentare del numero di reduce task attempt dichiarati (vedi Sezione 3.4.4 a pag.45) aumenta il numero di partizioni, e quindi, a parità di dati, diminuisce la taglia di ciascuna; in questo modo è dunque possibile rilassare i vincoli, si suppone a spesa di qualche overhead.

Conclusioni

Grazie al lavoro svolto, è possibile trarre alcune conclusioni di carattere generale riguardo all'implementabilità ed alle prestazioni dell'algoritmo, ed alcune indicazioni relative al modello.

Implementabilità e Prestazioni

Durante lo svolgimento della Tesi si è visto come l'implementazione Hadoop, allo stato attuale, non comprenda in modo nativo il concetto di algoritmo multi-round, ma riconosca come massimo livello di granularità il job, concetto analogo al singolo round. Questo ha importanti risvolti sulla implementabilità degli algoritmi sviluppati per il modello MR(m,M), sia in termini di prestazioni che di complicazione del codice e difficoltà di debug.

La focalizzione di Hadoop in termini di singoli round penalizza le prestazioni in quanto comporta un uso aggressivo del disco. I dati intermedi tra un round e il successivo vengono trattati come dati finali, la cui replicazione e consistenza è un obbiettivo prioritario del framework. Non solo vengono sempre scritti su disco, ma vengono replicati dalla parte HDFS per garantirne la tolleranza agli errori.

Scrivere algoritmi multi-round su Hadoop ha richiesto poi di spostare una parte della logica all'interno del codice del driver: in questo modo il codice si complica in modo sostanziale, poiché le API della libreria non permettono di incapsulare in modo naturale e veloce il setup dei singoli job; la logica progettuale della libreria infatti è di assegnare al driver un semplice ruolo di configuratore. All'interno del codice del driver diventano anche fondamentali le interazioni con HDFS: poiché i dati tra i round sono trattati come definitivi è necessario eliminarli dal driver per evitare che il loro accumularsi limiti la scalabilità dell'algoritmo.

La presenza di logica applicativa all'interno del driver rende poi importante poter testare anche questo codice nel dettaglio. Hadoop non permette però di iso-

lare questo codice rispetto al resto dell'infrastruttura, rendendo il debug criptico e quasi proibitivo con gli usuali strumenti.

Anche a causa dell'uso intensivo del disco, si è visto come l'overhead provocato dalla rappresentazione delle chiavi giochi un ruolo fondamentale nelle prestazioni, specie per matrici di grandi dimensioni. Se possibile, è perciò da preferire nella implementazione di MR-algoritmi una rappresentazione che faccia uso di m aggregando le coppie destinate allo stesso Reducer, diminuendo il numero di chiavi utilizzate a parità di dati.

Guardando gli scopi del paradigma, sviluppare MR-algoritmi in Hadoop è, allo stato attuale, un compito tutt'altro che semplice anche per la presenza di vincoli complessi sulla scalabilità rispetto ai parametri; questi vincoli sono fortemente dipendenti dai processi interni di Hadoop, e rendono quindi la sua conoscenza dettagliata importante.

Sembra quindi che, allo stato attuale, Hadoop non sia sufficientemente maturo per ottenere prestazioni accettabili per la moltiplicazione tra matrici; questo risultato sembra probabilmente estendibile ad algoritmi multi-round in generale. Rimane comunque necessario eseguire uno studio più approfondito dei parametri di Hadoop e per maggiori dimensioni sia del cluster che dell'input.

Modello

Per quanto riguarda il modello, esso ha dimostrato una certa utilità nello sviluppo nonostante le difficoltà incontrate. In particolare l'uso di parametri m ed M permette di caratterizzare il comportamento dell'algoritmo date le risorse disponibili. L'approccio tradizionalmente presente nella letteratura tecnica è di natura opposta e prevede da una parte di presupporre una sovrabbondanza di risorse rispetto al compito assegnato, dall'altra di evitare di memorizzare elementi all'interno dei Reducer, per evitare problemi con lo spazio di lavoro. Tutto ciò ha come sbocco naturale una rappresentazione per singoli elementi. Per questo motivo, come visto nella moltiplicazione matriciale, l'utilizzo del modello può permettere quindi di trarre vantaggio da una rappresentazione più sintetica grazie all'uso del parametro m.

La funzione di costo utilizzata nel modello, ossia il numero di round, sembra una buona metrica a livello generale, sebbene non prenda in considerazione alcuni aspetti rilevanti come la quantità di dati e il numero di chiavi distinte. Il numero

di round permette di individuare l'andamento dei tempi entro un margine sufficientemente amplio rispetto ai limiti strutturali del cluster. Avvicinandosi però ai vincoli posti dalle risorse effettivamente presenti si è visto come il presentarsi di errori, dovuti ai limiti dei dischi locali, possa provocare il peggioramento sensibile delle prestazioni a fronte di un minor numero di round.

L'assunzione principale del modello, riguardo al peso della fase di shuffle, si è rivelata di difficile verifica a causa delle modalità esecutive di Hadoop, che si differenziano molto rispetto a quelle paradigmatiche. Essa risulta confermata solamente in modo intuitivo dai risultati sperimentali ottenuti con matrici di piccola taglia, in cui il calcolo svolge un ruolo secondario.

Per quanto riguarda l'interpretazione dei parametri, m svolge evidentemente il ruolo dello spazio di lavoro, mentre il meccanismo implementativo di Hadoop svincola i dati recuperati da una singola macchina rispetto a questo parametro. Il parametro M svolge invece un ruolo assimilabile allo spazio su disco globale necessario per un round lontano dai vincoli posti dalle risorse disponibili, nelle cui vicinanze invece diventano dominanti le costanti implementative.

Estendibilità

Questo lavoro trova infine una sua importante giustificazione proiettato verso il futuro. Se infatti i problemi presentatisi con l'apparato sperimentale hanno limitato il numero di esperimenti svolti, l'esperienza accumulata ha permesso di identificare alcuni aspetti critici su cui indagare; infine il codice sviluppato presenta un ottimo punto di partenza: l'astrazione per il debugging sviluppata così come lo strumento di monitoraggio, se pur non strettamente innovativi, sono una utile base per affrontare problemi similari.

Appendice A

Nomenclatura Essenziale

Come accennato in Sezione 2.1, la maggior fonte di confusione nell'approccio ad Hadoop a partire dalla teoria riguarda l'uso del termine Reducer. Nella teoria infatti con questo termine viene vista nella maggior parte dei casi una singola macchina o una singola invocazione della funzione reduce. In Hadoop invece con il termine Reducer si intende un oggetto che esegue multipli reduce, ed elabora un insieme di gruppi con la stessa chiave detto partizione.

In questa appendice è proposta una sinossi dei principali termini Hadoop e del loro significato.

A.1 Nomenclatura Hadoop

A.1.1 Principale

- **Mapper**: oggetto comprensivo di stato interno, implementa la funzione Map tramite un metodo. Processa tutti i record contenuti in un input split, uno alla volta.

- **Reducer**: oggetto comprensivo di stato interno, implementa la funzione Reduce tramite un metodo. Processa tutti i record contenuti in una partizione, un gruppo alla volta. L'accesso al gruppo avviene tramite un iteratore.

- **split**: insieme di dati che viene fornito ad un Mapper. Corrisponde solitamente a un blocco di HDFS localmente presente nella macchina che esegue il Map Task associato al Mapper.

- **partizione**: insieme di gruppi che vengono forniti in input ad uno stesso Reducer.

- **gruppo**: lista ordinata di coppie che viene utilizzata come input ad una singola chiamata al metodo reduce.

A.1.2 Schedulazione

- **job**: granularità massima riconosciuta da Hadoop in modo nativo, corrisponde ad un round. Molta della configurazione di Hadoop è personalizzabile per ciascun job, come ad esempio il numero di Map/Reduce Task, le classi (da usare come Mapper/Reducer/Partitioner etc.), input/output directory, tolleranza agli errori. Consente di condividere in modo semplice eventuali parametri personalizzati (es. la regex di un grep distribuito o l'arietà dell'albero in prefixsums).

- **task**: unità logica (resiliente) di Hadoop: di norma un job fallisce se un suo task fallisce. Il jobtracker assegna i task attempt dai task non ancora eseguiti. Un task ha successo quando un suo task attempt ha successo. Il numero di task dipende da vari fattori, diversi per map task e reduce task, illustrati in Sottosezione 3.4.4.

- **task attempt**: singola esecuzione di task; per un task possono esistere anche più attempt eseguiti in contemporanea (esecuzione speculativa); per ogni task, un solo task attempt termina con successo: in caso di esecuzioni parallele, alla terminazione del primo i rimanenti vengono terminati dai tasktracker su istruzione del jobtracker. Ogni attempt utilizza un singolo oggetto di tipo Mapper o Reducer. Gli attempt svolgono, oltre alle chiamate ai metodi di Mapper e Reducer, anche le parti relative allo shuffle, equamente divise tra map attempt e reduce attempt.

- **spill**: file scritto alla fine dei map attempt, o in ingresso dei reduce attempt. Contiene dati i dati intermedi tra Mapper e Reducer.

A.1.3 Shuffle esteso

- **Partitioner**: classe che definisce le partizioni per ciascuna coppia a partire da chiave e valore. Il numero di partizioni è liberamente definibile, ma dovrebbe essere mantenuto minore o uguale al numero di Reduce Task.

- **GroupComparator**: classe che implementa un comparatore per le chiavi. Serve a definire i gruppi. Può definire un ordine diverso rispetto a SortComparator.

- **SortComparator**: classe che implementa un comparatore per le chiavi. Serve a definire l'ordine all'interno di un gruppo. Può definire un ordine diverso rispetto a GroupComparator.

- **Combiner**: oggetto comprensivo di stato interno che implementa il metodo combine. Anche quando definito, viene utilizzato opzionalmente da un task attempt alla fine di un map task per ridurre la comunicazione necessaria nella fase di shuffle. Combine deve applicare una versione opportunamente modificata di reduce che riceve in input gli elementi locali appartenenti a un gruppo. Può essere utilizzato solo per particolari funzioni reduce associative. Non esiste garanzia riguardo al numero di chiamate al metodo combine.

A.1.4 Infrastruttura

- **jobtracker**: demone principale di Hadoop per la parte MapReduce. Svolge le scelte di schedulazione dei task, assegnando i task attempt In caso di job concorrenti, effettua scelte di priorità tra i vari job. Tiene infine traccia dell'andamento globale di ciascun job e dello storico dei job eseguiti.

- **tasktracker**: demone worker per la parte MapReduce. Definisce un certo numero di task slot; si occupa di rendere disponibile i dati agli attempt, e di lanciare le JVM figlie dove questi vengono eseguiti, monitorandone l'esecuzione.

- **task slot** o **slot**: consiste in una unità di esecuzione che può essere assegnata ad un task attempt; sono specifici per tipo di attempt: di parla quindi di map slot e reduce slot. Ad ogni slot corrisponde una JVM indipendente, che

comunica con il tasktracker. Il numero di slot rappresenta uno dei vincoli alla parallelizzazione del sistema – vedi Sezione 3.5 a pag. 55.

- **namenode**: demone principale di Hadoop per la parte HDFS. Tiene in memoria la tabella di allocazione e gestisce la replicazione dei blocchi tra i vari datanode.

- **datanode**: demone worker per la parte HDFS, si occupa di memorizzare e fornire i dati. Solitamente è eseguito su una macchina insieme al tasktracker per sfruttare la località nei map task attempt.

- **driver**: programma utente che si occupa di definire, configurare e lanciare i vari job. Viene eseguito da su una macchina esterna al cluster. L'accesso all'input ed all'output dei vari job deve essere eseguito tramite la libreria HDFS.

Appendice B

Note su Hadoop

B.1 Hadoop e Slurm

Il cluster su cui è stata svolta la parte sperimentale è utilizzato anche per altri progetti di ricerca che non riguardano MapReduce e Hadoop. Per gestire l'allocazione delle risorse tra i vari utenti, nel cluster è installato SLURM [1]. La convivenza tra Hadoop e SLURM ha posto non pochi problemi.

L'obbiettivo che si è posto, per far interagire SLURM con Hadoop era quello di far eseguire i demoni Hadoop solo quando necessari. La soluzione implementata fa in modo che uno script SLURM esegua i demoni prendendo un lock su tutti i nodi del cluster. Nella filosofia di SLURM infatti agli utenti è consentito l'accesso solo ad un nodo, dal quale poter inviare i propri job utilizzando gli appositi comandi. Seguendo questa filosofia, gli script forniti con Hadoop per avviare i demoni non possono essere eseguiti in modo naturale, in quanto necessitano dell'accesso SSH su tutti i nodi. Il problema è stato risolto eseguendo lo script proposto nell'estratto di codice tramite SLURM.

Estratto di Codice 1 – Script da eseguire tramite slurm sul nodo master per avviare Hadoop

```
1 #!/bin/bash
2 # 2012/01/28 Paolo Rodeghiero
3 #
4 # Lancia Hadoop sul cluster per un tempo limitato
5 #
```

[1] SLURM: A Highly Scalable Resource Manager - https://computing.llnl.gov/linux/slurm/

```
6 # Uso:
7 #   sbatch -N<numero_nodi> start-mapred-slurm.sh <secondi_durata>
8 # (per eseguirlo su tutti i nodi: sbatch -N16 ...)
9
10 if [[ -z "$1" ]]; then
11   echo "specificare la durata in secondi di esecuzione del demone"
12   exit 1
13 fi
14
15 HADOOP_HOME=/home/rodeghiero/hadoop
16 SCRIPTS=~/scripts
17
18 # Esegue localmente al nodo principale il coodinatore
19 $HADOOP_HOME/bin/hadoop-daemon.sh start jobtracker
20 $HADOOP_HOME/bin/hadoop-daemon.sh start namenode
21
22 # Esegue lo script worker.sh su tutti i nodi
23 srun -l $SCRIPTS/slurm/hadoop/worker.sh $1
```

Estratto di Codice 2 – Script eseguito da Slurm su tutti i nodi, avvia i demoni necessari sui worker node

```
1 #!/bin/bash
2 # 2012/01/28 Paolo Rodeghiero
3 # Script to be launched by slurm to execute worker deamons for
4 # hadoop. Ut stay "alive" due to the monitoring script
5 # (host_monitor.sh) or the sleep command. When this script
6 # returns, slurm will kill the deamons.
7 #   usage: worker.sh <seconds_to_run>
8
9 # CONFIG #
10
11 HADOOP_HOME=/home/rodeghiero/hadoop
12 SCRIPTS=/home/rodeghiero/scripts
13 HOST=`hostname`
14
15 echo $HOST
16 # BEGIN #
17
18 if [[ -z "$1" ]]; then
19   echo "specificare la durata in secondi"
20   exit 1
```

```
21 fi
22
23 # Esegue i worker daemon  i nodi diversi da eridano10
24 if [ "$HOST" != "eridano10" ]; then
25         echo "esecuzione demoni worker"
26         $HADOOP_HOME/bin/hadoop-daemon.sh start tasktracker
27         $HADOOP_HOME/bin/hadoop-daemon.sh start datanode
28         $SCRIPTS/slurm/hadoop/host_monitor.sh $1
29 else
30         sleep $1
31 fi
```

Si noti che worker.sh viene eseguito su tutti i nodi, compreso il nodo master (come è considerato quello da cui vengono eseguiti i job slurm - eridano10 nel nostro caso). Poiché si è preferito non far eseguire tasktracker e datanote sul nodo master, esso è filtrato tramite una condizione sul nome host. È stato necessario porre una chiamata sospensiva poiché in caso contrario SLURM considerava lo script terminato e di conseguenza terminava tutti i processi figli.

Un primo problema di questa soluzione è la necessità di stimare il tempo di lock necessario. SLURM infatti non può eseguire il codice del driver, in quando tutti i nodi sono occupati dal job che esegue i demoni Hadoop. Infine utilizzare Hadoop insieme a SLURM in questo modo produce l'importante effetto di inibire il meccanismo di resilienza fonito da Hadoop. In caso di guasto di un nodo durante una esecuzione, SLURM interviene terminando tutti i demoni su tutti i nodi.

B.2 Requisiti per la risoluzione dei nomi

Durante l'installazione di Hadoop sul cluster una particolare difficoltà è stata riscontrata nel configurare la risoluzione dei nomi. Nella documentazione di Hadoop si raccomanda, per garantire scalabilità rispetto al numero di nodi, di eseguire la risoluzione dei nomi delle macchine del cluster tramite un DNS, garantendo la risoluzione competa ed inversa del nome completo di dominio (FQDN). Nello stesso modo si consiglia di utilizzare dhcp per la configurazione degli indirizzi.

La configurazione presente nel cluster però differisce da quella consigliata: tutti i nodi solo collegati tramite indirizzi IP statici, in due sottoreti: 192.168.2.X, che corrisponde ad una rete 10 Gbps ed 192.168.1.X che corrisponde ad una rete

1 Gbps. La risoluzione dei nomi è affidata al file /etc/hosts configurato su ogni nodo.

La documentazione non risulta molto dettagliata in riguardo a questa situazione. I requisiti che sono stati ricavati per un corretto funzionamento di Hadoop sono:

- il nodo dove gira namenode/jobtracker deve avere il nome configurato in `mapred.job.tracker` e `fs.default.name` risolto sia diretto che inverso sull'ip da cui deve ascoltare le richieste. In particolare, se risolve su un indirizzo di loopback, non sarà possibile per nessun nodo aggiungersi al cluster;

- il nodo dove gira namenode/jobtracker deve poter fare la risoluzione inversa di tutti i nomi host dei worker. Se ciò non avviene, tutti i reduce attempt falliscono;

- tutti i nodi devono poter risolvere `mapred.job.tracker` e `fs.default.name` almeno in modo diretto. Se ciò non avviene, questi nodi non potranno aggiungersi al cluster Hadoop;

Appendice C

Codice Sviluppato

Questa appendice ha lo scopo di documentare brevemente il codice sviluppato nel corso della Tesi. Come illustrato nell'introduzione, esso è disponibile in formato digitale insieme alla Tesi e pubblicamente presso http://www.linux.it/~rod/thesis, rilasciato sotto licenza Affero GNU Public License, versione 3. [1]

Nel corso dello sviluppo sono emersi due problemi riguardanti il codice: uno relativo alla difficoltà di debug dei driver per MR-algoritmi, l'altro relativo alla gestione dei dati sperimentali.

C.1 Problematiche di debug

Come ampliamente descritto in Sezione 1.3, uno degli scopi di MapReduce è quello di semplificare lo sviluppo di applicazioni parallele. Se il design dell'applicazione può effettivamente prescindere dall'implementazione di fasi complesse quali quelle di distribuzione dei dati, rimangono altri problemi che rendono difficile lo sviluppo: il debug è uno di questi.

In generale, la natura distribuita dell'esecuzione rende difficile isolare i problemi nel codice. Questo, combinato con la latenza necessaria a rieseguire un job, tende a rallentare notevolmente lo sviluppo. Hadoop fornisce alcuni strumenti utili a risolvere questi problemi:

- MiniDFSCluster e MiniMRCluster, permettono di creare in memoria dei piccoli cluster virtuali su cui eseguire il codice, da utilizzare per scrivere classi di test;

[1] http://www.gnu.org/licenses/agpl.html

- LocalJobRunner, modalità di esecuzione di Hadoop che esegue MapReduce utilizzando il filesystem locale ed un singolo JobTracker. È però vincolato ad avere un solo slot per tipo (e `mapred.reduce.tasks` deve essere al più 1). Permette di agganciare undebugger.

- MRUnit, libreria di testing in-memory per scrivere semplici unit test per Hadoop.

MiniDFSCLuster e MiniMRCluster, così come LocalJobRunner, presentano il problema del recupero dei risultati. L'output di un job non viene inviato direttamente a Java, ma deve essere recuperato attraverso le primitive di accesso al file system ed interpretato. Lo stesso problema è presente nel preparare i dati di input per i test, che devono essere preparati su file, separato dal codice del test. Entrambi gli approcci permettono un analisi molto completa, utile per analizzare dettagli, ma risultano eccessivamente lenti per test ripetuti.

MRUnit permette invece di scrivere dei test in memoria molto veloci, e l'output viene verificato in modo semplice. Risulta però limitato: è possibile testare Mapper e Reducer in modo semplice separatamente, verificandone l'output direttamente, ma non il codice del driver. È possibile anche eseguire direttamente un job completo, ma vengono forniti pochi dettagli utili al debug. Nella versione fornita con Hadoop 0.20.203 non viene fornita una interfaccia con la classe Job, ma esso deve essere specificato separatamente.

Problematiche specifiche per MR-algoritmi

La combinazione degli strumenti disponibili risulta molto efficace per lavori in cui il driver non svolge un ruolo importante, ma diventa problematica nella traduzione in Hadoop di MR-algoritmi (vedi Sezione 4.2.1).

Poiché una parte fondamentale della logica applicativa è affidata al driver, risulta infatti difficile eseguire test semplici e ripetibili per controllare la correttezza durante lo sviluppo. Utilizzando infatti sia MiniDFSClster e MiniMRCluster sia MRUnit occorre riscrivere un nuovo driver, che si comporti in modo analogo a quello che si vuole verificare. Questo approccio, oltre ad essere filosoficamente errato, è estremamente prono ad errori. Si è quindi sentita la necessità di sviluppare un framework che potesse astrarre ad un livello superiore il driver, considerando

l'intero algoritmo come una entità unica e modellando anche le interazioni con HDFS. Questo framework prende il nome di Abstraction.

Un framework per il debug: Abstraction

Abstraction ha lo scopo di creare un livello di astrazione intorno al codice Hadoop, permettendo di verificare tramite unit testing il codice dei driver. L'idea è quella di poter sostituire in modo trasparente nel codice del driver l'oggetto Job con un oggetto dummy, che esegua la computazione su MRUnit ed evidenzi errori di configurazione. Si è cercato di ottenere questo scopo mantenendo la compatibilità con l'interfaccia pubblica di Hadoop senza modificarne il sorgente, sperando quindi di rimanere compatibili con le prossime versioni.

Per prima cosa sono state create delle nuove interfacce per Job e FileSystem, [2] mentre le implementazioni originali sono state rese compatibili incapsulandole con delle nuove classi[3] che implementano queste interfacce, aggirando così le dichiarazioni 'final' presenti nel codice originale. Infine sono state sviluppate delle semplici implementazioni Java di entrambi i concetti [4].

È stato poi introdotto il concetto di ExecutionFramework [5]: esso riunisce concettualmente una implementazione di Job ed una implementazione di FileSystem, svolgendo per essi il ruolo di factory. Esistono quindi due implementazioni di ExecutionFramework: una che fornisce riferimenti ad Hadoop, una alla implementazione Abstraction.

Il concetto di MR-algoritmo è stato tradotto con il concetto di Tool. L'interfaccia Tool permette di definire in fase di creazione del Tool un oggetto ExecutionFramework. Tutte le volte che all'interno del Tool viene creato un Job, viene utilizzato questo oggetto come factory, permettendo quindi di eseguire in maniera trasparente il codice del Tool sia su Hadoop che su Abstraction.

L'implementazione dell'oggetto Job in Abstraction esegue direttamente il calcolo, utilizzando MRUnit al suo interno. L'implementazione fa uso di Log4j per esporre, a vari livelli, i dettagli relativi alla fase di shuffle. Il FileSystem è invece implementato utilizzando un albero in memoria, restando però compatibile con

[2] thesis.abstraction.interfaces.*
[3] thesis.abstraction.mapping.*
[4] thesis.abstraction.unit.*
[5] thesis.abstraction.interfaces.ExecutionFramework

gli oggetti Path della libreria di Hadoop. Non sono però supportati Combiner e Partioner, in quando non supportati da MRUnit.

C.2 Analisi dei dati: MapReduceAnalytics

Come brevemente accennato in Sezione 5.3, analizzare i dati raccolti durante gli esperimenti è risultato un compito molto complesso, sia dal punto di vista della quantità di dati sia a causa dell'alto requisito di flessibilità.

Varie soluzioni sono state considerate e parzialmente implementate, rivelandosi inadeguate. Non era infatti noto a priori cosa si voleva misurare: in qualche modo si è trattato di esplorare il comportamento sotto molti punti di vista e livello di granularità (algoritmo, job, attempt...).

La soluzione migliore è stata affidarsi ad un motore SQL ed un linguaggio di query. Sebbene questo approccio non sia perfettamente scalabile, è risultato appropriato allo scopo ed ha permesso di evitare il lavoro necessario per installare e configurare uno strumento complesso come Chuckwa.

Il motore selezionato è stato HyperSQL[6], un motore in-memory ad alte prestazioni scritto in Java, che può essere utilizzato in modalità stand-alone. Il database è quindi portabile su varie piattaforme, e non necessita di un server per essere interrogato. Il codice sviluppato, contenuto nel pacchetto MapReduceAnalytics.jar, serve ad inizializzare il database, e provvede un set di parser per popolarlo. Per le interrogazioni è stato invece utilizzato il tool di interrogazione di HyperSQL , chiamato sqltool, ed un set di script sql allegati al codice.

Nel progetto era inizialmente coinvolto anche un meccanismo di generazione automatica di grafici tramite interrogazioni dirette al database, utilizzando il pacchetto JFreeChart[7] ed una struttura di configurazione via XML. Questo sistema si è rivelato però inappropriato a livello di flessibilità, e, dato il numero limitato di grafici necessari per la Tesi, è stata preferita una soluzione basata su sqltool e gnuplot[8]. Anche questo codice è incluso nei sorgenti in uno stato di bozza.

[6] `http://www.hsqldb.org`
[7] `http://www.jfree.org/jfreechart/`
[8] `http://www.gnuplot.info/`

C.3 Struttura del Codice

Il codice sviluppato nel corso di questo lavoro può essere diviso in due parti:

- una prima parte, contenuta nel pacchetto MapReduce.jar, riguarda la traduzione dell'MR-algoritmo per la moltiplicazione matriciale, la generazione dei dati sperimentali e la verifica di correttezza dei risultati (compreso il framework Abstraction);
- una seconda parte, contenuta nel pacchetto MapReduceAnalytics.jar invece cerca di risolvere i problemi legati al recupero dei dati sperimentali (contatori, metriche e tempi) per correlarli tra loro ed analizzarli.

C.4 Esempi di Codice

C.4.1 Driver

Estratto di Codice 3 – Driver per l'implementazione *b* (L1_TheoreticalBlockDMM.java)

```
/**
 * @author Paolo Rodeghiero <paolo.rod@gmail.com>
 */
public class L1_TheoreticalPipeBlockDMM implements Tool {

    private ExecutionFramework framework;
    private final Configuration conf;
    private final Path firstOperator;
    private final Path secondOperator;
    private final Path outputPath;
    private long M;
    private int radm;
    private int radn;
    private final int ariety;
    private final int K;
    private final int multTotRound;
    private final int sumTotRound;

    private Logger logger = Logger.getLogger(this.getClass());
    private ToolLogger toolLogger = ToolLogger.getLogger();
    private Random rand = new Random();


    public L1_TheoreticalPipeBlockDMM(Configuration conf, Path
        firstOperator, Path secondOperator, Path outputPath, long M
        , int radm, int radn) throws IOException {
         this(new HadoopFramework(conf), conf, firstOperator,
            secondOperator, outputPath, M, radm, radn);
    }

    public L1_TheoreticalPipeBlockDMM(ExecutionFramework framework,
         Configuration conf, Path firstOperator, Path
        secondOperator, Path outputPath, long M, int radm, int radn
        ) throws IOException {
[...]
```

```
30
31          this.framework = framework;
32          this.conf = new Configuration(conf);
33          this.firstOperator = firstOperator;
34          this.secondOperator = secondOperator;
35          this.outputPath = outputPath;
36          this.M = M;
37          this.radm = radm;
38          this.radn = radn;
39          this.ariety = radm*radm;
40          this.K = (int)Math.min(M / (radn * radn), radn / radm);
41          this.multTotRound = (radn / radm) / K;
42          this.sumTotRound = (int) Math.ceil(Math.log(K) / Math.log(
               ariety));
43          this.conf.setInt("thesis.parallelsums.tot_rounds",
               sumTotRound);
44          this.conf.setInt("thesis.matrixmultiplication.
               totalMultRounds", multTotRound);
45          this.conf.setInt("thesis.matrixmultiplication.radn", radn);
46          this.conf.setInt("thesis.matrixmultiplication.radm", radm);
47          this.conf.setLong("thesis.matrixmultiplication.M", M);
48          this.conf.setInt("thesis.matrixmultiplication.ariety",
               ariety);
49      }
50
51      @Override
52      public void setExecutionFramework(ExecutionFramework framework)
            {
53          this.framework = framework;
54      }
55
56      @Override
57      public boolean run(boolean verbose) throws IOException,
          JobFailedException {
58
59          toolLogger.setStart(this, radn, radm, M);
60          //Path setup
61          StorageLayer fs = framework.getFileSystem();
62          Path blocksPath = new Path("DenseMatrixMult_blocks_" + Long
               .toHexString(rand.nextLong()));
```

```
63          Path tempPath1 = new Path("DenseMatrixMult_tmp1_" + Long.
                toHexString(rand.nextLong()));
64          Path tempPath2 = new Path("DenseMatrixMult_tmp1_" + Long.
                toHexString(rand.nextLong()));
65
66          logger.info("RUNNING n: " + (radn * radn) + " M: " + M + "
                m: " + (radm * radm) +" K: " + K );
67
68          BlocksFactory blockConvFactory = new BlocksFactory(conf,
                framework);
69          logger.info("Sending elements to blocks");
70          blockConvFactory.getItems2BlocksJob(firstOperator,
                secondOperator, tempPath2).run(verbose);
71          logger.info("Assembling blocks, generating replication");
72          blockConvFactory.getReduceAssembleAndReplicateJob(tempPath2
                , blocksPath).run(verbose);
73
74          PipeBlockJobFactory multFactory = new PipeBlockJobFactory(
                framework, conf);
75          for (int round = 0; round < multTotRound-1; round++) {
76              Job mult = multFactory.getReducerPipeJob(round,
                    blocksPath, outputPath);
77              logger.info("Multiplication, round " + round);
78              mult.run(verbose);
79              fs.delete(blocksPath, true);
80              fs.rename(outputPath, blocksPath);
81          }
82
83          logger.info("Multiplication and disassembling, and sending
                to sums");
84          Job multDisassemble = multFactory.getReducerPipeLastJob(
                multTotRound-1, blocksPath, outputPath);
85          multDisassemble.run(verbose);
86
87          SumByElementJobFactory sumFactory = new
                SumByElementJobFactory(framework,conf);
88
89          for (int round = 0; round < sumTotRound; round++) {
90              logger.info("Sum, round " + round);
91              Job sum = sumFactory.getReducerOnlySum(round,
                    outputPath, tempPath1);
```

```
92              //recorder.record(sum, verbose);
93              sum.run(verbose);
94              fs.delete(outputPath, true);
95              fs.rename(tempPath1, outputPath);
96          }
97
98          //cleanup
99          fs.delete(blocksPath,true);
100         fs.delete(tempPath2,true);
101         toolLogger.setEnd(this, radn, radm, M);
102         return true;
103     }
104
105     [...]
106 }
```

C.4.2 Job Factory

Estratto di Codice 4 – Job factory per l'implementazione e (ItemReplicateAndMultFactory.java)

```
1
2 /**
3  *
4  * @author Paolo Rodeghiero <paolo.rod@gmail.com>
5  */
6 public class ItemReplicateAndMulFactory {
7
8     private Configuration base;
9     private final ExecutionFramework framework;
10
11    public ItemReplicateAndMulFactory(Configuration base,
         ExecutionFramework framework) {
12        this.framework = framework;
13        this.base = new Configuration(base);
14    }
15
16    /**
17     * Read the input, replicate the factors for this round. Then
          multiplicate and sum with pastOutput.
18     * Output just the partial sums.
19     *
```

```
20       * key semantics:
21       * I: ignored for operators, same as output for pastOutput.
22       * O: (i,j,k) i block row, j block col, k mult group
23       * @param round
24       * @param firstOp input
25       * @param secondOp input
26       * @param pastOutput input
27       * @param output output
28       * @return
29       * @throws IOException
30       */
31      public Job getJob(int round, Path firstOp, Path secondOp, Path
            pastOutput, Path output) throws IOException {
32          Configuration conf = new Configuration(base);
33          conf.setInt("thesis.matrixmultiplication.round", round);
34          Job multiplication = framework.getNewJob(conf);
35
36          multiplication.setReducerClass(ItemsMultiplierReducer.class
                );
37          multiplication.setMapperClass(ItemsReplicatorMapper.class);
38          multiplication.setJobName("ReplAndMult_Item_" + round);
39          multiplication.setJarByClass(this.getClass());
40
41          //input
42          multiplication.setInputFormatClass(SequenceFileInputFormat.
                class);
43          multiplication.addInputPath(firstOp);
44          multiplication.addInputPath(secondOp);
45          multiplication.addInputPath(pastOutput);
46
47
48          //output
49          multiplication.setOutputFormatClass(
                SequenceFileOutputFormat.class);
50          multiplication.setOutputKeyClass(IntTriple.class);
51          multiplication.setOutputValueClass(MatrixItem.class);
52          multiplication.setOutputPath(output);
53
54          return multiplication;
55
56      }
```

C.4.3 Mapper

Estratto di Codice 5 – Mapper utilizzato nell'implementazione *b* (BlockToProductsMapper.java)

```
1 /**
2  *
3  * @author Paolo Rodeghiero <paolo.rod@gmail.com>
4  */
5 public class BlocksToProductsMapper extends DefaultMatrixMultMapper
    <IntTriple, MatrixBlock, IntTriple, MatrixBlock> {
6
7
8     @Override
9     protected void map(IntTriple key, MatrixBlock value, Context
        context) throws IOException, InterruptedException {
10
11             int i = key.getA();
12             int j = key.getB();
13             int g = key.getC();
14             int l = round * K + g;
15             int h = MathUtil.mod(i + j + l, radn / radm);
16
17             if(value.isFirstOperator()) {
18                 IntTriple aKey = new IntTriple(i, MathUtil.mod(2 *
                    radn / radm + h - i - (l + K), radn / radm), g);
19                 context.write(aKey, value);
20             } else if(value.isSecondOperator()) {
21                 IntTriple bKey = new IntTriple(MathUtil.mod(h - j -
                    (l + K), radn / radm), j, g);
22                 context.write(bKey, value);
23             } else if(value.isPartialSum()) {
24                 context.write(key, value);
25             }
26
27     }
28
29
30
31 }
```

C.4.4 Reducer

Estratto di Codice 6 – Reducer utilizzato per la moltiplicazione matriciale in b(TheoBlocksPipeReducer.java)

```
1  /**
2   *
3   * @author Paolo Rodeghiero <paolo.rod@gmail.com>
4   */
5  public class TheoBlocksPipeReducer extends DefaultDMMReducer<
       IntTriple, MatrixBlock, IntTriple, MatrixBlock> {
6
7      private Logger logger = Logger.getLogger(this.getClass());
8      private int totalMultRound;
9
10     @Override
11     protected void reduce(IntTriple key, Iterable<MatrixBlock>
           values, Context context) throws IOException,
           InterruptedException {
12         BlockRealMatrix A = null;
13         BlockRealMatrix B = null;
14         BlockRealMatrix C = null;
15
16         for (MatrixBlock matrix : values) {
17             if (matrix.isFirstOperator()) {
18                 A = matrix.getMatrix();
19             }
20             if (matrix.isSecondOperator()) {
21                 B = matrix.getMatrix();
22             }
23             if (matrix.isPartialSum()) {
24                 C = matrix.getMatrix();
25             }
26         }
27
28         MatrixBlock result = null;
29
30         if(A==null || B==null){
31             throw new IllegalArgumentException("One of the two
                   operands is null: A " +(A==null) + " B " +(B==null)
                   );
32         } else if (C == null) {
33             result = new MatrixBlock(A.multiply(B), 0);
34             context.write(key, result);
```

```
35          } else {
36              result = new MatrixBlock(C.add(A.multiply(B)), 0);
37              context.write(key, result);
38          }
39
40          // change the keys and output operands for the next round.
41          if(round != totalMultRound-1) {
42              int i = key.getA();
43              int j = key.getB();
44              int g = key.getC();
45              int l = round * K + g;
46              int h = MathUtil.mod(i + j + l,radn / radm);
47
48              IntTriple aKey = new IntTriple(i, MathUtil.mod(h - i -
                    (l + K),radn / radm), g);
49              IntTriple bKey = new IntTriple(MathUtil.mod(h - j - (l
                    + K),radn / radm), j, g);
50              context.write(aKey, new MatrixBlock(A, 1));
51              context.write(bKey, new MatrixBlock(B, 2));
52          }
53      }
54      [...]
```

Bibliografia

[1] Apache Project. Apache commons math api documentation - block real matrix. `http://commons.apache.org/math/apidocs/org/apache/commons/math/linear/BlockRealMatrix.html`. consultata il 15/02/2012.

[2] Barroso, L., and Hölzle, U. *The datacenter as a computer: an introduction to the design of warehouse-scale machines.* Synthesis lectures in computer architecture. Morgan & Claypool, 2009.

[3] Chu-Carroll, M. Databases are hammers, mapreduce is a screwdriver. `http://scienceblogs.com/goodmath/2008/01/databases_are_hammers_mapreduc.php`, 2008. Consultata il 17 Gennaio 2012.

[4] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[5] Dean, J., and Ghemawat, S. Mapreduce: a flexible data processing tool. *Commun. ACM 53*, 1 (2010), 72–77.

[6] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. Twister: a runtime for iterative mapreduce. In *HPDC* (2010), S. Hariri and K. Keahey, Eds., ACM, pp. 810–818.

[7] Ekanayake, J., Pallickara, S., and Fox, G. MapReduce for Data Intensive Scientific Analyses. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on* (Los Alamitos, CA, USA, Dec. 2008), vol. 0, IEEE Computer Society, pp. 277–284.

[8] Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., and Svitkina, Z. On the complexity of processing massive, unordered, distributed data. *CoRR abs/cs/0611108* (2006).

[9] Frey, E. Mapreduce bash script. http://blog.last.fm/2009/04/06/mapreduce-bash-script. Consultata il 17 Gennaio 2012.

[10] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. *SIGOPS Oper. Syst. Rev. 37* (Oct. 2003), 29–43.

[11] Goodrich, M., Sitchinava, N., and Zhang, Q. Sorting, searching, and simulation in the mapreduce framework. *ArXiv e-prints* (jan 2011). 1101.1902; Provided by the SAO/NASA Astrophysics Data System.

[12] Gray, J. Sort benchmarks. http://sortbenchmark.org/. consultata il 28 Gennaio 2012.

[13] Halevy, A., Norvig, P., and Pereira, F. The unreasonable effectiveness of data. *IEEE Intelligent Systems 24*, 2 (2009), 8–12.

[14] He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. Mars: a mapreduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 260–269.

[15] JaJa, J. *An introduction to parallel algorithms.* Addison-Wesley, Reading, Massachusetts <etc.>, 1992.

[16] Karloff, H., Suri, S., and Vassilvitskii, S. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2010), SODA '10, Society for Industrial and Applied Mathematics, pp. 938–948.

[17] Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. Filtering: a method for solving graph problems in mapreduce. In *SPAA'11* (2011), pp. 85–94.

[18] Lin, J., and Dyer, C. *Data-intensive text processing with MapReduce.* Morgan & Claypool, 2010.

[19] Loddengaard, A. Cloudera's support team shares some basic hardware recommendations. http://www.cloudera.com/blog/2010/03/clouderas-support-team-shares-some-basic-hardware-recommendations/, March 2010. Consultata il 29 Febbraio 2012.

[20] Mao, Y., Morris, R., and Kaashoek, M. F. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep* (2010).

[21] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., and Stonebraker, M. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), SIGMOD '09, ACM, pp. 165–178.

[22] Pietracaprina, A., Pucci, G., Riondato, M., Silvestri, F., and Upfal, E. Space-round trafeoffs for mapreduce computations.

[23] Rafique, M. M., Rose, B., Butt, A. R., and Nikolopoulos, D. S. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing* (Washington, DC, USA, 2009), IPDPS '09, IEEE Computer Society, pp. 1–12.

[24] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (may 2010), pp. 1 –10.

[25] Shvanchko, K. Hdfs scalability: The limits to growth. *USENIX ;login: 2*, 35 (2010), 6–16.

[26] Shvanchko, K. Apache hadoop: the scalability update. *USENIX ;login: 3*, 36 (2011), 7–13.

[27] Spolsky, J. Can your programming language do this? `http://www.joelonsoftware.com/items/2006/08/01.html`, August 2006. Consultata il 16 Gennaio 2012.

[28] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33* (August 1990), 103–111.

[29] White, T. *Hadoop: The Definitive Guide*. O'Reilly, 2010.

[30] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 1–14.

Ringraziamenti

Ai miei relatori, per i consigli, il supporto e la comprensione, ed in particolare a Francesco Silvestri, per avermi seguito nei meandri dei dettagli tecnici; Alla mia famiglia ed i miei amici, che mi hanno supportato in questi anni; infine un ringraziamento particolare alla meticolosità e pazienza di mio padre, in veste di correttore di bozze.