Università degli Studi di Roma "La Sapienza"
Anno Accademico 2010-2011

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi

MapReduce: Modelli e Algoritmi

Relatori
Prof.ssa Irene Finocchi
Prof.ssa Rossella Petreschi

Candidato
Giuseppe Crisafulli
Matricola: 1144040

Alla mia famiglia
e a Caterina

Indice

Capitolo 1

Introduzione

L'aumento della quantità di dati digitali disponibili ha sollevato nuove problematiche nel campo dell'informatica. Infatti sia dal punto di vista industriale che scientifico è sempre più forte l'esigenza di risolvere problemi su grandi quantità di dati. Già nel passato sono nati architetture studiate per coordinare grandi quantità di risorse, come ad esempio Blue Gene [8]. Queste tuttavia sono poco scalabili e fortemente centralizzate.
D'altra parte, l'abbassamento del costo dell'hardware e il rallentamento nella crescita di potenza dei processori ha reso disponibili a tutti computer mediamente potenti e connessi tramite internet. Questi potrebbero essere usati per la computazione di grandi moli di dati, tuttavia bisogna superare l'ostacolo legato al coordinamento efficiente di tali risorse. Su questa idea sono nati progetti come SETI@Home [53] che permette di distribuire l'analisi di segnali radio provenienti dallo spazio su un claster eterogeneo di computer anche molto distanti geograficamente. Tali progetti si sono rilevati molto efficaci per risolvere problemi su enormi quantità di dati, tuttavia nessuno di questi rappresenta una soluzione generale al problema.

In questo contesto nasce nel 2004 MapReduce [18], un sistema che permette di combinare computazione parallela, distribuita e sequenziale su dati di grandi dimensione. L'idea degli autori è di costruire un framework semplice che si potesse usare su un qualsiasi cluster di computer per risolvere un qualsiasi tipo di problema.

Il framework è ispirato alle funzioni *map* e *reduce* usate nella programmazione funzionale. Un algoritmo è composto di base da un passo di map e uno di reduce, questa coppia di funzioni compone un round. Durante un round il passo di map si occupa di raggruppare i dati, mentre il passo di reduce processa tali gruppi di dati. Il fatto di dover definire solo due funzioni semplifica notevolmente il compito del progettista di algoritmi.
Un altro aspetto che rende MapReduce un sistema molto semplice da usare è il fatto che l'implementazione è stata progettata per occuparsi di tutti gli aspetti di organizzazione del cluster, quali: backup, fallimenti, comunicazione,

distribuzione dei task e recovery.

In breve tempo MapReduce si è diffuso sia in campo scientifico che commerciale. Non a caso è stata istituita una conferenza ad esso dedicata, la International Workshop on MapReduce and its Applications, che è oggi alla terza edizione.

1.1 Motivazioni e obiettivo della tesi

Nonostante MapReduce si sia velocemente affermato, sono pochi i lavori che hanno cercato di analizzare il framework da un punto di vista teorico. L'obiettivo di questa tesi è proprio quello di affrontare questo aspetto, analizzando e confrontando i contributi non sperimentali.

Sono stati proposti alcuni modelli ideati per MapReduce [40] [27] [24], questi presentato analogie, ma discordano fortemente su alcuni punti. L'aspetto di maggiore difficoltà per la modellazione è rappresentato dalla scelta dei parametri del modello. Infatti il framework offre vari aspetti che possono essere considerati come parametri, tra i più importanti troviamo: il numero di macchine, la loro potenza, la loro memoria, la velocità di comunicazione, la complessità di map e reduce, lo spazio utilizzato e il numero di round. Studiare gli algoritmi che sono stati definiti per questi modelli permette di capire l'importanza di scegliere quali parametri considerare, quali no e quali limitazioni introdurre.

L'obiettivo della tesi non è solo quello di analizzare quanto già fatto, ma anzi partendo da questa analisi, di proporre una modellazione di MapReduce orientata al confronto delle prestazioni. Per questo introduciamo il modello $\mathcal{MR}$ e la misura del *tempo sequenziale su H macchine* (TSH). Tale misura permette il confronto non solo di algoritmi definiti per modelli differenti, ma anche un confronto asintotico con algoritmi sequenziali. Per dimostrare l'efficacia del modello vengono presentati come esempi degli algoritmi per il conteggio delle occorrenze e per le somme prefisse.

Infine si è scelto di chiudere la tesi studiando nuove soluzioni per un problema classico: l'ordinamento. Questo è certamente tra i primi problemi studiati in ogni nuovo modello ma non è era ancora stato affrontato in MapReduce. Per questo sono stati presentati e analizzati due algoritmi ispirati a soluzioni sequenziali.

1.2 Struttura della tesi

La tesi è strutturata in due parti. Nella prima verranno presentati alcuni modelli di computazione parallela, il framework MapReduce e una raccolta dei principali risultati ottenuti in MapReduce. La seconda parte è dedicata alla descrizione dei contributi innovativi di questa tesi, questi includo il modello

$\mathcal{MR}$, alcuni esempi e due algoritmi di ordinamento.
In dettaglio la tesi è composta dai capitoli:

- **Capitolo 2**, introduce il framework MapReduce, descrivendone l'implementazione, e richiama i modelli di computazione parallela classici, BSP e PRAM.
- **Capitolo 3**, descrive i modelli MUD, MapReduce Class e I/O-Memory-Bound MapReduce, le simulazioni di BSP e PRAM in questi modelli e raccoglie i principali algoritmi MapReduce.
- **Capitolo 4**, presenta il nuovo modello $\mathcal{MR}$ discutendo le motivazioni che hanno portato alla sua ideazione. Inoltre descrive due esempi di algoritmi per questo modello, uno per il problema del word count e l'altro per le somme prefisse.
- **Capitolo 5**, descrive due algoritmi $\mathcal{MR}$ di ordinamento ispirati ai classici radix sort e counting sort e le confronta con le versioni originali sequenziali.
- **Capitolo 6**, conclude il lavoro discutendo i risultati della tesi.

Capitolo 2

Computazione parallela e MapReduce

A dispetto della legge di Moore, già nel passato si era capito che l'aumento della velocità dei processori non era l'unico modo di incrementare le prestazioni di un processo. Un altra strategia per migliorarne le prestazioni è infatti quella di parallelizzare le computazioni.
Con la nascita di sistemi paralleli come IBM 704 [37], D825 [5], C.mmp [57], Synapse N+1 [46], ILLIAC IV [54], Cray-1 [52], Blue Gene [8] e molti altri, nacque l'esigenza di creare classificazioni e modelli per tali sistemi. Per il problema della classificazione fu creata la tassonomia di Flynn che, con qualche integrazione successiva, tuttora copre la maggior parte dei sistemi sequenziali e paralleli. Per quanto riguarda la modellazione, a partire dagli anni '80 sono stati introdotti modelli di vario tipo, ma non esiste un vero e proprio modello leader. Tra i più importanti troviamo PRAM [25], BSP [56], LogP [16], BMF [7] e CLUMBS [10].

Dopo un periodo di disinteresse, negli ultimi anni l'aumento della quantità di dati disponibili ha fatto tornare in auge la computazione parallela. Un area in cui sono nati molti sistemi paralleli e distribuiti è quella del calcolo scientifico. Ad esempio progetti come Einstein@home [1], SETI@home [53], GPUGrid [31] sono nati dall'esigenza di distribuire su tante macchine la computazione di enormi quantità di dati di svariato tipo come misurazioni spaziali, risultati di esperimenti atomici, sequenze di DNA, legami molecolari, ecc.

Un altro ambito che richiede capacità di calcolo sempre maggiori è internet. Infatti, l'aumento continuo di dati disponibili sul web e la necessità di analisi di tali dati ha sollevato il problema di organizzare e utilizzare efficientemente in parallelo risorse di calcolo distribuite. Da questa esigenza sono nati sistemi quali MapReduce [18] che offrono un compromesso tra calcolo parallelo, sequenziale e distribuito.

Prima di introdurre il soggetto della nostra tesi, cioè MapReduce, richia-

miamo brevemente due dei modelli classici di computazione parallela: BSP e PRAM.

2.1 PRAM

Il modello PRAM (Parallel Random Access Machine) è stato introdotto da Fortune e Wyllie [25] nel 1978 ed è probabilmente il più diffuso tra i modelli di computazione parallela.

Il modello prevede un certo numero P di processori, ognuno dei quali è una RAM (Random Access Machine [4]) con un insieme di registi e una memoria locale. Tutti i processori lavorano in modo sincrono e condividono una memoria di dimensione N. Il modello rappresenta una sorta di architettura a memoria condivisa in cui lo scambio di messaggi tra processori avviene solo tramite tale memoria.

A ogni passo ogni processore può leggere una locazione dalla memoria condivisa o scriverne una, e può eseguire ogni operazione RAM sulla sua memoria locale.

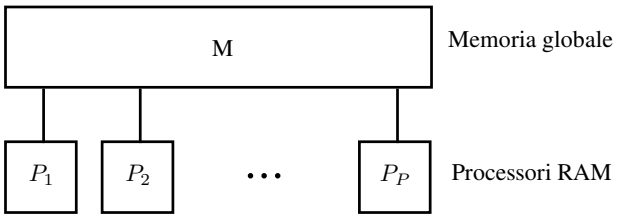

Figura 2.1: Schema modello PRAM.

A seconda delle politiche di accesso alla memoria condivisa distinguiamo tra diversi tipi di PRAM:

- **EREW** (Exclusive Read, Exclusive Write) - ogni locazione può essere letta e scritta da un solo processore per ciclo.
- **CREW** (Concurrent Read, Exclusive Write) - ogni cella di memoria può essere letta da tutti i processori ma scritta da uno solo.
- **CRCW** (Concurrent Read, concurrent Write) - a ogni passo ogni processore può leggere o scrivere ogni locazione di memoria condivisa.

La configurazione ERCW non è nell'elenco perché le operazioni di scrittura sono considerare come le più onerose e se queste fossero possibili, a parità di complessità anche le letture lo sarebbero.

Per la configurazione CRCW è necessario affrontare il problema della concorrenza delle scritture. Infatti se ad ogni passo più processori possono scrivere nella stessa locazione di memoria, allora in un unico passo una

locazione di memoria potrebbe ricevere più richieste di scrittura. Queste vengono gestite in vari modi a seconda della politica adottata, quelle più comuni sono:

- **Casuale** - arbitrariamente viene scelta una delle richieste di scrittura.
- **Priorità** - se ai processori è associata una certa priorità, la scrittura del processore con priorità maggiore è l'unica eseguita.
- **Operatore** f - sia f un operatore commutativo e associativo, e siano $v_1, v_2, \ldots, v_t$ i valori da scrivere nella locazione di memoria. Il valore scritto è quello ottenuto da $f(v_1, v_2, \ldots, v_t)$.

Il fatto che il modello PRAM rappresenti un estensione del modello sequenziale RAM permette di progettare un algoritmo parallelo usando le sole istruzioni PRAM più le istruzioni di lettura e scrittura sulla memoria globale. Gli algoritmi risultano quindi molto snelli e la loro analisi molto simile a quella del modello RAM.
Tuttavia gli aspetti che il modello ignora lo rendono irrealistico rispetto alle architetture parallele vere e proprie. Infatti, una PRAM non considera il costo di alcuni aspetti come la sincronizzazione (nessun sistema reale è sincrono), l'affidabilità (nella realtà i processori possono fallire), la località dei dati, la velocità di comunicazione e il passaggio dei messaggi (eseguito solo via memoria condivisa).

D'altra parte, ignorare questi aspetti ha reso PRAM un modello perfetto per la ricerca e in particolare per la definizione teorica di soluzioni parallele ai problemi. Infatti nei decenni sono stati definiti molti algoritmi PRAM per la risoluzione dei più classici problemi dell'informatica.

Per quanto riguarda la complessità esistono due misure fondamentali per gli algoritmi PRAM, il tempo di esecuzione e il costo. Il primo si calcola come differenza tra il tempo in cui tutti i processori hanno terminato il loro lavoro e il tempo di inizio del primo processo, mentre il costo è il prodotto tra il numero di processori e il tempo di esecuzione.
Rispetto alla architetture reali, queste misure sono appropriate rispetto alle prestazioni reali solo se l'ampiezza di banda è sufficientemente grande e se la computazione è distribuita in modo sufficientemente uniforme da colmare la latenza della comunicazione.

Per ridurre l'astrazione del modello sono stati introdotti negli anni vari modelli che si basano sulla definizione di PRAM ma che introducono alcuni degli aspetti pratici delle architetture parallele, tra questi i più importanti sono: APRAM[15] (Asynchronous PRAM), BPRAM [2] (Block transfer memory PRAM), LPRAM [3] (unlimited Local memory PRAM), WPRAM [45] (Weakly coherent PRAM), YPRAM [20] e HPRAM [35] (Hierarchical PRAM).

2.2 BSP

Nel 1989 Valiant introdusse il modello BSP (Bulk Synchronous Parallel) [56], che rappresenta per certi versi un via di mezzo tra un modello architetturale e un modello teorico. Esso si basa su tre componenti:

- P coppie processore/memoria.
- Una rete di comunicazione o router, che si occupa dela comunicazione di messaggi punto a punto.
- Una barriera di sincronizzazione per i processori.

La figura 2.2 mostra lo schema dei componenti, mentre in figura 2.3 vediamo lo schema di computazione di un algorimo BSP.

Una computazione in BSP è composta da superstep, ognuno dei quali è formato da tre passi:

- **Computazione locale**: I processori eseguono operazioni usando solo la loro memoria locale. Le computazioni locali sono da considerarsi indipendenti, nel senso che queste avvengono in modo asincrono.
- **Comunicazione**: Durante questa fase i processori inviano messaggi tra loro, questi non saranno ricevuti prima del prossimo superstep.
- **Barriera di comunicazione**: Un processore raggiunge questa fase quando ha terminato la sua fase di comunicazione. Quando tutti i processori raggiungono questa fase i messaggi inviati nella fase di comunicazione vengono resi disponibili nella memoria dei processori destinatari.

Sia il **time step** il tempo necessario ad effettuare un'operazione locale. I parametri di un algoritmo BSP sono tre:

- **p**: Il numero di coppie processore/memoria usate.

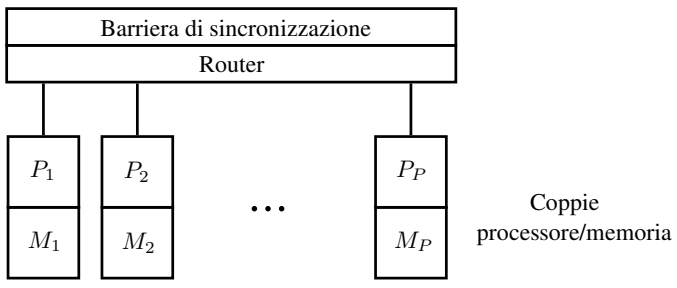

Figura 2.2: Schema modello BSP.

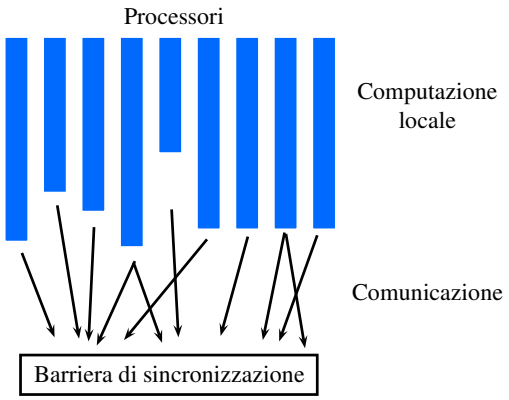

Figura 2.3: Computazione di algoritmo BSP.

- **l**: Il minimo numero di time step tra due sincronizzazioni successive. Il parametro l indica la periodicità delle sincronizzazioni, questa dipende dall'hardware ma può anche essere controllata direttamente dal software introducendo degli opportuni time step vuoti.
- **g**: Il throughput del router, inteso come la capacità di quantità di informazioni consegnate. Se una h-relazione è un modello di comunicazione in cui ogni processore invia e riceve al più h messaggi ad ogni superstep, allora hg è il tempo impiegato dal router a inviare una h-relazione.

Il parametro g è legato alla bisezione della larghezza di banda della rete di comunicazione, ma non è esattamente la stessa cosa perché esso dipende anche da altri fattori come il protocollo di rete, la strategia di routing, la gestione del buffer di processori, ecc.

Per quanto riguarda la valutazione delle prestazioni di un algoritmo BSP, notiamo che questa dipende da come l'algoritmo è stato progettato (tramite p), ma anche dalle caratteristiche hardware e software del sistema BSP usato (da cui dipendono l e g). Il costo di un superstep che implementa una h-relazione è:

$$h \cdot g + l + m$$

Dove con m indichiamo il tempo di computazione locale massimo. Esistono anche altri tipi di misure del costo di un algoritmo, tuttavia anche per queste i parametri chiave per l'efficienza sono h e g. Questi sono gli unici dipendenti dall'algoritmo e ridurre tali parametri permette, a parità di hardware, di abbassare il costo dell'algoritmo.

Dal punto di vista teorico, il modello BSP è chiaramente più complesso del modello PRAM, e questa complessità lo rende forse meno adatto alla progettazione di soluzioni parallele generali e a alto livello. Tuttavia tener conto dei parametri l e g che sono legati all'architettura hardware, rende BSP un modello più vicino alle realtà dei sistemi paralleli. Anche la scelta di usare

una barriera di sincronizzazione, piuttosto che assumere la disponibilità di processori sincroni, rende il modello BSP più facilmente realizzabile su di un qualsiasi sistema parallelo reale.

2.3 MapReduce Framework

Il framework MapReduce è stato introdotto da Dean e Ghemawat [18] come un modello per la computazione distribuita di dati di grandi dimensioni. Il modello permette di disegnare in modo semplice e veloce algoritmi che vengono eseguiti su cluster le cui dimensioni possono variare dalle poche decine alle migliaia di macchine.
MapReduce viene utilizzato ogni giorno da Google per alcuni tra i suoi più importanti tasks che coinvolgono petabytes di dati [19]. Inoltre anche la sua versione open source, Hadoop, nata come progetto open source nel 2010, viene largamente utilizzata da grandi compagnie come Yahoo! (che ha contribuito alla sua implementazione), eBay, Facebook, Twitter e IBM [34].

Il framework nasce dall'esigenza degli autori e di molti altri a Google di implementare centinaia di computazioni al giorno che processano grosse quantità di dati, come documenti, logs e pagine web, per ottenere dati strutturati di vario tipo. Molte di queste computazioni sono semplici e possono essere divise su più macchine, in modo da velocizzare le procedure e distribuire l'input. Le analogie tra queste procedure ha spinto gli autori a ideare un'astrazione ispirata alle primitive *map* e *reduce* del linguaggio funzionale Lisp [32]. La funzione map viene applicata ad ogni record e genera un set di coppie intermedie chiave-valore che saranno passate alla funzione reduce. Questa verrà applicata a tutte le coppie con la stessa chiave in modo da processare adeguatamente i dati raggruppati in precedenza dalla funzione map.
Una computazione può essere vista come una sequenza $\mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_k, \rho_k$, dove μ_i e ρ_i sono rispettivamente le funzioni map e reduce all'i-esimo round. MapReduce è un modello che su un sistema distribuito alterna computazione parallela e sequenziale, in quanto ogni coppia di funzioni μ_i, ρ_i vengono eseguite su più macchine in parallelo, ma la sequenza di round è sequenziale.

La complessità di un algoritmo può essere valutata in base a quattro parametri: il numero di macchine (mapper e/o reducer) necessari, la complessità di una singola funzione di map e/o reduce, la quantità di spazio utilizzata dalle singole macchine e il numero di round.
In particolare, è molto importante limitare il numero di macchine e lo spazio utilizzato da ognuna di esse, in modo che sia quanto meno sublineare nella dimensione dell'input. Non dobbiamo infatti dimenticare che in genere i problemi processati con questo framework hanno input di dimensione molto grande, è quindi impensabile utilizzare ad esempio $O(n)$ processori se l'input è fatto da Gigabytes o Petabytes di dati.

2.3.1 Implementazione

MapReduce è stato pensato per permettere a qualsiasi utente di utilizzare un sistema distribuito di larga scala, senza preoccuparsi di tutti i problemi pratici che comporta gestire un simile sistema. Infatti, il framework si occupa automaticamente dei dettagli riguardanti il partizionamento dei dati, lo scheduling dei thread sulle macchine, il controllo dei fallimenti dei nodi e la gestione della comunicazione tra le macchine e tra i nodi.
Illustriamo in questa paragrafo alcuni dettagli dell'implementazione presentata in [18].

Modello di programmazione

Ogni algoritmo viene eseguito in più round, ogni round consiste nell'esecuzione di un passo di map e uno di reduce, separati da un passo automatico di shuffle. Il passo di map, eseguito dai mapper, prende in input delle coppie $\langle$chiave k,valore $v\rangle$, una alla volta, e restituisce una o più coppie intermedie $\langle$chiave k_{new},valore $v_{new}\rangle$. Il passo di shuffle si occupa di mandare tutte le coppie intermedie che hanno la stessa chiave a un unico reducer. Dopo che tutte le map avranno esaurito il loro input, i reducer riceveranno le coppie intermedie corrispondenti e potranno cominciare la loro computazione. Al termine della fase di Reduce, il loro output rappresenterà il risultato finale dell'algoritmo o l'input per il prossimo round.

Il design di un algoritmo prevede la definizione dei passi di map e reduce, vediamo di seguito un esempio di pseudocodice:

```
map(String key, String value):
    // key:  nome file di log
    // value:  contenuto del file di log
    for each URL w in value:
       EmitIntermediate(w, 1);

reduce(String key, Iterator values):
    // key:  una URL
    // values:  una lista di uni
    for each v in values:
       result += Int(v);
    Emit(key,result)
```

La funzione map processa file di log di pagine web e ritorna in output delle coppie $\langle URL, 1\rangle$. La reduce prenderà tutte le coppie con uguale URL e sommerà gli uni per conoscere il conteggio degli accessi alla URL.

In generale, i tipi in input-output delle funzioni map e reduce sono due: le coppie e le liste. La funzione map prende delle coppie $\langle k_m, v_i\rangle$ e ritorna una

lista di coppie $list(\langle k_r, v_r \rangle)$, mentre la reduce prende una coppia $\langle k_r, list(v_r) \rangle$ e ritorna una lista $list(v_r)$. Da notare che il dominio delle chiavi intermedie può essere differente dal dominio delle chiavi in input alla map.

Esecuzione

Come abbiamo già detto, quando un algoritmo viene eseguito su un certo cluster di macchine, il processing dei dati viene diviso su più macchine ed eseguito in parallelo. I dati in input alla map vengono partizionati automaticamente in M parti e distribuiti a altrettanti mapper che li processano in parallelo. Lo spazio delle chiavi intermedie viene partizionato in R insiemi, dove R è il numero di reducer che eseguiranno questi task.
In figura 2.4, vengono illustrati i meccanismi che si avviano quando viene invocato un algoritmo MapReduce; Elenchiamo il dettaglio delle azioni principali:

- Sulla macchina dell'utente che ha avviato l'algoritmo l'input viene diviso in M parti. La dimensione di queste è tipicamente compresa tra 16 e 64 megabyte, tuttavia l'utente può specificare tramite un parametro dedicato la dimensione delle partizioni. Una volta diviso l'input, molte copie dell'algoritmo vengono avviate sulle macchine del cluster.

- Una di queste copie ha dei compiti speciali e viene chiamata *master*, le altre si occuperanno di processare i dati e si chiamamo *worker*. Il master ha il compito di assegnare di volta in volta gli M task di map e gli R task di reduce ai worker liberi.

- I worker che eseguono un task di map leggono la loro parte di input e creano le coppie chiave-valore che passano alla funzione map definita in precedenza dall'utente. Le coppie intermedie prodotte dalla funzione vengono memorizzate in memoria principale. Periodicamente queste vengono salvate su disco in una delle R regioni create in corrispondenza dei futuri reducer. La posizione e il numero di queste coppie dev'essere comunicata al master.

- Quando il master assegna un task di reduce a un worker, questo legge le coppie intermedie dai dischi locali dei worker che hanno eseguito le map (mapper) utilizzando delle chiamate a procedure remote. Quando il reducer ha letto tutti i dati li ordina per chiave, in quanto se lo spazio delle chiavi è più grande di R, a ogni reducer possono essere associate più chiavi. Una volta organizzate le coppie intermedie per chiave, tutte le coppie con stessa chiave vengono date in input alla funzione reduce definita dall'utente e il risultato viene salvato in locale.

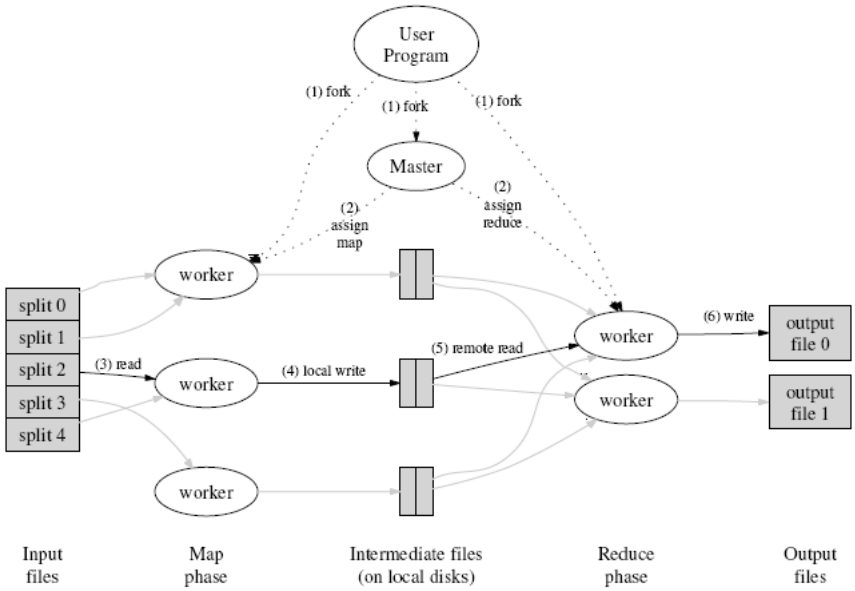

Figura 2.4: Panoramica di esecuzione di un algoritmo. Tratta da [30]

.

- Quando tutti i task di map e reduce sono stati completati il master richiama il programma utente che aveva avviato l'algoritmo. L'utente potrà leggere l'output della computazione in R file, uno per ogni reducer.

Bisogna precisare che di solito l'utente non legge gli R file di output ma li dà direttamente in input a un altro algoritmo MapReduce.

Partizionamento dei task

Abbiamo visto nello schema generale che le funzioni di map e reduce vengono divise rispettivamente in M e R task paralleli. Gli autori del framework sottolineano che il numero di worker dev'essere in genere al disotto di M e R. Ad esempio una configurazione consigliata è $M = 200000$ e $R = 5000$ con 2000 worker.
Solitamente M è molto più grande del numero di worker, il sistema automaticamente sceglierà M in modo da dividere l'input in blocchi di dimensione compresa tra 16 e 64 Mb, questo favorisce la località nelle comunicazioni (come vedremo nella prossimo paragrafo). R è invece un piccolo multiplo del numero di worker, il motivo è che per ogni reducer verrà creato un separato file di output e si vogliono evitare file piccoli.

D'altra parte le quantità M e R non possono essere eccessivamente grandi. Il fatto che il master debba tenere informazioni su ogni task crea un bound

pratico; Il master dovrà prendere $O(M+R)$ decisioni e questo richiede almeno $O(M*R)$ stati mantenuti in memoria.

Comunicazione

Il framework apparentemente richiede un ampiezza di banda molto grande per comunicare i dati in input, i dati intermedi e i dati di output all'interno del cluster. Tuttavia il framework è stato studiato in modo da ridurre il più possibile l'uso della banda. I dati vengono salvati sui dischi dei worker utilizzando il GFS [26], ovvero il filesystem distribuito creato e utilizzato da Google. GFS divide i dati in blocchi solitamente con dimensione di 64 Mb, e ne memorizza più copie (tre in genere) su più macchine. Il master al momento di decidere lo scheduling dei task, cercherà di assegnare un task alla macchina che già contiene una delle copie dei dati di quel task, oppure a una macchina che è vicina a un altra che contiene quei dati.
Con questo semplice meccanismo si cerca di sfruttare la località dei dati per far viaggiare i dati una sola volta o poco più all'interno del cluster.

Fallimenti

Il framework è stato pensato per occuparsi anche dei fallimenti delle singole macchine, esistono per questo vari meccanismi.

Il master assume grande importanza nel controllo dei fallimenti. Innanzitutto contatta periodicamente ogni worker, se per un certo tempo non riceve nulla da un worker allora lo considera fallito. Se il worker sta eseguendo o ha eseguito un task di map, questo viene considerato non eseguito e viene riassegnato a un altro worker inattivo. Lo stesso avviene se il worker stava eseguendo un task di reduce. Da notare che è necessario rieseguire ogni task di map eseguito da worker falliti. Infatti, il risultato del task di map è memorizzato localmente sul disco del worker che lo esegue, quindi dopo il suo fallimento tale risultato è da considerarsi irraggiungibile.

L'implementazione presentata in [18] non è in grado di supportare i fallimenti del master. Questo per svolgere i propri compiti deve tenere delle strutture in cui memorizza lo stato della computazione, lo stato di tutti i worker, comprese le locazioni dei dati temporanei. Quando il master fallisce queste informazioni non sono disponibile per nessun altro worker, quindi non può essere eletto un nuovo master. Tuttavia, il master salva periodicamente un checkpoint di queste strutture, questo è visualizzabile dall'utente che ha avviato l'algoritmo e può essere utilizzato per capire a che punto delle computazione si è verificato il fallimento.

Backup

La computazione di un algoritmo di MapReduce, e più in generale di un qualsiasi algoritmo parallelo, può essere rallentata dal fatto che una delle

macchine impieghi un tempo eccessivo per completare il proprio task. Molti sono i motivi che possono giustificare questo comportamento: la macchina in questione può avere un disco molto danneggiato che compie molti errori in lettura, oppure esegue il codice molto lentamente perché le sue risorse hardware, come la CPU, la memoria o la banda, sono molto limitate rispetto alle altre macchine.

Gli autori del framework hanno ideato un meccanismo dedicato a evitare queste situazioni. Quando la computazione sta per completarsi, viene creato un elenco dei task ancora al lavoro e ognuno di questi task viene assegnato a uno dei worker liberi come task di backup. Il programma si ritiene concluso quando o i task dell'elenco o i task di backup hanno terminato. Utilizzando questo meccanismo, gli autori hanno riscontrato un buona riduzione del tempo di esecuzione a fronte di un piccolo aumento dell'utilizzo delle risorse del cluster.

2.3.2 Commenti al framework

Nonostante MapReduce sia stato complessivamente ben accolto, esistono svariate critiche sia sull'idea che sulla realizzazione del framework. Le prime critiche sono state avanzate da esperti di architetture e database paralleli, quali ad esempio DeWitt e Stonebraker che in [21] e [49] confrontano MapReduce con i database relazionali paralleli. Gli autori dell'articolo vedono MapReduce come un sistema per il data warehousing e lo giudicano poco strutturato e per niente innovativo. Per questo citano sistemi già noti e largamente utilizzati che sono certamente più completi del framework di Google. Questi giudizi sono certamente corretti ma non intaccano la validità dell'approccio adottato da MapReduce, infatti il framework non è nato propriamente per il data warehousing, come obiettato anche in [38] da Jorgenson.
In seguito altri hanno manifestato opinioni concordanti al giudizio di DeWitt e Stonebraker, ad esempio criticando l'impossibilità di integrazione di MapReduce con altri sistemi noti e la mancanza di strumenti classici di DBMS che avrebbero facilitato l'analisi dei dati.

Se consideriamo MapReduce solo come un sistema di computazione parallela/distribuita, sono altre le critiche che possono essere mosse.
Innanzitutto, il fatto di non avere meccanismi di indirizzamento dei dati rende complesse alcune operazioni. Ad esempio per confrontare dei dati di cui si conosce la chiave è necessario utilizzare un intero round. Infatti bisogna utilizzare un passo di map per mandare tali dati ai reducer e un passo di reduce per effettuare il confronto vero e proprio.

Inoltre non esiste una vera e propria memoria su cui salvare informazioni temporanee, questo sfavorisce l'utilizzo di strutture dati nel corso di più round. Ad esempio un algoritmo che utilizza un albero molto grande che viene aggiornato ad ogni round, dovrà usare del tempo macchina per codificare e decodificare ad ogni round tale albero su più coppie chiave/valore.

A difesa di MapReduce c'è certamente la semplicità offerta al progettista di algoritmi. Infatti è proprio l'assenza di memoria ausiliaria, di strutture dati, di strumenti di DBMS e di meccanismi di indirizzamento che permette di definire un algoritmo semplicemente definendo due funzioni.

Capitolo 3

Modelli di computazione per MapReduce

Dal momento dell'introduzione di MapReduce sono stati introdotti modelli formali adatti alle caratteristiche del framework. Definire un modello permette di confrontare le capacità del framework con quelle di altri modelli classici e di capire quali classi di problemi sono risolvibili efficientemente. In questo capitolo vedremo alcune formalizzazioni presentate negli ultimi anni e alcuni algoritmi di simulazione di modelli classici quali PRAM e BSP.

3.1 MapReduce Class

Karloff e altri in [40] introducono un modello di algoritmo MapReduce sottolineando tre aspetti del framework: la memoria, il numero di macchine e il tempo di computazione. Gli autori si concentrano sulla necessita di limitare queste tre quantità con grandezze sublineari nella dimensione n dell'input, formalmente:

Definizione 3.1.1. *Preso un $\epsilon > 0$, un algoritmo in $\mathcal{MRC}^i$ è una sequenza $\langle \mu_1, \rho_1, \ldots, \mu_R, \rho_R \rangle$ di passi map e reduce il cui output è corretto con probabilità almeno $3/4$, dove:*

- *Ogni μ_r è una funzione map randomizzata implementata da una RAM con $O(\log n)$ words, che usa $O(n^{1-\epsilon})$ spazio e opera in tempo polinomiale in n.*
- *Ogni ρ_r è una funzione reduce randomizzata implementata da una RAM con $O(\log n)$ words, che usa $O(n^{1-\epsilon})$ spazio e opera in tempo polinomiale in n.*
- *Lo spazio totale, $\sum_{\langle k;v \rangle \in U'_r}(|k| + |v|)$, occupato dalle coppie $\langle k; v \rangle$ date in output da μ_r per ogni round r è $O(n^{2-2\epsilon})$.*

- *Il numero di round R è $O(\log^i n)$.*

Gli algoritmi della classe $\mathcal{MRC}$ sono di fatto probabilistici, gli autori indicano con $\mathcal{DMRC}$ la classe che contiene le versioni deterministiche di questi algoritmi.

Un'altra importante limitazione che gli autori impongono è che la quantità di macchine a disposizione dell'algoritmo sia $O(n^{1-\epsilon})$. Questa limitazione combinata ai primi due punti della definizione implica che la memoria a disposizione di tutto il cluster di macchine è $O(n^{2-2\epsilon})$, da cui segue il punto tre della definizione. Un ulteriore conseguenza del fatto che la memoria disponibile per ogni mapper o reducer è $O(n^{1-\epsilon})$ è il fatto che ogni coppia $\langle k, v\rangle$ deve occupare meno di $O(n^{1-\epsilon})$ spazio.

Gli autori sottolineano inoltre l'importanza del passo di shuffle in cui le coppie intermedie prodotte dalla funzione map vengono raggruppate per chiave e mandate ai reducer. Nel lemma 3.1 di [40] viene dimostrato che grazie alle limitazioni proposte nella definizione 3.1.1 il passo di shuffle può essere eseguito sulle $O(n^{1-\epsilon})$ macchine utilizzando solo la loro memoria.

Per dimostrare che un algoritmo appartiene alla classe $\mathcal{MRC}$ è sufficiente dimostrare che i vincoli di spazio e la probabilità di successo di tale algoritmo seguono la definizione 3.1.1. Gli autori in [40] presentano un algoritmo per il calcolo del minimum spanning tree (MST) di un grafo denso e ne dimostrano l'appartenenza alla classe MRC con opportune dimostrazioni.

La semplicità della definizione della classe $\mathcal{MRC}$ ha permesso agli autori di definire una tecnica generale per il disegno di algoritmi della classe. L'idea è che un algoritmo deve utilizzare funzioni con certe proprietà, l'uso di tali funzioni implica direttamente l'appartenenza dell'algoritmo alla classe. Formalmente queste funzioni devono essere espresse dalla composizione di altre funzioni che agiscono sulla partizione dell'input.

Definizione 3.1.2. *Sia S un insieme. Una funzione f definita su S si dice $\mathcal{MRC}$-**parallelizzabile** se esistono due funzioni g e h per cui:*

- *Per ogni partizione $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ di S, dove $\bigcup_i T_i = S$ e $T_i \cap T_j = \emptyset$ per $i \neq j$, f può essere calcolata come:*

$$f(S) = h(g(T_1), g(T_2), \ldots, g(T_k))$$

- *g e h possono essere espresse in $O(\log n)$ bit.*
- *g e h possono essere eseguite in tempo polinomiale in $|S|$ e ogni output di g ha dimensione $O(\log n)$.*

Grazie al lemma 6.1 di [40] è possibile introdurre un nuovo algoritmo della classe $\mathcal{MRC}$ semplicemente definendo una serie di funzioni $\mathcal{MRC}$-parallelizzabili. Di fatto il lemma 6.1 permette al progettista dell'algoritmo di concentrarsi solo sulla struttura del problema, ignorando come l'input

viene distribuito sui reducer. Utilizzando questa tecnica gli autori presentano degli algoritmi per il calcolo dei frequency moment e per il problema della s-t connettività su grafo sparso non orientato.

3.1.1 Simulazione PRAM CREW

Sempre in [40] viene presentata una simulazione di algoritmi di PRAM CREW con algoritmi della classe $\mathcal{DMRC}$. Col teorema 7.1 di [40] gli autori dimostrano che è possibile simulare ogni algoritmo per PRAM CREW che usa $O(n^{2-2\epsilon})$ memoria totale, $O(n^{2-2\epsilon})$ processori e $t = t(n)$ tempo con un algoritmo della classe $\mathcal{DMRC}$ che opera in $O(t)$ round. Nella dimostrazione viene presentato il sistema con cui le informazioni vengono comunicate tra i processori e la memoria CREW. Questo è sufficiente a dimostrare che l'algoritmo è nella classe $\mathcal{DMRC}$ perché gli autori sfruttano la definizione di funzione $\mathcal{MRC}$-parallelizzabile.

L'idea è di usare $O(n^{2-2\epsilon})$ reducer per simulare i processori e altri $O(n^{2-2\epsilon})$ reducer per simulare ogni locazione della memoria condivisa. In questo modo ogni reducer simula uno dei processori durante un passo dell'algoritmo PRAM, al termine di ogni passo il reducer invia un certo numero di richieste di letture e scritture di locazioni di memoria, che rappresentano le locazioni di memoria che il processore scrive in quel passo e i dati che vuole leggere nel prossimo passo.

La funzione map viene usata per mandare le richieste di scrittura e lettura ai giusti reducer ad ogni round. Per far questo si indica con b_i^t la coppia $\langle indirizzo, valore\rangle$ che l'i-esimo processore vuole leggere al passo t (passo dell'algoritmo PRAM), e con w_i^t la coppia $\langle indirizzo, valore\rangle$ che il processore i vuole scrivere sempre al passo t.

Poiché i reducer non possono salvare dati in memoria permanentemente per più round, ogni reducer che simula un locazione della memoria PRAM a ogni round crea una coppia $\langle indirizzo, valore\rangle$ che invia a se stesso.

Chiaramente il numero di operazioni eseguite da ogni reducer è costante. Infatti i reducer che simulano i processori ricevono un numero costante di dati e inviano un numero costante di richieste di scrittura e lettura, e i reducer che simulano la memoria ricevono al più un numero di scritture costanti (perché la PRAM è CREW) che possono essere performate in tempo costante.

3.1.2 Work Efficiency e filtering

Lattanzi e altri in [41] utilizzano il modello $\mathcal{MRC}$ come base per la definizione di algoritmi $\mathcal{MRC}^0$ e di una nuova tecnica per risolvere problemi in MapReduce chiamata *filtering*. Gli autori inoltre introducono le definizioni di *total work* e di *work efficient* per gli algoritmi MapReduce, ispirandosi alle misure di costo del modello PRAM.

Definizione 3.1.3. *Dato un algoritmo MapReduce A che opera su un input di dimensione N, sia $p_i(N)$ il numero di macchine utilizzate da A durante il round i e sia $t_i(N)$ il tempo nel caso pessimo dell'algoritmo A durante il round i. Il* ***total work*** *dell'algoritmo A è calcolato come:*

$$w(N) = \sum_{i=1}^{r} w_i(N) = \sum_{i=1}^{r} p_i(N)t_i(N)$$

Se $w(N)$ è minore o uguale al tempo del migliore algoritmo sequenziale che risolve lo stesso problema di A, allora si dice che A è ***work efficient****.*

Questa definizione permette di confrontare gli algoritmi $\mathcal{MRC}$ con gli algoritmi sequenziali. Sulla base della definizione di classe $\mathcal{MRC}$ gli autori di [41] propongono una tecnica di progettazione degli algoritmi chiamata *filtering*. L'idea principale è quella ridurre la dimensione dell'input in modo distribuito ed ottenere così un input più piccolo che può facilmente essere computato su una singola macchina. Questa tecnica combinata ai vincoli della classe $\mathcal{MRC}^0$ è stata usata dagli autori per risolvere alcuni problemi su grafi densi quali: albero di copertura minimo, matching massimale, matching pesato approssimato, copertura dei nodi e degli archi approssimata e taglio minimo.

3.2 Algoritmi I/O-Memory-Bound MapReduce

Goodrich, Sitchinava e Zhang in [27] presentano una modellazione formale degli algoritmi MapReduce che pone il focus sulla quantità di memoria utilizzata dai reducer. Mentre altri, vedi [40] e [24], limitano la memoria con delle quantità espresse in funzione dell'input, come $O(\log n)$ o $O(n^{1-\epsilon})$, gli autori di questo modello rendono la memoria un parametro arbitrario che utilizzano nel disegno e nell'analisi degli algoritmi.

Sia $n_{r,i}$ la quantità di memoria usata dal i-esimo mapper/reducer durante il round r e sia M l'upper bound di queste quantità. Un algoritmo MapReduce è definito come:

Definizione 3.2.1. *Un algoritmo* ***I/O-Memory-Bound MapReduce*** *di parametro M è un algoritmo MapReduce per cui vale* $\forall r, i \quad n_{i,r} \leq M$.

Gli autori introducono inoltre una misura delle prestazioni degli algoritmi legata alla memoria usata durante tutti i round, formalmente:

Definizione 3.2.2. *Dato un algoritmo I/O-Memory-Bound MapReduce indichiamo con C la* ***communication complexity****, ovvero la quantità totale di input e output per tutti i mapper e reducer durante tutta la computazione dell'algoritmo. Sia C_r la communication complexity del round r, C si calcola come $\sum_{r=0}^{R-1} C_r$.*

La definizione di communication complexity viene utilizzata dagli autori per esprimere il tempo di esecuzione T dell'implementazione di un algoritmo in MapReduce, questa viene calcolata come:

$$T = \Omega\Big(\sum_{r=0}^{R-1}(t_r + L + C_r/B)\Big) = \Omega(t + RL + C/B)$$

Dove t_r indica il tempo di esecuzione del singolo round r, t è la somma dei t_r, L è il tempo di latenza della comunicazione e B è l'ampiezza di banda della rete di comunicazione. Questo lower bound sul tempo viene semplificato considerando che il tempo di esecuzione del singolo reducer è dominato dalla dimensione dell'input preso, cioè da M.

$$T = \Omega(R(M + L) + C/B)$$

Gli algoritmi proposti in [27] sono disegnati per un generico M che viene utilizzato per analizzare R e C. Infatti considerando che L e B sono dipendenti dal cluster di macchine, solo la communication complexity e il numero di round appaiono nella definizione di tempo.

Vediamo di seguito due algoritmi del modello I/O-Memory-Bound MapReduce che simulano algoritmi BSP e PRAM, sempre introdotti in [27].

3.2.1 Simulazione BSP

Abbiamo visto nel paragrafo 2.2 il modello parallelo BSP introdotto da Valiant nel 1989; Goodrich, Sitchinava e Zhang dimostrano che è possibile simulare un algoritmo BSP che esegue R superstep, usa memoria totale N e $P \leq N$ processori, con un algoritmo I/O-Memory-Bound MapReduce.
L'algoritmo utilizza P reducer ognuno dei quali ha bisogno di memoria $M = \lceil \frac{N}{P} \rceil$. Il motivo di tale necessità è che la memoria della BSP (di dimensione N) è formata dalle P memorie locali dei processori, ognuno di essi viene simulato da un reducer che quindi dove avere spazio almeno $\frac{N}{P}$. Come sappiamo ogni processore della BSP esegue una certa computazione e al termine del superstep invia al più M messaggi alla barriera. La barriera di sincronizzazione viene simulata dalla funzione map che riceve tali messaggi e li invia ai corrispondenti reducer destinatari.

Inizialmente la BSP è rappresentata dall'insieme di P processori $\{p_1, p_2, \ldots, p_P\}$ e dall'insieme di celle di memoria $\{m_{1,1}, m_{1,2}, \ldots, m_{p,m}\}$, dove $m_{i,j}$ è la j-esima cella assegnata al processore i. La computazione di un superstep si svolge semplicemente usando le map per inviare le richieste ai reducer destinatari e utilizzando i reducer per eseguire le computazioni parallele dei singoli processori PRAM. Tali reducer inviano al termine di ogni round il proprio insieme di celle di memoria in modo da poterlo ricevere nuovamente al round successivo.

Gli autori di [27] dimostrano nel teorema 2 che l'algoritmo di simulazione di BSP opera in R round, con reducer che usano $M = \lceil \frac{N}{P} \rceil$ memoria e con communication complexity $O(RN)$.
L'algoritmo di simulazione di BSP è stato usato da Goodrich in [28] per risolvere problemi di computazione geometrica come il *sorting*, l'*all-nearest neighbors* in una dimensione, il problema delle *convex hull* in 2 e 3 dimensioni e per *fixed-dimensional linear programming*. Tutti questi problemi sono stati risolti usando algoritmi BSP che grazie all'algoritmo di simulazione sono stati trasformati in algoritmi I/O-Memory-Bound MapReduce.

A dispetto di quanto detto nel paragrafo precedente, per l'algoritmo di simulazione di BSP la memoria usata dai singoli reducer è limitata da una quantità definita, $M = \lceil \frac{N}{P} \rceil$. Nel prossimo paragrafo vedremo un esempio che meglio rappresenta lo spirito del modello I/O-Memory-Bound MapReduce, infatti vedremo l'algoritmo di simulazione di PRAM è progettato per un M arbitrario.

3.2.2 Simulazione PRAM CRCW

Goodrich e altri in [27] hanno presentato una simulazione del modello PRAM a lettura e scrittura concorrente su MapReduce. In particolare è stato proposta una simulazione della PRAM f-CRCW, dove scritture concorrenti a stesse locazioni in memoria vengono risolte tramite un operatore commutativo f.

L'input della simulazione è un algoritmo PRAM A che opera con un insieme di P processori $p_1, \ldots, p_P$, usando un insieme di celle di memoria $m_1, \ldots, m_N$, dove N è da dimensione della memoria condivisa e $P = N^{O(1)}$ (ad esempio $\log_M P = O(\log_M N)$).

Rispetto alla simulazione del modello PRAM CREW vista nel paragrafo 3.1.1, in questo caso c'è la difficoltà aggiuntiva dovuta alla gestione delle richieste di scrittura. Infatti ogni cella di memoria può ricevere P richieste di scrittura che dovranno essere valutate efficientemente nei reducer e ognuno di essi ha $M \leq P$ memoria a disposizione.

L'idea per superare questa difficoltà è quella di utilizzare degli *invisible funnel trees* (abbreviato IFT) per orientare le richieste di lettura e scrittura concorrenti a ogni cella di memoria. Un IFT è un albero non orientato $O(M)$-ario (branching factor $d = M/2$ e altezza $L = \lceil log_d P \rceil$) radicato in una cella di memoria che ha una foglia per ognuno dei P processori. La radice si trova al livello 0 e le foglie al livello $L - 1$. I nodi di un albero vengono etichettati con coppie, in modo che ad esempio il nodo k-esimo al livello l sia definito con la coppia $v = (l, k)$. Questo permette a ogni nodo di identificare facilmente il padre e i figli; ad esempio il generico nodo v ha padre $p(v) = (l - 1, \lfloor k/d \rfloor)$ e figli $w_q = (l + 1, k \cdot d + q)$ $\forall q \in [1, d]$.

La simulazione di un singolo passo dell'algoritmo di PRAM di fatto prevede la computazione del passo e il routing delle richieste di lettura e scrittura negli N IFTs, in dattaglio:

1. **Lettura, fase bottom-up**. Ogni processore p_i che deve leggere un locazione m_j invia una richiesta di lettura all'i-esima foglia del j-esimo IFT, ad esempio la richiesta $(j, L-1, i)$.
   Per ogni livello l da 1 a $L-2$, ogni nodo v invia la richiesta di lettura ricevuta al ruond precedente al padre $p(v)$.

2. **Lettura, fase top-down**. Le radici dei nodi che hanno ricevuto una o più richieste di lettura inviano il proprio valore ai figli che l'hanno richiesto. Allo stesso modo, per i livelli che vanno da 1 a $L-2$ ogni nodo che riceve un valore dal padre, invia tale valore ai figli che nella fase bottom-up l'avevano richiesto.

3. **Computazione interna**. All'inizio di questa fase ogni processore ha ricevuto i dati richiesti, quindi esegue la sua computazione interna e, se necessario, invia una richiesta di scrittura su una certa locazione di memoria.

4. **Scrittura, bottom-up**. Per ogni livello l da $L-2$ a 1, ogni nodo v che riceve $k < d$ richieste di scrittura dai suoi figli applica la funzione f all'input $z_1, \ldots, z_k$. Il nodo invia tale risultato al padre, se v non è radice, o modifica il valore corrente in memoria, se v è la radice.

Il teorema 3 in [27] dimostra che la simulazione usa $R = \Theta(T * \log_M N)$ round con *communication complexity* $C = O((\log_M N) \cdot T \cdot (N + P))$, per simulare un algoritmo PRAM f-CRCW che opera in T passi usando P processori e N celle di memoria.

3.3 MUD

Feldman e altri in [24] definiscono una modellazione di algoritmi per sistemi che permettono una computazione distribuita su grande quantità di dati non ordinati, in inglese *massive*, *unordered*, *distributed computation* (abbreviato MUD). La modellazione è delineata delle seguenti definizioni.

Definizione 3.3.1. *Un algoritmo **MUD** è una tripla $m = (\Phi, \oplus, \eta)$, dove:*

- $\Phi : \Sigma \to Q$ *è detta la funzione **locale** e mappa un oggetto in input in un messaggio.*
- $\oplus : Q \times Q \to Q$ *è l'operatore di **aggregazione** che mappa una coppia di messaggi in un singolo messaggio.*
- *L'operatore $\eta : Q \to \Sigma$ di **post-processing** che produce l'output finale.*

Sia T un arbitrario albero binario con n foglie, dove dato un input I, $|I| = n$, questo è disposto sulle foglie con una permutazione arbitraria. Si indica con $m_T(x)$ il messaggio $q \in Q$, risultato dell'applicazione di $\oplus$ a una sequenza $\Phi(x_1), \Phi(x_2), \ldots, \Phi(x_n)$ lungo la topologia di T.

Definizione 3.3.2. *Si dice che un algoritmo MUD m* ***computa*** *una funzione f se $f = \eta(m_T(\cdot))$ per ogni T.*

Questo modello permette di disegnare algoritmi semplicemente definendo le tre funzioni $\Phi, \oplus$ e η. Infatti, dato un problema che può essere espresso da una funzione f, è possibile risolvere tale problema con un algoritmo MUD m che *computa* la funzione f. Le complessità di tempo e di spazio di un algoritmo MUD corrispondono rispettivamente alla massima complessità di tempo e di spazio delle singole funzioni $\Phi, \oplus$ e η.

Tra i sistemi che utilizzando algoritmi MUD gli autori collocano anche MapReduce e Hadoop. Infatti fanno corrispondere le funzioni di map e reduce alle funzione locale e all'operatore di aggregazione.

In [24] viene mostrato come gli algoritmi MUD sono equivalenti agli algoritmi di streaming simmetrici. Più precisamente dimostrano che ogni funzione simmetrica che può essere calcolata da un algoritmo di streaming può anche essere calcolata da un algoritmo MUD, con simili quantità di spazio e complessità di comunicazione.

3.4 Algoritmi

Oltre hai modelli citati nei precedenti paragrafi, molti sono gli algoritmi proposti per il framework MapReduce.

Sono stati risolti alcuni problemi classici su grafi sparsi o densi. Kang e altri [39] mostrano come calcolare il diametro di un grafo di grandi dimensioni (ad esempio 1.5 miliardi di nodi e 5.5 miliardi di archi). Das e altri [17] hanno proposto algoritmi di clustering su grafi per filtrare le notizie di Google News in modo personalizzato. Goodrich e Mitzenmacher in [29] mostrano un algoritmo MapReduce per il cuckoo hashing [47] che opera in $O(\log N)$ round con communication complexity $O(N)$. Altri in [55] hanno usato MapReduce per contare il numero di triangoli di un grafo.
Oltre a questi, Chierichetti e altri in [11] presentano un algoritmo MapReduce per il problema del *Max-k-Covering*, adattando un algoritmo greedy classico al framework parallelo ottenendo soluzioni con lo stesso fattore di approssimazione. Gli autori presentando inoltre un interessante confronto tra le prestazioni della versione classica e quella della versione MapReduce utilizzando vari testset; tale confronto mostra che in generale l'algoritmo sequenziale è peggiore dell'algoritmo MapReduce per $k > 100$.
Sempre per i problemi su grafi ricordiamo gli algoritmi per il minimo albero di copertura, la connettività e Matching visti nei predenti paragrafi.

Esistono varie implementazioni dell'algoritmo PageRank in MapReduce, Lin e Schatz [43] hanno recentemente proposto una nuova implementazione che usando delle tecniche per il processing efficiente di grafi di grandi dimensioni migliora le prestazioni dei PageRank di 69% su un grafo di 1.4 miliardi di nodi.

Oltre che per la computazione di problemi su grafi, MapReduce è stato usato in molte altre aree. Grande è stato lo sviluppo per l'apprendimento automatico, sono stati proposti algoritmi per apprendimento come naive Bayes, k-means clustering, EM e SVM [14]. Altrettanto sviluppato è lo studio di soluzioni per l'elaborazione del linguaggio naturale. Ad esempio sono stati proposti algoritmi per la similarità a coppie [23], conteggio occorrenze [42], indirizzamento [44], co-clustering [48].

MapReduce è stato anche usato per operazioni su database spaziali, cioè su dati di tipo vettoriale o raster per operazioni quali la computazione di immagini aeree e la costruzione di R-Trees [13].

Numerosi sono stati i lavori che hanno analizzato l'implementazione del framework stesso o che ne hanno proposte di nuove. Bhatotia e altri [6] hanno sviluppato il framework ICOOP che estende MapReduce verso la possibilità di processare dati che cambiano durante la computazione, infatti il sistema rileva gli update e automaticamente provvede a cambiare l'output in modo efficiente riusando la computazione già eseguita sul vecchio input.
Ranger e altri in [51] analizzano MapReduce dal punto di vista dell'implementazione, confrontandola con le API di sistemi paralleli a basso livello come P-thread e propongono una nuova implementazione di MapReduce, Phoenix. Altri hanno proposto nuove implementazioni di MapReduce che includono nuove funzionalità, ad esempio citiamo Twister [22].

Di seguito presentiamo una tabella che raccoglie alcuni dei risultati più significativi dal punto di vista della progettazione e dell'analisi degli algoritmi. In tabella troviamo, oltre alle descrizione del problema risolto, anche tutti gli aspetti che riguardano la complessità dell'algoritmo MapReduce quali il numero di round, la complessità del round, il numero di macchine e la memoria a disposizione di ogni macchina.

E' interessante notare come molti algoritmi in tabella non presentano alcuni aspetti della complessità. Il motivo è che gli autori a seconda del modello adottato ritengono alcune misure più importanti di altre. Chiaramente questo fenomeno rende più difficile il confronto tra gli algoritmi. Tale considerazione rappresenta la motivazione principale di quanto viene proposto nel prossimo capitolo, dove introdurremo una modellazione del framework MapReduce orientata allo studio delle prestazioni degli algoritmi.

Problema	Ref.		Complessità		
		# Macchine	# Round	Memoria x Macchina	Complessità round
Minimo albero di copertura per grafo denso, $\|V\| = N$, $\|E\| = m \geq N^{1+c}$, $0 < c < 1$	[40]	N^c	2	$O(n^{1-\epsilon})$ $\epsilon \in (0,1)$	MST classico
k^{th} **frequency moment**	[40]	$O(n^{1-\epsilon})$	2	$O(N^{1-\epsilon})$	$O(N^{2-2\epsilon})$
s-t **connectivity** per grafo sparso non orientato, $G = (V, E)$ $\|E\| = N$	[40]	$O(n^{1-\epsilon})$	$O(\log N)$	$O(n^{1-\epsilon})$	Calcolo minimo
Simulazione di algoritmo **CREW PRAM** che usa $O(n^{2-2\epsilon})$, $O(n^{2-2\epsilon})$ processori e $t = t(n)$ tempo	[40]	$O(n^{2-2\epsilon})$	$O(t)$	costante	costante
Simulazione algoritmo **CRCW f-PRAM** che lavora in T passi usando P processori e spazio N.	[27]		$\Theta(T \log_M P)$	M	
Simulazione di algoritmo **BSP** che opera in R superstep, con memoria totale N, usando $P \leq N$ processori.	[27]	P	R	$\lceil \frac{N}{P} \rceil$	
Calcolo delle **somme prefisse** di N numeri.	[27]		$O(\log_M N)$	M	
Multi-search su albero T di dimensione N, con N query da eseguire su T	[27]		$O(\log_M N)$	M	

Problema	Ref.		Complessità		
		# Macchine	# Round	Memoria x Macchina	Complessità round
cuckoo hashing su input di dimensione N.	[29]		$O(\log N)$		
1-dimensional all nearest neighbors su una collezione di N oggetti comparabili.	[28]		$O(\log_M N)$	M	
planar convex hull di un insieme S di N punti nel piano.	[28]		$O(\log_M N)$	M	
planar convex hull di un insieme S di N punti in $\mathcal{R}^3$.	[28]		$O(\log_M N)$	M	
Minimo albero di copertura su grafo c-denso, con $G = (V, E)$, $\|V\| = n$ e $\|E\| = n^{1+c}$.	[41]	$\Theta(n^{c-\epsilon})$	$\lceil \frac{c}{\epsilon} \rceil$	$O(n^{1+\epsilon})$	$O(m\alpha(m, n))$
Matching massimale su grafo c-denso non pesato, con $G = (V, E)$, $\|V\| = n$ e $\|E\| \leq n^{1+c}$.	[41]	$\Omega(n^{1+\epsilon})$	$3 \lceil \frac{c}{\epsilon} \rceil$	$n^{1+\epsilon}$	
2-approssimazione della **copertura dei vertici** di un grafo c-denso non pesato, con $G = (V, E)$, $\|V\| = n$ e $\|E\| \leq n^{1+c}$.	[41]	$\Omega(n^{1+\epsilon})$	3	$n^{1+2c/3}$	

Problema	Ref.		Complessità		
		# Macchine	# Round	Memoria x Macchina	Complessità round
8-approssimazione del **matching massimale** di un grafo c-denso pesato, con $G = (V, E)$, $\|V\| = n$, $\|E\| = n^{1+c}$ e $W = \max w(e) : e \in E$.	[41]	$O(\log W)$	4	$n^{1+2c/3}$	$O(m)$
3/2-approssimazione della **copertura degli archi minima** su grafo c-denso, con $\|V\| = n$ e $\|E\| \leq n^{1+c}$.	[41]	$n^{1+\epsilon}$	$3\lceil \frac{c}{\epsilon} \rceil + 1$	$n^{1+\epsilon}$	$O(m)$
Taglio minimo su grafo c-denso, con $\|V\| = n$ e $\|E\| \leq n^{1+c}$.	[41]	n^{1+c}	$O(\frac{1}{\epsilon^2})$	$n^{1+\epsilon}$	

Capitolo 4

Algoritmi $\mathcal{MR}$

Proponiamo in questo capitolo le nostre considerazioni su quanto visto nei primi capitoli a proposito di MapReduce e dei modelli di computazione parallela. Verrà esposta una nuova modellazione di MapReduce con l'intenzione di sottolineare gli aspetti del framework che riteniamo di maggiore importanza.

Come esempio di utilizzo del nuovo modello presenteremo un algoritmo per il calcolo delle somme prefisse e lo confronteremo con un algoritmo MapReduce già esistente. Questo permetterà di capire quali vantaggi il modello porta alla progettazione e alla valutazione dell'algoritmo.

4.1 Algoritmi MR

Alla luce di quanto detto nei primi due capitoli a proposito di MapReduce e dei modelli di computazione parallela, proponiamo in questo capitolo un modello che raccoglie il nostro punto di vista. L'obiettivo è quello di definire un modello che permetta il confronto delle prestazioni temporali di vari algoritmi MapReduce definiti in qualsiasi tipo di modellazione. Per far questo si è pensato di porre l'accento su alcuni aspetti degli algoritmi, lasciandone altri liberi.

Gli aspetti di maggiore importanza sono certamente il numero di reducer e la memoria utilizzata dai singoli reducer, queste due quantità devono essere limitate dalla dimensione dell'input. La memoria usata dai reducer sarà, come per la MapReduce Class (vedi paragrafo 3.1), limitata da $N^{1-\epsilon}$ per un certo $\epsilon \in (0, 1)$. Per quanto riguarda il numero di reducer, si è pensato di generalizzare questo concetto distinguendo tra numero di macchine H e numero di reducer F. Infatti il numero di reducer è dipendente dalle chiavi che vengono assegnate dalla funzione map e questo dipende da come l'algoritmo è stato disegnato. Invece il numero di macchine è una quantità legata all'implementazione e all'esecuzione dell'algoritmo. In genere un algoritmo in MapReduce richiede un certo F che è spesso più grande dell'H a disposizione, vorremmo esprimere la complessità di un algoritmo anche in funzione di H

in quanto ci aspettiamo che un buon algoritmo sia scalabile, cioè che migliori le sue prestazioni al crescere di H. Proporremo formalmente una misura del tempo di esecuzione degli algoritmi in funzione di H e la useremo per capire quanto un algoritmo è scalabile.

4.1.1 Definizioni

Definiamo formalmente le prime nozioni di base. Quando parleremo di coppia $\langle k, v\rangle$, ci riferiremo a una coppia ordinata di stringhe binarie e diremo che k e v sono rispettivamente la chiave e il valore della coppia.

Definizione 4.1.1. *Una funzione* $\Pi : A \mapsto B$ *dove* $A = \{\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle\}$ *e* $B = \{\langle k_{\Pi_1}, v_{\Pi_1}\rangle, \ldots, \langle k_{\Pi_n}, v_{\Pi_n}\rangle\}$ *è detta funzione di* ***map*** *se e solo se esiste un operatore detto* ***mapper*** $\pi : \langle k, v\rangle \mapsto \{\langle k_{\pi_1}, v_{\pi_1}\rangle, \ldots, \langle k_{\pi_q}, v_{\pi_q}\rangle\}$ *per cui vale:*

$$\Pi(\{\langle k, v\rangle\}) = \pi(\langle k, v\rangle) \quad \forall \langle k, v\rangle \in A$$

La definizione di map appena data richiede implicitamente che l'output della funzione dipenda solo dalla singola coppia in input, in accordo con quanto richiesto dal framework.

Dato un insieme di coppie $A = \{\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle\}$, diremo che $K_A = \{k : \forall \langle k, v\rangle \in A\}$ è l'insieme delle chiavi di A.

Definizione 4.1.2. *Una funzione* $\Delta : A \mapsto B$*, dove* $A = \{\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle\}$ *e* $B = \{\langle k_{\Delta_1}, v_{\Delta_1}\rangle, \ldots, \langle k_{\Delta_n}, v_{\Delta_n}\rangle\}$*, è chiamata funzione* ***reduce*** *se e solo se esiste un operatore detto* ***reducer*** $\delta : A_k \mapsto B_k$*, con* $A_k = \{\langle k, v\rangle \in A\}$*, per cui vale:*

$$\Delta(A) = \bigcup_{k \in K_A} \delta(A_k)$$

Presa una generica funzione di reduce Δ definita sull'input A, questa verrà calcolata da F reducer δ_j, con F che corrisponde al numero di chiavi di A, cioè a $|K_A|$.
Le definizioni appena date permettono di definire una funzione map Π (reduce Δ) definendo semplicemente il comportamento del mapper π (rispettivamente del reducer δ) o viceversa.

Useremo il termine **macchina** per indicare un processore in grado di eseguire un mapper π (o un reducer δ) in un certo tempo T_π (rispettivamente T_δ). Infine definiamo quali sono le caratteristiche di un algoritmo in MapReduce.

Definizione 4.1.3. *Sia A un insieme di N coppie. Un* ***algoritmo*** $\mathcal{MR}$ *è una sequenza $\Pi_1, \Delta_1, \Pi_2, \Delta_2, \ldots, \Pi_R, \Delta_R$ di R funzioni map e reduce, per cui Π_1 prende in input A e Δ_R ritorna l'output dell'algoritmo. Inoltre:*

- *Indichiamo con M la massima quantità di memoria richiesta per eseguire un reducer o un mapper dell'algoritmo, deve valere che $M \leq N^{1-\epsilon}$, per un certo $\epsilon \in (0, 1)$.*

Da notare che a differenza di altri modelli, come ad esempio la $\mathcal{MRC}$ vista nel paragrafo 3.1, l'unica limitazione che imponiamo è quella sulla memoria dei mapper-reducer. Anche solo questo vincolo limita notevolmente la libertà del progettista di algoritmi. Si pensi ad esempio a un algoritmo che opera su di un grafo rappresentato con liste di adiacenza per cui a ogni lista corrisponde una coppia in MapReduce. Già questa scelta potrebbe nel caso peggiore ($\exists v \in V$ t.c. $grado(v) = N - 1$) violare il vincolo sulla memoria e rendere l'algoritmo non $\mathcal{MR}$.

Vediamo adesso come analizzare la bontà di un algoritmo $\mathcal{MR}$. La nostra analisi si concentra sulla complessità temporale degli algoritmi, in particolare su quella che chiameremo tempo sequenziale su H macchine (abbreviato TSH). Prima di calcolare il tempo dell'algoritmo dovremmo calcolare il tempo del singolo round. Questo è il tempo per eseguire tutti i mapper su H macchine più il tempo per processare gli F reducer sulle stesse H macchine.

Definizione 4.1.4. *Sia r un generico round di un algoritmo $\mathcal{MR}$ A in cui vengono eseguite le funzioni map Π e reduce Δ su input I. Il **tempo sequenziale del round** r eseguito su H macchine è:*

$$TSH_r(A) = \left\lceil \frac{|I|}{H} \right\rceil \cdot T_\pi + \left\lceil \frac{F}{H} \right\rceil \cdot T_\delta$$

Nella nostra definizione di TSH_r assumiamo implicitamente che il numero di mapper-reducer assegnato a ogni macchina sia uniforme. Tuttavia sappiamo che il cluster su cui eseguiamo l'algoritmo MapReduce può contenere macchine con prestazioni differenti, quindi saremmo portati a pensare che questa assunzione non sia vera.
Dimostreremo invece che l'uniformità sul numero di mapper-reducer vale, per farlo indicheremo con un coefficiente $p_i \in (0, 1)$ le prestazioni della generica macchina h_i del cluster. Per prestazioni della macchina intendiamo una combinazione di vari aspetti quali velocità di CPU, banda di comunicazione, latenza, quantità di memoria principale, velocità di lettura/scrittura del disco ecc... Il coefficiente p_i è tanto più piccolo quanto la macchina h_i è più lenta a eseguire i task assegnati. Prese due macchine h_i e h_j, se ad esse viene assegnato lo stesso task S che richiede T_S passi, le macchine impiegano rispettivamente tempo $p_i \cdot T_S$ e $p_j \cdot T_S$ per eseguire tale task. Se $p_i \leq p_j$ allora vale che $p_i \cdot T_S \leq p_j \cdot T_S$, cioè la macchina h_j termina prima di h_j.
Nella dimostrazione del seguente teorema useremo il termine task per indicare genericamente un mapper o un reducer e assumeremo che l'assegnazione dei task avvenga come nell'implementazione descritta nel paragrafo 2.3.1, cioè ogni nuovo task viene associato alla prima macchina inattiva.

Teorema 4.1.1. *L'assegnazione dei task (mapper/reducer) in MapReduce avviene in modo uniforme. Cioè dato un cluster di H macchine e K task uguali S, che possono essere eseguiti ognuno in T_S passi, vale per qualche*

$j \in [1, H]$:

$$p_j \cdot T_S + p_j \cdot \left\lceil \frac{K}{H} \right\rceil \cdot T_S \geq \max_{t=1}^{H} \{p_t \cdot N_t \cdot T_S\}$$

dove $h_1, h_2, \ldots, h_H$ sono le macchine, $p_1, p_2, \ldots, p_H$ sono i loro coefficienti di prestazione e N_i è il numero di task che vengono assegnati alla macchina h_i.

Dimostrazione. Se tutti i coefficienti di prestazioni hanno lo stesso valore il teorema è banalmente vero. Supponiamo inviece che i coefficienti $p_1, p_2, \ldots, p_H$ possona avere differenti valori. Sia p_i il coefficiente maggiore, cioè $\forall j \in [1, H] \quad p_j \leq p_i$. E' banale capire che il massimo tempo di esecuzione è quello della macchina h_i, cioè:

$$\max_{t=1}^{H} \{p_t \cdot N_t \cdot T_S\} \leq p_i \cdot N_i \cdot T_S + p_i \cdot T_s \tag{4.1}$$

Cioè la macchina che termina per l'ultima è la più lenta a meno di un task. Infatti se h_i non è l'ultima a finire un task, allora ci sarà un altra macchina h_r che termina per ultima e la differenza tra questo tempo e il tempo dell'ultimo task eseguito da h_i è certamente limitato dal $p_i \cdot T_S$, infatti nessuno può eseguire un task in più tempo.

Valendo la disuguaglianza 4.1, sarà sufficiente mostrare che $N_i \leq \lceil \frac{K}{H} \rceil$ per dimostrare il teorema.
Supponiamo per assurdo che $N_i > \lceil \frac{K}{H} \rceil$. In accordo con l'implementazione descritta nel paragrafo 2.3.1 i task vengono assegnati alle macchine libere in un qualche ordine, supponiamo (senza perdere di generalità) che tale ordine sia $h_1, h_2, \ldots, h_H$. Ogni macchina h_q con $q < i$, avrà un numero di task assegnati per lo meno pari a N_i poiché quando ogni macchina è libera le verrà assegnato un task prima di h_i (a causa dell'ordinamento), task che terminerà in minor tempo (perché $p_q \leq p_i$). Quindi ad ogni altra assegnazione le verrà dato un task prima che questo avvenga per h_i.
Poiché per le macchine h_d con $d > i$ vale lo stessa considerazione, possiamo affermare che $N_d \geq N_i - 1 \geq \lceil \frac{K}{H} \rceil$. Il meno uno è dovuto al fatto che nel caso peggiore all'ultima assegnazione l'ultimo task viene assegnato alla macchina h_i e tutte le macchine h_d con $d > i$ non possono pareggiare il numero di assegnazioni N_i.
Il numero totale di task assegnati dev'essere K, quindi:

$$K = \sum_{t=1}^{H} N_t = \sum_{t=1}^{i-1} N_t + N_i + \sum_{t=i+1}^{H} N_t \geq \sum_{t=1}^{i-1} N_i + N_i + \sum_{t=i+1}^{H} \left\lceil \frac{K}{H} \right\rceil >$$

$$> \sum_{t=1}^{i-1} \left\lceil \frac{K}{H} \right\rceil + \left\lceil \frac{K}{H} \right\rceil + \sum_{t=i+1}^{H} \left\lceil \frac{K}{H} \right\rceil = \sum_{t=1}^{H} \left\lceil \frac{K}{H} \right\rceil = \left\lceil \frac{K}{H} \right\rceil \cdot H \geq \frac{K}{H} \cdot H = K$$

Siamo quindi arrivati a un assurdo ($K > K$) dovuto al fatto di aver supposto che $N_i > \lceil \frac{K}{H} \rceil$. □

Per quanto dimostrato dal teorema, possiamo affermare che asintoticamente è corretto usare l'assunzione di uniformità sui task nella misura TSH_r, in particolare se questa si riferisce al tempo della peggiore macchina del cluster. Terminiamo le definizioni introducendo il tempo sequenziale di un intero algoritmo $\mathcal{MR}$.

Definizione 4.1.5. *Dato un algoritmo $\mathcal{MR}$ A il **tempo sequenziale su H macchine** è:*

$$TSH(A) = \sum_{r=1}^{R} TSH_r(A) \leq R \cdot \max_{r \in (1,R)} (TSH_r(A))$$

Di fatto la misura del tempo che abbiamo definito è espressa in funzione del numero di macchine del cluster. Questo ci permette di applicare la nostra valutazione a qualsiasi tipo di algoritmo. Infatti spesso alcuni algoritmi sono in grado di funzionare solo per un particolare numero di reducer, utilizzando la nostra misura saremo in grado di vedere quanto buoni sono questi algoritmi al variare di H. Vedremo nel prossimo paragrafo un esempio che sottolinea questo aspetto.

Dalle definizioni 4.1.3 e 4.1.5 è possibile dimostrare che l'esecuzione di una sequenza di algoritmi $\mathcal{MR}$ è ancora un algoritmo $\mathcal{MR}$, formalmente:

Teorema 4.1.2. *Una sequenza di algoritmi $\mathcal{MR}$ $A_1, A_2, \ldots, A_p$ è un algoritmo $\mathcal{MR}$ A se:*

- *L'insieme I input di A è anche input di A_1.*
- *Per ogni $j \in (1, p]$, l'insieme di output dell'algoritmo A_{j-1} corrisponde all'insieme di input dell'algoritmo A_j.*
- *L'output di A_p è anche l'output di A.*

Tale algoritmo $\mathcal{MR}$ ha tempo sequenziale su H macchine pari alla somma dei tempi sequenziali dei singoli algoritmi:

$$TSH(A) = \sum_{i=1}^{p} TSH(A_i)$$

Dimostrazione. Ogni algoritmo A_i è $\mathcal{MR}$, quindi è definito come una sequenza di funzioni map-reduce $\Pi_{1_i}, \Delta_{1_i}, \ldots, \Pi_{R_i}, \Delta_{R_i}$ in cui la massima memoria M_i usata dai mapper-reducer è limitata $N^{1-\epsilon}$. Ciò implica che:

$$A_1, A_2, \ldots, A_p = \Pi_{1_1}, \Delta_{1_1}, \ldots, \Pi_{R_1}, \Delta_{R_1}, \ldots\ldots, \Pi_{1_p}, \Delta_{1_p}, \ldots, \Pi_{R_p}, \Delta_{R_p} = A$$

è ancora un algoritmo $\mathcal{MR}$. Segue inoltre che il numero R di ruond di A è $\sum_{i=1}^{p} R_i$, con R_i il numero di round dell'algoritmo A_i.
Il tempo sequenziale su H macchine dell'algoritmo A è per definizione:

$$TSH(A) = \sum_{r=1}^{R} TSH_r(A) = \sum_{i=1}^{p} \sum_{r=1}^{R_i} TSH_r(A_i) = \sum_{i=1}^{p} TSH(A_i)$$

□

4.1.2 Esempio di algoritmo MR: Word Count

Indicheremo con *word count* il problema di contare il numero di occorrenze di un insieme di oggetti in una lista in input. Ad esempio dato in input un documento e un insieme di parole vorremo contare quante volte queste parole occorrono nel documento. Questo problema è uno dei più semplici nel campo dell'elaborazione del linguaggio e viene spesso utilizzato come esempio di algoritmo del framework MapReduce. Di seguito vediamo lo pseudocodice di una soluzione classica[1] del problema:

```
map(String key, String value):
    // key:  posizione nel documento
    // value:  parola nella posizione key
       EmitIntermediate(value, 1);

reduce(String key, Iterator values):
    // key:  una parola
    // values:  una lista di uni
    for each v in values:
       result += ParseInt(v);
    Emit(AsString(result))
```

L'algoritmo viene eseguito in un solo round e utilizza un numero di reducer pari al numero di parole distinte contenute nei documenti. Inoltre le funzioni di map e reduce rispettano le definizioni imposte dal nostro modello, tuttavia nel caso peggiore il limite sulla memoria usata dai reducer non vale. Infatti se tutte le parole sono uguali, tutto l'input viene mandato a un unico reducer, questo dovrà quindi utilizzare una quantità di memoria pari a N (con N indichiamo il numero di parole del documento in input).

Supponiamo sia K il numero di parole dell'insieme alfabeto. Considerando solo il caso medio (quando ognuna della K parole ha $O(N/K)$ occorrenze), potremmo affermare che questa procedura è un algoritmo $\mathcal{MR}$ con tempo sequenziale su H macchine pari a:

$$TSH = 1 \cdot (1 \cdot \frac{N}{H} + \frac{K}{H} \cdot \frac{N}{K}) = \frac{N}{H}$$

Questo semplice esempio mostra come la limitazione sulla memoria dei mapper-reducer, costringe il progettista di algoritmi a definire un algoritmo fortemente bilanciato. Infatti se non si tiene conto di tale aspetto è facile progettare un algoritmo che nel caso peggiore non è $\mathcal{MR}$. Nel prossimo

[1] Algoritmo tratta da [18] e modificato.

capitolo vedremo una soluzione al problema del conteggio delle occorrenze che anche nel caso peggiore rientra nel modello $\mathcal{MR}$.

Da notare che se nel nostro modello avessimo incluso anche una limitazione al numero di reducer l'algoritmo avrebbe un altro caso peggiore. Infatti se tutte le parole del documento sono distinte il numero di reducer diventa $N \geq N^{1-\epsilon}$.

4.2 Somme prefisse

Il problema delle somme prefisse è un problema classico che è stato studiato su vari modelli di computazione, anche paralleli, in quanto viene usato in molti contesti come ad esempio per l'ordinamento, come nel counting sort, o per grafi e alberi, come per il Tour di Eulero.
Dato un vettore A di N interi, il problema delle somme prefisse è quello di calcolare per ogni $A[i]$, $0 \leq i \leq N-1$, la somma $\sum_{j=0}^{i} A[j]$.

Nei prossimi paragrafi presenteremo alcune soluzioni che sono state proposte su PRAM[50] e su MapReduce[27], e introdurremo un nuovo approccio ideato in questa tesi che presenta la migliore complessità su MapReduce.

4.2.1 Somme prefisse su PRAM

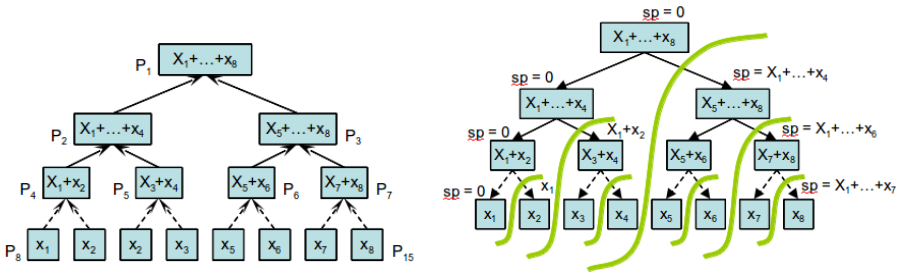

Figura 4.1: Somme prefise su PRAM. Immagine tratta da [50].

Esistono vari algoritmi che risolvono efficientemente il problema delle somme prefisse per il modello PRAM, questi si differenziano a seconda del tipo di PRAM utilizzata. Presenteremo qui la soluzione su albero binario PRAM EREW poiché è quella più simile agli algoritmi che vedremo per MapReduce.

L'albero binario utilizzato ha N foglie che contengono i valori del vettore di input, vediamo in figura 4.1 un esempio con un vettore di dimensione 8. L'algoritmo si compone di due fasi:

- **Fase top-down**. Ogni foglia invia il proprio valore al padre.
  Per ogni livello l, da $l = \log N - 1$ a $l = 0$, ogni nodo v che riceve

due valori dai rispettivi figli, somma questi valori e invia il risultato al padre.

- **Fase bottom-up**. La radice invia il valore ricevuto da sinistra al figlio di destra e termina.
  Per ogni livello l, da $l = 0$ a $l = \log N - 1$, ogni nodo v riceve un valore sp dal padre, lo somma al valore v_s ricevuto al passo precedente dal figlio sinistro e lo invia al figlio destro. Inoltre invia il valore sp al figlio sinistro.
  Le foglie riceveranno un valore dal padre che sommeranno al proprio per ottenere la somma prefissa finale.

L'algoritmo impiega un tempo parallelo logaritmico.

4.2.2 Somme prefisse con *invisible funnel tree*

L'algoritmo proposta da Goodrich e altri[27] si basa sulla soluzione per PRAM vista nel paragrafo precedente. In questo caso gli autori sostituiscono l'albero binario con un invisible funnel tree (abbreviato IFT).
Un IFT è un albero non orientato $O(M)$-ario (branching factor $d = M/2$ e altezza $L = \lceil log_d N \rceil$) radicato in una cella di memoria che ha una foglia per ognuno delle N posizioni del vettore A in input. La radice si trova al livello 0 e le foglie al livello $L - 1$. I nodi di un albero vengono etichettati in modo che ad esempio il nodo k-esimo al livello l è definito come $v = (l, k)$. Questo permette a ogni nodo di identificare facilmente il padre ($p(v) = (l - 1, \lfloor k/d \rfloor)$) e i figli (il q-esimo figlio di v sarà $w_q = (l + 1, k \cdot d + q)$).

La computazione si compone di due fasi, nella prima gli elementi del vettore vengono sommati tra loro viaggiando dalle foglie alla radice, nella seconda i risultati delle somme parziali verranno distribuiti dai padri ai figli. In tal modo alla fine della seconda fase ogni nodo avrà avuto la possibilità di completare la propria somma, foglie comprese. Il risultato calcolato dalle foglie corrisponderà alla somma parziale degli elementi. L'algoritmo 1 mostra i dettagli della procedura.

In [27] gli autori dimostrano che data un vettore di N numeri, l'algoritmo proposto può calcolare le somme prefisse in $O(\log_M N)$ round con communication complexity $O(N \log_M N)$. Gli autori non discutono però la complessità di un singolo round, tentiamo di analizzarla per confrontarla nel prossimo paragrafo con la complessità dell'algoritmo proposto in questa tesi.
In un generico passo dell'algoritmo il cluster di macchine dovrà svolgere la computazione di tutti i nodi di un livello l. L'albero è d-ario ($d = M/2 = O(M)$), quindi il numero di nodi al livello l è d^l, con $1 \leq l \leq \log_M N$, questi nodi sono eseguiti su H macchine, quindi ogni macchina esegue la computazione di $O(d^l/H)$ nodi.
Discutiamo adesso il tempo sequenziale di ogni nodo: nel caso peggiore ci troviamo in fase top-down, in cui ogni nodo somma tutti i valori ricevuti dai

Algorithm 1 Somme prefisse [27]

- Fase 1, bottom-up:
for $l = L - 1$ to 1 **do**
  for all v a livello l **do**
    if v è foglia **then**
      riceve a_i;
      invia $s_v = a_i$ al padre $p(v)$;
    else
      $w_0, w_1, \ldots, w_{d-1}$ sono i figli da sinistra a destra;
      riceve $A_V(r) = \{s_{w_0}, s_{w_1}, \ldots, s_{w_{d-1}}\}$ dai figli;
      $s_v = \sum_{j=0}^{d-1} s_{w_j}$;
      invia s_v al padre $p(v)$;
    end if
  end for
end for
- Fase 2, top-down:
for $l = 0$ to $L - 1$ **do**
  for all v a livello l **do**
    if v è radice **then**
      riceve $A_v(r) = \{s_{w_0}, s_{w_1}, \ldots, s_{w_{d-1}}\}$ dai figli nella fase 1;
      per ogni figlio w_i, crea $s'_i = \sum_{j=0}^{i-1} s_{w_j}$ e lo invia a w_i;
    else
      riceve $s_{p(v)}$ dal padre $p(v)$;
      if v è foglia **then**
        a_k è il suo elemento;
        $a_k + s_{p(v)}$ è il suo valore finale;
      else
        per ogni figlio w_i, crea $s'_i = s_{p(v)} + \sum_{j=0}^{i-1} s_{w_j}$ e lo invia a w_i;
      end if
    end if
  end for
end for

figli, questo richiede d passi tali somme. Ne consegue che ogni macchina esegue $O(M)$ operazioni per ognuno degli d^l/H nodi che gli sono stati assegnati. La complessità del singolo round nel caso peggiore è:

$$\max_{r\in(1,R)}(TSH_r) = \frac{d^l}{H}\cdot d \cong \frac{M^l}{H}\cdot M =$$

$$= \frac{M^{l+1}}{H} \leq \frac{M^{log_M N+1}}{H} = \frac{M\cdot N}{H}$$

Sappiamo dal lemma 1 in [27] che il numero di round dell'algoritmo è $O(\log_M N)$ quindi il tempo sequenziale su H macchine dell'algoritmo è:

$$TSH = \frac{M \cdot N}{H} \cdot (\log_M N) = \frac{M \cdot N}{H} \cdot \frac{\log N}{\log M} = \frac{N \log N}{H} \cdot \frac{M}{\log M}$$

4.2.3 Algoritmo $\mathcal{MR}$ per somme prefisse

L'idea di quest'algoritmo è quella di usare le capacità aggreganti del passo di Map per rendere il problema più semplice e risolverlo in pochi round.
In MapReduce non esiste la nozione di vettore di input, infatti l'input è rappresentato solo da un insieme di coppie non ordinate. Reppresenteremo il vettore V in input con N coppie $< i, V[i] >$.

Al primo round procederemo dividendo il vettore in sottovettori di dimensione $N^{1-\epsilon}$ (per un qualche $0 < \epsilon < 1$) e grazie alle Map indirizzeremo ogni sottovettore a un reducer che calcolerà le somme profisse parziali di quel sottovettore. L'output di un reducer sarà rappresentato dalle stesse coppie prese in input in cui il valore è stato sostituito con quello della somma parziale.
Al secondo round i mapper come prima indirizzeranno le coppie di uno stesso sottovettore a un unico reducer, con la differenza che alcune coppie verranno mandate a più di un reducer. Ogni coppia $\langle i, V[i] \rangle$, in cui i è l'indice più alto per un j-esimo sottovettore, verrà mandata a tutti i reducer che si occuperanno dei sottovettori con indici maggiori di j. Quindi il k-esimo reducer ($k > 0$) riceverà non solo i valori del k-esimo sottovettore ma anche tutte le somme parziali dei sottovettori con indice inferiore di k. A questo punto sommando tutti i valori ricevuti ogni reducer potrà calcolare le somme prefisse totali.

Vediamo nell'algoritmo 2 i dettagli dei due round. L'output dell'ultimo mapper sarà composto da coppie $\langle i, a_i \rangle$ dove $a_i = \sum_{j=0}^{i} V[i]$, da queste possiamo facilmente ricomporre il vettore di somme prefisse. In figura 4.2 vediamo un esempio di esecuzione dell'algoritmo su di un vettore di dimensione 16.

Analizziamo quindi la complessità dell'algoritmo proposto, considereremo anche la commnunication complexity in modo da poter confrontare le nostre prestazioni con quelle ottenute in [27].

Teorema 4.2.1. *L'algoritmo 2 è un algoritmo $\mathcal{MR}$ che risolve il problema delle somme prefisse di N interi in 2 round, con tempo sequenziale su H macchine $O(\frac{N^{1+\epsilon}}{H})$ e communication complexity $O(N + N^{2\epsilon})$.*

Dimostrazione. L'algoritmo è certamente $\mathcal{MR}$ perché è definito come una serie di mapper e reducer, ognuno dei quali utilizza una quantità di memoria al più grande $N^{1-\epsilon} + N^{\epsilon} \leq O(N^{1-c})$, per un qualche $c \in (0, 1)$.
Il numero di round è 2 per costruzione. Al primo round i mapper eseguono lavoro costante su N coppie, mentre i reducer deve sommare tutti gli $N^{1-\epsilon}$ valori ricevuti, questi dovranno prima essere ordinati rispetto alla posizione

Algorithm 2 Somme prefisse, algoritmo costante.

- Mapper 1:
Input: coppie $\langle i, V[i]\rangle$ con $0 \leq i \leq N-1$
$k := \lfloor \frac{i}{N^\epsilon} \rfloor$;
output$(\langle k, (i, V[i])\rangle)$;
- Reducer 1:
Input: vettore di b coppie $V_k = \{(i, V[i]), (j, V[j]), \ldots, (z, V[z])\}$
$S_k = \text{sort}(V_k)$;
$a := 0$;
for $p = 0$ **to** b **do**
  $\langle i, V[i]\rangle = S_k[p]$;
  $a = a + V[i]$;
  output$(\langle i, a\rangle)$;
end for
- Mapper 2:
Input: coppie $\langle i, a_i\rangle$ con $0 \leq i \leq n-1$
$k := \lfloor \frac{i}{N^\epsilon} \rfloor$;
output$(\langle k, (i, a_i)\rangle)$;
if $\lfloor \frac{i+1}{b} \rfloor > k$ **then**
  for $k = \lfloor \frac{i+1}{b} \rfloor$ **to** $\lfloor \frac{n}{b} \rfloor$ **do**
    output$(\langle k, (i, a_i)\rangle)$;
  end for
end if
- Reducer 2:
Input: vettore di $b + k$ coppie $V_k = \{(i, a_i), (j, a_j), \ldots, (z, a_z)\}$
$S_k = \text{sort}(V_k)$;
$a := 0$;
for $p = 0$ **to** k **do**
  $\langle i, a_i\rangle = S_k[p]$;
  $tot = tot + a_i$;
  for $p = k$ **to** $b + k$ **do**
    $\langle i, a_i\rangle = S_k[p]$;
    $a = tot + a_i$;
    output$(\langle i, a\rangle)$;
  end for
end for

nel vettore. Tale ordinamento può essere fatto in tempo lineare poiché gli elementi da ordinare sono interi in un intervallo grade $N^{1-\epsilon}$ quindi:

$$TSH_1 = \frac{N}{H} + \frac{N^{1-\epsilon}}{H} = \frac{N}{H}$$

Nel secondo round i mapper ricevono N coppie e alcune di queste dovranno

essere inviate ad al più N^ϵ reducer. I reducer invece ricevono al più $N^{1-\epsilon}+N^\epsilon$ coppie che bisogna solo sommare, la complessità del round è quindi:

$$TSH_2 \leq \frac{N}{H} \cdot N^\epsilon + \frac{N^\epsilon}{H}(N^\epsilon + N^{1-\epsilon}) = \frac{N^{1+\epsilon}}{H} + \frac{N^{2\epsilon}}{H} + \frac{N}{H} = O\left(\frac{N^{1+\epsilon}}{H}\right)$$

Il tempo sequenziale del secondo round è certamente il massimo, infatti:

$$\frac{N^{1+\epsilon}}{H} > \frac{N}{H} \Leftrightarrow N^\epsilon > 1 \Leftrightarrow \epsilon > 0$$

Il tempo sequenziale su H dell'intero algoritmo è quindi:

$$TSH = R \cdot \max_{r \in (1,R)} (TSH_r) = O\left(\frac{N^{1+\epsilon}}{H}\right)$$

Per quanto riguarda la quantità di dati comunicati, notiamo che al primo round verranno inviati esattamente N dati, mentre nel secondo $N + \sum_{j=1}^{N^\epsilon - 1} j = O(N + N^{2\epsilon})$. □

Confrontando questo algoritmo con quello proposto in [27] notiamo che il numero di round è inferiore se $N \geq M^2$. Per quanto riguarda il tempo sequenziale, il nostro algoritmo usa N^c per un qualche $c \in (0, 1)$, quindi a parità di memoria utilizzata ($M = N^c$):

$$\frac{N^{1+\epsilon}}{H} < \frac{N \cdot M}{H} \frac{\log}{\log M} \Leftrightarrow N^\epsilon < N^c \frac{\log N}{\log N^c} \Leftrightarrow N^\epsilon < \frac{N^c}{c} \Leftrightarrow N^\epsilon < \frac{N^c}{<} \frac{N^c}{c} \Leftrightarrow \epsilon < c$$

Ma c è sempre maggiore di c in quanto è stato scelto in modo che $N^c \geq N^\epsilon + N^{1-\epsilon}$.

Rispetto al tempo di un algoritmo sequenziale, l'algoritmo 2 ha prestazioni migliori se si hanno a disposizione un numero di macchine pari a:

$$\frac{N^{1+\epsilon}}{H} \leq N \quad \Leftrightarrow \quad N^\epsilon \leq H$$

Questo significa che, se scegliamo di dividere il vettore in sottovettori di dimensione $N^{1-\epsilon}$ per un fissato ϵ, allora abbiamo bisogno di almeno N^ϵ macchine per ottenere un tempo che sia inferiore a quello di un banale algoritmo sequenziale.

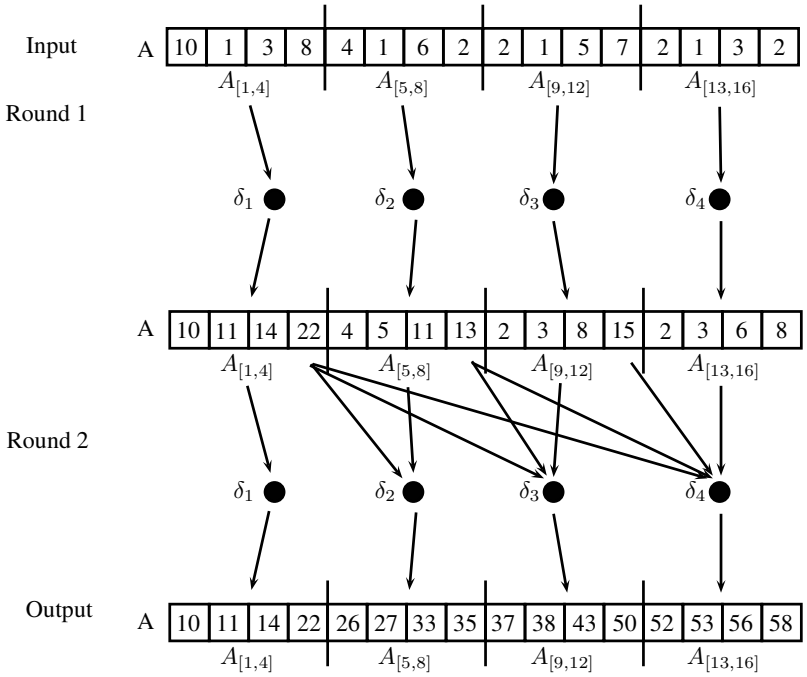

Figura 4.2: Esempio di esecuzione dell'algoritmo 2 su di un vettore A di dimensione $N = 9$, con $\epsilon = 1/2$ e $N^{\epsilon} = N^{1-\epsilon} = 3$. Le frecce indicano a quali reducer i mapper inviano i dati.

Capitolo 5

Ordinamento

L'ordinamento di dati è uno dei problemi fondamentali dell'informatica. Non a caso è stato tra i primi problemi affrontati in ogni nuovo modello proposto. Anche per il framework MapReduce sono state proposte soluzioni al problema. Lo stesso team di Google nel 2008 ha annuinciato di essere in grado di ordinare 1 TB di dati in 68 secondi usando un cluster di 1000 macchine [1]. Tale esperimento è stato eseguito seguendo le regole di un famoso benchmark e ha battuto il precedente record di 209 secondi con 910 macchine[2].

Oltre ai risultati sperimentali solo Goodrich in [28] presenta un algoritmo di cui fornisce un'analisi asintotica della complessità. Tuttavia tale algoritmo viene ottenuto usando un algoritmo BSP simulato grazie alla procedura che abbiamo visto nel paragrafo 3.2.1.

Affrontiamo in questo capitolo il problema dell'ordinamento, considerando due algoritmi sequenziali classici come il counting sort e il radix sort e presentando due versione parallele nel modello $\mathcal{MR}$ definito nel capitolo precedente.

5.0.4 Counting sort

Prima di presentare la nostra versione dell'algoritmo in MapReduce, rivediamo brevemente la versione sequenziale.

Il counting sort ordina N elementi in tempo sequenziale $O(N)$ se gli elementi del vettore sono interi nell'intervallo da 1 a k, con $k = O(N)$. Questa tecnica di ordinamento non è basata sul confronto degli elementi, l'idea è quella di determinare la posizione di un elemento x contando quanti elementi nel vettore sono più piccoli di x.

Nell'algoritmo 3 vediamo lo pseudocodice dell'algoritmo; questo prende in input un vettore A di N posizioni e restituisce in output un vettore B di N posizioni e utilizza un terzo vettore di appoggio C di k posizioni.

[1] Tratto da http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.
[2] Maggiori informazioni all'indirizzo http://sortbenchmark.org/.

Algorithm 3 Counting sort, algoritmo sequenziale.

Input: A, B, k
for $j = 1$ TO k **do**
  $C[j] = 0$;
end for
for $j = 1$ TO N **do**
  $C[A[j]] = C[A[j]] + 1$;
end for
for $i = 2$ TO k **do**
  $C[i] = C[i] + C[i-1]$;
end for
for $j = N$ TO 1 **do**
  $B[C[A[j]]] = A[j]$;
  $C[A[j]] = C[A[j]] - 1$;
end for

L'algoritmo con una prima passata di A conta il numero di elementi uguali e salva questa informazione nel vettore C. Poi esegue le somme prefisse del vettore C e quindi scorre nuovamente A in ordine inverso, inserendo gli elementi in B nelle posizioni indicate da C. Da notare che durante l'ultima scansione di A è necessario decrementare le posizioni di C in modo che stessi elementi di A vengano posizionati in posizioni decrescenti. Quest'ultimo accorgimento rende l'algoritmo stabile, cioè elementi con lo stesso valore appaiono in B nello stesso ordine in cui appaiono in A.

Banalmente la complessità dell'algoritmo è $O(N)$, con $K = O(N)$, in quanto il primo e il terzo ciclo svolgono $O(K)$ iterazioni, mentre il secondo e il quarto eseguono $O(N)$ iterazioni.

5.0.5 Algoritmo $\mathcal{MR}$ per counting sort

Presentiamo in questo paragrafo un algoritmo $\mathcal{MR}$ per l'ordinamento ispirato all'algoritmo sequenziale per il counting sort. Come visto nel paragrafo precedente, la versione sequenziale del counting sort è composta da quattro fasi che corrispondono ai quattro cicli, che sono: inizializzazione del vettore C, conteggio degli elementi, somme prefisse su C e ordinamento del vettore. La procedura che proponiamo esegue queste quattro fasi in parallelo su un numero generico H di macchine.

Come nel paragrafo precedente, indichiamo con A il vettore di input ($|A| = N$) e con B il vettore di output, mentre C ($|C| = k$) è il vettore ausiliario usato dall'algoritmo per contare il numero di occorrenze degli elementi e determinare le loro posizioni finali. Come nel caso dell'algoritmo per le somme prefisse, i vettori verranno rappresentati da coppie, ad esempio il vettore A di input è rappresentato dall'insieme di coppie $\{\langle i, A[i]\rangle : \forall i \in [1, N]\}$.

L'algoritmo utilizza un idea simile a quella usata per le somme prefisse in 4.2.3, infatti l'input viene diviso in parti uguali e assegnato ai reducer. Il loro numero è $N^{1-\epsilon}$, per un certo $\epsilon \in (0,1)$, quindi il vettore in input A viene diviso in sottovettori di dimensione $\frac{N}{N^{1-\epsilon}} = N^{\epsilon}$. Indicheremo con $A_{[x:y]}$ il vettore che contiene gli elementi di A dalla posizione x alla posizione y, estremi inclusi. L'i-esimo reducer δ_i si occuperà del sottovettore $A_{[((i-1)N^{\epsilon}):(iN^{\epsilon}-1)]}$ sia nella fase iniziale di conteggio delle occorrenze che nella fase di ordinamento.

Presentiamo di seguito le varie fasi dell'algoritmo, per ognuna di esse verrà proposto un vero e proprio algoritmo $\mathcal{MR}$ di cui verrà analizzato il tempo sequenziale su H macchine.

Conteggio occorrenze

Le prime due fasi del counting sort sequenziale prevedono l'inizializzazione di C e il conteggio degli elementi. Questo verrà fatto in due round con un algoritmo $\mathcal{MR}$. Il conteggio è di fatto un problema di word count ma non possiamo la utilizzare la procedura vista in 4.1.2 perché non è un algoritmo $\mathcal{MR}$.

Come già detto il vettore A viene diviso in $N^{1-\epsilon}$ sottovettori e assegnati a altrettanti reducer. Nel primo round ogni reducer conterà gli elementi distinti nel suo sottovettore creando una sorta di C parziale, che chiamiamo C_{par}. Al secondo round useremo k reducer per combinare i vettori C_{par} creati al round precedente per ottenere il C finale. In dettaglio:

- **Mapper 1**: Invia ogni coppia $\langle i_A, A[i_A]\rangle$ al reducer δ_j dove $(j-1)N^{\epsilon} \leq i_A \leq jN^{\epsilon} - 1$.
- **Recucer 1**: Il reducer δ_j riceve il sottovettore $A_{[((j-1)N^{\epsilon}):(jN^{\epsilon}-1)]}$ sotto forma di N^{ϵ} coppie $\langle i_A, A[i_A]\rangle$. Scorre tali coppie per ottenere le coppie $\langle i_{C_{Par_j}}, C_{Par_j}[i_{C_{Par_j}}]\rangle$, dove:

  $$C_{Par_j}[i_{C_{Par_j}}] = |\{A[t] = i_{C_{Par_j}} : t \in [(j-1)N^{\epsilon}), (jN^{\epsilon} - 1)]\}|$$

  Infine ritorna in output sia le coppie $\langle i_A, A[i_A]\rangle$ ricevute che le coppie $\langle i_{C_{Par_j}}, C_{Par_j}[i_{C_{Par_j}}]\rangle$ create.
- **Mapper 2**: Riceve due tipi di coppie, $\langle i_A, A[i_A]\rangle$ e $\langle i_{C_{Par_j}}, C_{Par_j}[i_{C_{Par_j}}]\rangle$. Invia ogni coppia $\langle i_A, A[i_A]\rangle$ al reducer δ_j dove $(j-1)N^{\epsilon} \leq i_A \leq jN^{\epsilon}-1$. Ogni coppia $\langle i_{C_{Par_j}}, C_{Par_j}[i_{C_{Par_j}}]\rangle$ viene inviata al reducer δ_{i_C}.
- **Reducer 2**: Le coppie ricevute di tipo $\langle i_A, A[i_A]\rangle$ vengono emesse nuovamente. Il reducer riceve inoltre un insieme di coppie $\langle i_{C_{Par_1}}, C_{Par_1}[i_{C_{Par_1}}]\rangle, \langle i_{C_{Par_2}}, C_{Par_2}[i_{C_{Par_2}}]\rangle, \ldots, \langle i_{C_{Par_q}}, C_{Par_j}[i_{C_{Par_j}}]\rangle$ con $q \leq N^{1-\epsilon}$, queste vengono usate per calcolare $C[i_C] = \sum_{t=1}^{N^{1-\epsilon}} C_{Par_t}[i_{C_{Par_t}}]$ e poi ritornate in output. Viene inoltre inviata una nuova coppia $\langle i_C, C[i_C]\rangle$.

L'output dell'algoritmo è composto da N coppie $\langle i_A, A[i_A]\rangle$ che rappresentano il vettore A in input, k coppie $\langle i_C, C[i_C]\rangle$ che compongono il vettore C e un certo numero di coppie $\langle i_{C_{Par_q}}, C_{Par_q}[i_{C_{Par_q}}]\rangle$. Il numero di queste ultime è sicuramente minore di $N^{1-\epsilon} \cdot N^{\epsilon}$, perché sono create dagli $N^{1-\epsilon}$ reducer al primo round e ogni reducer ne può creare al più una per ognuno degli N^{ϵ} elementi di A a disposizione. Ci riferiremo alla coppia $\langle i_{C_{Par_q}}, C_{Par_q}[i_{C_{Par_q}}]\rangle$ come alla posizione i_C del vettore C_{Par_q}. Teniamo questi $N^{1-\epsilon}$ vettori per le successive fasi dell'algoritmo.

Dimostriamo adesso che l'algoritmo è $\mathcal{MR}$:

Teorema 5.0.2. *L'algoritmo per il conteggio delle occorrenze è un algoritmo $\mathcal{MR}$ e il suo tempo sequenziale su H macchine è:*

$$TSH = O\left(\frac{N^{2-\epsilon}}{H}\right)$$

Dimostrazione. Un reducer nel primo round riceve N^{ϵ} coppie di A e crea il vettore C_{Par}, questo in realtà non è un vettore perché se lo fosse occuperebbe spazio $k = O(N)$ e l'algoritmo violerebbe il vincolo di memoria. C_{Par} di fatto sarà un qualche tipo di dizionario, come ad esempio un albero AVL. Questo occupa spazio pari al numero di elementi inseriti, che sono meno di N^{ϵ}. Nel secondo round il reducer δ_i riceve al più $N^{1-\epsilon}$ coppie del tipo $C_{Par_j}[i]$ e al più N^{ϵ} coppie di elementi di A, quindi usa un totale di $N^{\epsilon} + N^{1-\epsilon} = O(N^{1-p})$ record, con $p \in (0, 1)$. Segue che l'algoritmo è $\mathcal{MR}$.

Il fatto di usare dei dizionari implica che nel primo round i reducer impiegheranno tempo $N^{\epsilon} \log N^{\epsilon}$ per scorrere A e creare i dizionari C_{Par}. Il tempo sequenziale del primo round è:

$$TSH_1 = \left(\frac{N}{H} + \frac{N^{1-\epsilon}}{H} \cdot N^{\epsilon} \log N^{\epsilon}\right) = O\left(\frac{N}{H} \log N^{\epsilon}\right)$$

Nel secondo round il tempo speso da ogni reducer è pari al tempo per scorrere e sommare un numero di coppie minore o uguale a $N^{1-\epsilon}$, segue che:

$$TSH_2 = \left(\frac{N + N^{\epsilon} \cdot N^{1-\epsilon}}{H} + \frac{k}{H} N^{1-\epsilon}\right) = O\left(\frac{N^{2-\epsilon}}{H}\right)$$

Tra i due round, il secondo ha tempo maggiore asintoticamente infatti:

$$\frac{N}{H} \log N^{\epsilon} < \frac{N^{2-\epsilon}}{H} \Leftrightarrow N \log N^{\epsilon} < N^{2-\epsilon} \Leftrightarrow \log N < \frac{N^{1-\epsilon}}{\epsilon}$$

□

Somme prefisse di C

Dopo aver applicato l'algoritmo per il conteggio conosciamo C e dobbiamo calcolare le somme prefisse di C. Utilizzeremo l'algoritmo 2 proposto nel

paragrafo 4.2.3. Abbiamo dimostrato che tale procedura è un algoritmo $\mathcal{MR}$ e sappiamo che il suo tempo sequenziale su H macchine è $O\left(\frac{k^{1+\epsilon}}{H}\right)$, che per $k = O(N)$ è pari a $O\left(\frac{N^{1+\epsilon}}{H}\right)$.

Bisogna precisare che durante le fasi dell'algoritmo per le somme prefisse, le coppie che rappresentano il vettore di input A e i vettori C_{Par} devono essere preservate, cioè mandate dai mapper ai reducer e dai reducer ai mapper. Questo implica un passaggio di al più $N + N^{1-\epsilon} \cdot N^{\epsilon} = N + N$ coppie. Ma l'algoritmo 2 già si occupava di almeno $O(k)$ coppie, quindi in totale dovrà occuparsi di $O(k + N + N) = O(N)$ coppie. Ciò implica che la complessità dell'algoritmo non varia da quanto dimostrato nel paragrafo 4.2.3.

Ordinamento di A

Applicando le due procedure viste nei paragrafi precedenti abbiamo a disposizione il vettore C delle somme prefisse, il vettore A di input nonché gli $N^{1-\epsilon}$ vettori C_{Par}. In questo paragrafo vedremo come trasformare l'ultimo ciclo for dell'algoritmo 3 in un algoritmo $\mathcal{MR}$.

Utilizziamo $N^{1-\epsilon}$ reducer, ogni reducer δ_j prende il suo sottovettore $A_{[((j-1)N^{\epsilon}):(jN^{\epsilon}-1)]}$ e la parte di C che gli occorre per ordinarlo. Bisogna precisare che non è sufficiente inviare le corrette posizioni di C ai reducer, infatti sarà necessario prima modificarle per ottenere differenti valori per differenti reducer. Ad esempio supponiamo che i reducer δ_1 e δ_2 hanno entrambi degli elementi R da ordinare, in particolare il vettore $A_{[0:N^{\epsilon}-1]}$ contiene 2 occorrenze di R e il vettore $A_{[N^{\epsilon}:2N^{\epsilon}-1]}$ ne contiene una (vedi figura 5.1). Se $C[R] = I$ e inviamo il valore I a entrambi i reducer, il primo scriverà i suoi record in posizione $I, I-1$ e il secondo in posizione I. In questo modo dei record sono stati sovrascritti e sono andati persi, e alcune posizioni di B sono rimaste vuote (vedi figura 5.2). Questo non sarebbe successo se al primo reducer fosse stato mandato il valore I e al secondo il valore $I - 2$.

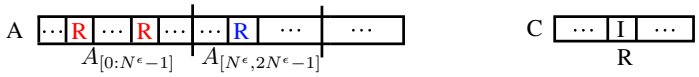

Figura 5.1

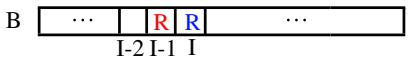

Figura 5.2

Come evidenziato nell'esempio dovremmo creare dei vettori C modificati per ogni reducer, che chiameremo C_{Mod}. Da notare che il vettore C_{Mod_1} del primo reducer è proprio C, mentre al secondo dovremmo sottrarre dai

valori di C il numero di elementi contenuti in $A_{[0:N^\epsilon-1]}$, il terzo sarà ottenuto sottraendo ai valori di C_{Mod_2} il numero di elementi contenuti in $A_{[N^\epsilon:2N^\epsilon-1]}$ e così via. Ricordiamo che dalle prime due fasi sono stati preservati i vettori C_{Par} che contengono proprio il numero di occorrenze degli elementi nei sottovettori di A, quindi basterà combinare i vettori C_{Par} e il vettore C per ottenere i C_{Mod}.

L'algoritmo di ordinamento si compone di due round, nel primo utilizza k reducer per creare i vettori C_{Mod}, durante il secondo si usano $N^{1-\epsilon}$ reducer, il j-esimo reducer ordinerà gli elementi di $A_{[((j-1)N^\epsilon):(jN^\epsilon-1)]}$ usando C_{Mod_j}. In dettaglio:

- **Mapper 1**: Riceve in input tre tipi di coppie: $\langle i_A, A[i_A]\rangle$, $\langle i_C, C[i_C]\rangle$ e $\langle i_{C_{Par_t}}, C_{Par_t}[i_C]\rangle$.
  Invia le coppie $\langle i_A, A[i_A]\rangle$ al reducer δ_j dove $(j-1)N^\epsilon \leq i_A \leq jN^\epsilon - 1$.
  Ogni coppia $\langle i_{C_{Par_j}}, C_{Par_j}[i_C]\rangle$ viene inviata al reducer δ_{i_C}.
  Le coppie $\langle i_C, C[i_C]\rangle$ vengono mandate al reducer i_C.

- **Reducer 1**: Il reducer riceve e manda in output ogni coppia del tipo $\langle i_A, A[i_A]\rangle$. Il reducer δ_j con $j = i_C$ riceve le coppie $\langle i_{C_{Par_1}}, C_{Par_1}[i_C]\rangle, \langle i_{C_{Par_2}}, C_{Par_2}[i_C]\rangle, \ldots, \langle i_{C_{Par_q}}, C_{Par_q}[i_C]\rangle$ e la coppia $\langle i_C, C[i_C]\rangle$, scorre tali coppie creando $q \leq N^{1-\epsilon}$ coppie $\langle i_{C_{Mod_t}}, C_{Mod_t}[i_C]\rangle$ dove:

  - $C_{Mod_1}[i_C] = C[i_C]$
  - $C_{Mod_t}[i_C] = C_{Mod_{t-1}}[i_C] - C_{Par_{t-1}}[i_C] \quad \forall t \in (1, q]$

  Le coppie del tipo $\langle i_{C_{Mod_t}}, C_{Mod_t}[i_C]\rangle$ vengono emesse in output.

- **Mapper 2**: Riceve in input due tipi di coppi: $\langle i_A, A[i_A]\rangle$ e $\langle i_{C_{Mod_t}}, C_{Mod_t}[i_{C_{Mod_t}}]\rangle$. Manda le coppie $\langle i_A, A[i_A]\rangle$ al reducer δ_j dove $(j-1)N^\epsilon \leq i_A \leq jN^\epsilon - 1$.
  Ogni coppia $\langle i_{C_{Mod_t}}, C_{Mod_t}[i_{C_{Mod_t}}]\rangle$ viene inviata al reducer δ_t.

- **Reducer 2**: Il reducer δ_j prende in input il suo sottovettore $A_{[((j-1)N^\epsilon):(jN^\epsilon-1)]}$ sottoforma di coppie $\langle i_A, A[i_A]\rangle$ e le q coppie $\langle i_{C_{Mod_j}}, C_{Mod_j}[i_C]\rangle$, con $q \leq N^\epsilon$, da cui crea un dizionario C_{Mod_j}. Scorre le coppie del sottovettore di A e determina la nuova posizione degli elementi in B, creando le coppie $\langle i_B, B[i_B]\rangle$ secondo la regola:

  $$B[C_{Mod_j}[A[i_A]]] \leftarrow A[i_A] \qquad \forall \langle i_A, A[i_A]\rangle \quad \text{ricevuta in input}$$

  Le N^ϵ coppie così create vengono restituite in output.

Dimostriamo che la procedura di due round appena descritta è un algoritmo $\mathcal{MR}$.

Teorema 5.0.3. *L'algoritmo per l'ordinamento di A è un algoritmo $\mathcal{MR}$ e il suo tempo sequenziale su H macchine è:*

$$TSH = O\left(\frac{N^{2-\epsilon}}{H}\right)$$

Dimostrazione. Durante il primo round i k reducer ricevono al più $N^{1-\epsilon}$ coppie del tipo $\langle i_{C_{Par_q}}, C_{Par_q}[i_C]\rangle$ e altre N^ϵ coppie di elementi di A, mentre nel secondo solo le N^ϵ coppie di A più le N^ϵ coppie del proprio C_{Mod}. Sia i reducer che i mapper non usano altra memoria se non quella occupata dall'input, poiché $N^\epsilon + N^{1-\epsilon} = O(N^{1-c})$ per un qualche $c \in (0,1)$, l'algoritmo è $\mathcal{MR}$. I mapper nel primo e nel secondo round si occupano di al più $N + N^\epsilon \cdot N^{1-\epsilon} = 2N$ coppie, ogni coppia ha dimensione inferiore a $N^{1-\epsilon}$ e viene processata in tempo costante. Questo implica che i mapper rispettano il limite sulla memoria e che il loro tempo di esecuzione su H macchine è $O(\frac{N}{H})$.
La complessità del primo round è:

$$TSH_1 = \left(\left\lceil\frac{N}{H}\right\rceil + \left\lceil\frac{k}{H}\right\rceil N^{1-\epsilon}\right) = O\left(\frac{N^{2-\epsilon}}{H}\right)$$

Nel secondo round il tempo di esecuzione cresce perché bisogna creare il dizionario C_{Mod}, il tempo sequenziale diventa quindi:

$$TSH_2 = \left(\left\lceil\frac{N}{H}\right\rceil + \left\lceil\frac{N^\epsilon}{H}\right\rceil N^{1-\epsilon}\log N^\epsilon\right) = O\left(\frac{N}{H}\log N^\epsilon\right)$$

Banalmente il tempo sequenziale del primo round è asintoticamente maggiore, infatti:

$$\frac{N}{H}\log N^\epsilon < \frac{N^{2-\epsilon}}{H} \Leftrightarrow \log N^\epsilon < N^{2-\epsilon}$$

□

Algoritmo completo

L'algoritmo 4 presenta lo schema completo della procedura per il counting sort su MapReduce e in figura 5.7 vediamo un esempio di esecuzione dell'algoritmo.

Per il teorema 4.1.2 questo è un algoritmo $\mathcal{MR}$ il cui tempo sequenziale su H macchine è pari alla somma dei tempi degli algoritmi che lo compongono:

$$TSH = O\left(\frac{N^{2-\epsilon}}{H}\right) + O\left(\frac{N^{1+\epsilon}}{H}\right) + O\left(\frac{N^{2-\epsilon}}{H}\right) = O\left(\max\left\{\frac{N^{2-\epsilon}}{H}, \frac{N^{1+\epsilon}}{H}\right\}\right)$$

Il tempo sequenziale del counting sort classico è $O(N)$, questo rispetto al tempo sequenziale del nostro algoritmo è peggiore se:

$$\left(\epsilon < \frac{1}{2}\right) \qquad \frac{N^{2-\epsilon}}{H} < N \quad \Leftrightarrow \quad H > N^{1-\epsilon}$$

Algorithm 4 Counting sort, algoritmo MR.
Input: coppie $\langle i_A, A[i_A]\rangle$ $\forall i_A \in [1, N]$.

- FASI 1-2
Procedura di conteggio occorrenze su coppie $\langle i_A, A[i_A]\rangle$;
Output: coppie $\langle i_C, C[i_C]\rangle$ $\forall i_c \in (1, k)$;
- FASE 3
Algoritmo 2 per calcolo somme prefisse delle coppie $\langle i_C, C[i_C]\rangle$
Output: coppie $\langle i_C, C_s[i_C]\rangle$ $\forall i_c \in (1, k)$;
- FASE 4
Algoritmo per l'ordinamento delle coppie $\langle i_A, A[i_A]\rangle$ usando le coppie $\langle i_C, C_s[i_C]\rangle$.
Output: N coppie $\langle i_B, B[i_B]\rangle$.

$$\left(\epsilon > \frac{1}{2}\right) \qquad \frac{N^{1+\epsilon}}{H} < N \quad \Leftrightarrow \quad H > N^{\epsilon}$$

Cioè l'algoritmo proposto ha prestazioni migliori della procedura sequenziale se il numero di macchine è maggiore almeno di $\sqrt{N}$. Da notare che il motivo principale per cui tale richiesta è così onerosa è che abbiamo considerato $k = O(N)$. Si capirà meglio questo aspetto nel paragrafo 5.1.2 dove presenteremo il counting sort per $k = 10$ e vedremo come le prestazioni migliorano.

Stabilità

A differenza del counting sort sequenziale, l'algoritmo presentato non mantiene la proprietà di stabilità. Ci sono due punti dell'algoritmo, in particolare nell'ultima fase, che fanno perdere questa proprietà.
Il primo punto si trova nel round 1 della fase di ordinamento, quando vengono creati i vettori C_{Mod}. Supponiamo ad esempio che il record E appaia due volte, una volta nel sottovettore $A_{[0:N^{\epsilon}-1]}$ e una volta nel sottovettore $A_{[N^{\epsilon}:2N^{\epsilon}-1]}$ e supponiamo sia $C[E] = I$ (vedi figura 5.3). Nel round 1 della fase di ordinamento il reducer δ_E si occuperà di creare le posizioni $C_{Mod_1}[E]$ e $C_{Mod_2}[E]$, in queste inserirà in modo arbitrario i valori I e $I-1$. Se assegna il valore I al vettore C_{Mod_1} e $I-1$ al vettore C_{Mod_2}, all'ultimo round, il δ_1 assegna il suo record alla posizione I di B mentre il reducer δ_2 scrive il suo record E nella posizione $I-1$ di B, invertendo di fatto l'ordine dei record rispetto alle posizioni iniziali (vedi figura 5.4). Per risolvere questo problema è sufficiente che nel round 1 dell'algoritmo di ordinamento ogni reducer ordini i valori C_{Par_t} ricevuti in ordine decrescente. In questo modo nell'esempio, i valori assegnati sarebbero stati $C_{Mod_2} = I$, $C_{Mod_1} = I-1$.

Anche con l'accorgimento visto sopra non siamo in grado di garantire la proprietà di stabilità. Infatti se un reducer deve ordinare più occorrenze di un elemento, queste non vengono assegnate con lo stesso ordine presente in A. Supponiamo ad esempio che nel vettore $A_{[0:N^{\epsilon}-1]}$ ci siano due occorrenze

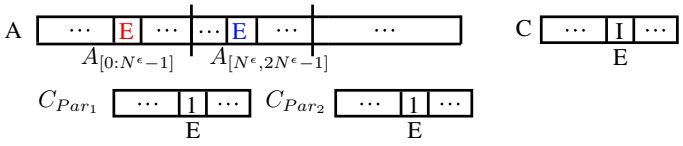

Figura 5.3

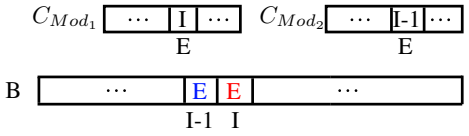

Figura 5.4

dell'elemento E e indichiamo con E_1 ed E_2 rispettivamente la prima e la seconda occorrenza nel vettore (vedi figura 5.5). Il reducer δ_1 dovrà ordinare gli elementi del sottovettore $A_{[0:N^\epsilon-1]}$ ma riceve tali elementi in un ordine arbitrario. Se riceve E_1 prima di E_2, e $C_{Mod} = I$, inserisce E_1 in posizione I e E_2 in posizione $I-1$ di B (vedi figura 5.6). Per risolvere tale problema è necessario ordinare gli N^ϵ elementi di A ricevuti.

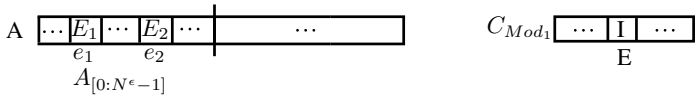

Figura 5.5

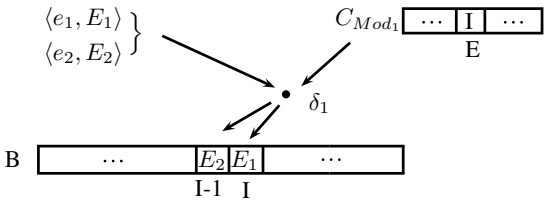

Figura 5.6

Le due modifiche proposte garantiscono la stabilità e mantengono l'algoritmo $\mathcal{MR}$, infatti non viene utilizzata memoria aggiuntiva. Neanche il tempo sequenziale cresce in quanto per entrambi le modifiche richiedono solo l'ordinamento di elementi a chiave intera. Ad esempio gli N^ϵ elementi di $A_{[((j-1)N^\epsilon):(jN^\epsilon-1)]}$ ricevuti dal reducer δ_j sottoforma di coppie $\langle i_A, A[i_A]\rangle$ possono essere ordinate creando un vettore di N^ϵ e inserendo i valori delle coppie, $A[i_A]$, scorrendo le chiavi, i_a, in tempo lineare.

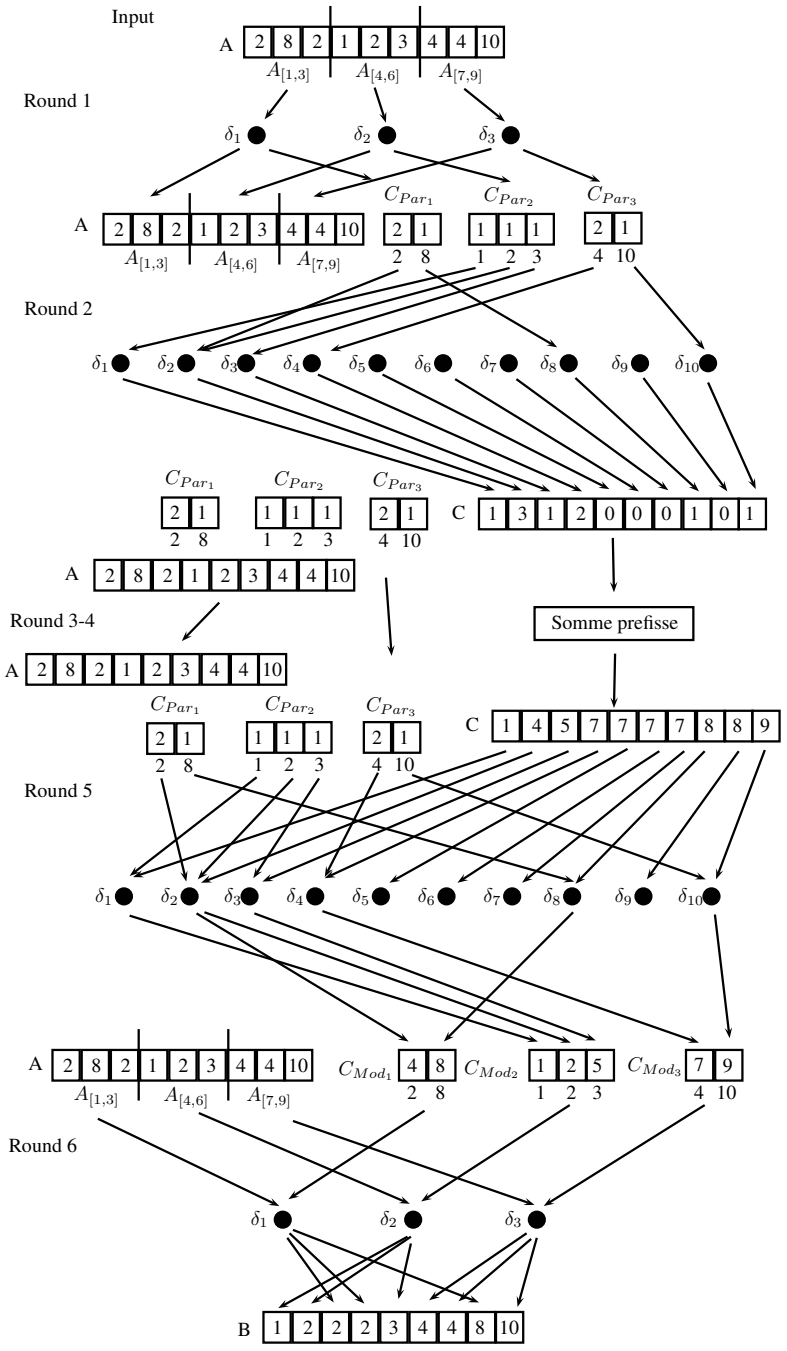

Figura 5.7: Esempio di esecuzione dell'algoritmo di counting sort su di un vettore A. Le frecce indicano a quali reducer i mapper inviano i dati. Nei round 2-5 si assume che il vettore A venga suddiviso tra i reducer come nel primo round.

5.1 Radix sort

Tra gli algoritmi di ordinamento classici non comparativi, oltre al counting sort, troviamo il *radix sort*. Una prima versione del radix sort risale al 1887 da un lavoro di Herman Hollerith per la macchine tabulatrici [36]. L'algoritmo è stato largamente utilizzato da gli anni '20 in poi per l'ordinamento di schede perforate. Più di recente il radix sort è stato usato per risolvere problemi in ambito parallelo, ad esempio in [9] viene usato all'interno di un algoritmo per la codifica e la decodifica degli alberi etichettati.

Presentiamo di seguito la versione classica dell'algoritmo radix sort (tratta da [12]) e quindi una versione parallela dell'algoritmo in MapReduce che sfrutta quanto ottenuto nel paragrafo 5.0.5 per il counting sort.

5.1.1 Radix sort sequenziale

L'algoritmo è in grado di ordinare N record a chiave intera con complessità $O(hN)$, dove h è il numero massimo di cifre delle N chiavi. L'idea dell'algoritmo è di ordinare le chiavi cifra a cifra, partendo dalla cifra meno significativa e ripetendo il procedimento h volte. Lo pseudocodice dell'algoritmo preso un vettore A in input è:

```
RadixSort(A, h):
    for i = 1 to h:
       Ordina A rispetto alla cifra i-esima
```

La correttezza dell'algoritmo dipende dal fatto che il passo di ordinamento dev'essere eseguito da un algoritmo stabile, come ad esempio il counting sort. Anche l'efficienza dell'algoritmo dipende da questo punto. Se ogni cifra è nell'intervallo $[1, k]$ usando il counting sort che ha complessità $O(N + k)$, è possibile ordinare N numeri in $O(hN + hk)$, cioè in tempo lineare.

5.1.2 Counting sort $\mathcal{MR}$ con $k = 10$

Come per il radix sort sequenziale useremo in MapReduce il counting sort come algoritmo di ordinamento stabile. Supporremo di dover ordinare record a chiave intera espressa in notazione decimale, per questo motivo non useremo l'algoritmo 4 visto nel paragrafo 5.0.5 per il counting sort. Infatti tale algoritmo è stato analizzato per $k \leq O(N)$, ma essendo in questo caso il numero di possibili cifre distinte pari a $k = 10$, le prestazioni possono essere migliori.

Rivediamo quindi le fasi dell'algoritmo 4 proponendo delle modifiche o semplicemente rianalizzando il tempo sequenziali sfruttando il fatto che $k = 10$.

Conteggio occorrenze

Supponiamo di ricevere in input un vettore A di N record con $k = 10$ chiavi distinte. Come per l'algoritmo 4 divideremo il vettore in $N^{1-\epsilon}$ sottovettori di dimensione N^{ϵ}, ognuno dei quali verrà gestito da uno degli $N^{1-\epsilon}$ reducer. Nella prima fase dell'algoritmo 2 viene creato il vettore C tale che $\forall i \in [1, k]$ $C[i] = |\{A[j] = i : j \in [1, N]\}|$. Come visto nel paragrafo 5.0.5 si procede in due round, nel primo i reducer ricevono i sottovettori e creano dei vettori parziali C_{Par}, mentre nel secondo k reducer sommano le posizioni dei C_{Par} ottenendo il vettore C.

Rispetto a quanto definito nel paragrafo 5.0.5 è sufficiente modificare solo il comportamento dei reducer nel primo round. Durante il primo round ogni reducer scorre il proprio sottovettore e crea un dizionario per rappresentare il vettore C_{Par}. Notiamo che non è necessario usare un dizionario in quanto rappresentare C_{Par} come vettore richiede solo $O(k)$ memoria aggiuntiva, cioè una quantità costante.

Questa modifica mantiene l'algoritmo definito in 5.0.5 nella classe $\mathcal{MR}$, infatti ogni reducer usa una quantità di memoria minore o uguale di quella usata nella versione originale.

Considerando il fatto che $k = 10$ il tempo sequenziale su H macchine dell'algoritmo appena descritto diventa per il primo round:

$$TSH_1 = \left(\left\lceil \frac{N}{H} \right\rceil + \left\lceil \frac{N^{1-\epsilon}}{H} \right\rceil \cdot N^{\epsilon} \right) \leq O\left(\frac{N}{H} \right)$$

e per il secondo:

$$TSH_2 = \left(\left\lceil \frac{N + k \cdot N^{1-\epsilon}}{H} \right\rceil + \left\lceil \frac{k}{H} \right\rceil N^{1-\epsilon} \right) \leq O\left(\frac{N}{H} \right)$$

Complessivamente l'algoritmo per il conteggio delle occorrenze ha tempo sequenziale:

$$TSH = O\left(\frac{N}{H} \right)$$

Somme prefisse C

Una volta calcolato il vettore C l'algoritmo di counting sort prevede il calcolo delle somme prefisse su C. Nell'algoritmo 4 si usa l'algoritmo 2 per questo scopo. Tuttavia notiamo che è in questo caso essendo la dimensione di C pari a $k = 10$ non è necessario invocare altre procedure. Infatti in un unico round potremmo passare il vettore C a un reducer che con k somme ne ottiene le somme prefisse.

Durante questo round, useremo la funziona map per mantenere i vettori A e C_{Par} in memoria e per mandare C a un reducer che calcola le somme prefisse. Il tempo sequenziale di questo round è:

$$TSH = \left(\left\lceil \frac{N + k \cdot N^{1-\epsilon}}{H} \right\rceil + \left\lceil \frac{N^{1-\epsilon}}{H} \right\rceil \cdot N^{\epsilon} \right) \leq O\left(\frac{N}{H}\right)$$

Da notare che usiamo comunque $N^{1-\epsilon}$ reducer, questi dovranno ricevere e reinviare i loro sottovettori di A.

Ordinamento di A

L'ultima fase del counting sort prevede l'ordinamento del vettore di input utilizzando il vettore C ottenuto nelle fasi precedenti. Come visto nel paragrafo 5.0.5, questa fase può essere eseguita da un algoritmo $\mathcal{MR}$ che opera in 2 round. Nel primo round usando C e i vettori C_{Par} si ottengono i vettori C_{Mod} che nel secondo round saranno usati dagli $N^{1-\epsilon}$ reducer per ordinare i sottovettori di A. Utilizzeremo qui esattamente lo stesso procedimento descritto in 5.0.5 (comprese le modifiche di stabilità del paragrafo 5.0.5), tuttavia rivediamo l'analisi dell'algoritmo.

Notiamo innanzitutto che essendo $k = 10$ nel primo round useremo solo 10 reducer, il che significa che il tempo sequenziale diventa:

$$TSH_1 = \left(\left\lceil \frac{N}{H} \right\rceil + \left\lceil \frac{10}{H} \right\rceil N^{1-\epsilon} \right) \leq O\left(\frac{N}{H}\right)$$

Per il secondo round il tempo sequenziale resta inalterato:

$$TSH_2 = \left(\left\lceil \frac{N}{H} \right\rceil + \left\lceil \frac{N^{\epsilon}}{H} \right\rceil N^{1-\epsilon} \right) \leq O\left(\frac{N}{H}\right)$$

Ciò implica che complessivamente:

$$TSH = O\left(\frac{N}{H}\right) + O\left(\frac{N}{H}\right) = O\left(\frac{N}{H}\right)$$

Algoritmo completo

L'algoritmo per il counting sort ricalca quello definito nell'algoritmo 4 ma utilizza le tre fasi descritte nei tre paragrafi precedenti. Per il teorema 4.1.2 sappiamo che il suo tempo sequenziale su H macchine è:

$$O\left(\frac{N}{H}\right) + O\left(\frac{N}{H}\right) + O\left(\frac{N}{H}\right) = O\left(\frac{N}{H}\right)$$

5.1.3 Algoritmo $\mathcal{MR}$ per il radix sort

L'algoritmo $\mathcal{MR}$ per il radix sort ordina un vettore A di N record a chiave intera, con h il massimo numero di cifre della chiavi. Per farlo utilizza h volte l'algoritmo appena definito per il counting sort, ordinado le cifre a partire dalla meno significativa. Grazie al teorema 4.1.2 tale algoritmo è $\mathcal{MR}$ poiché

itera un algoritmo $\mathcal{MR}$. La correttezza dell'algoritmo segue dal fatto che la struttura è identica a quella del radix sort sequenziale visto nel paragrafo 5.1.1.
Infine il tempo sequenziale su H macchine è pari a:

$$TSH = h \cdot \left(\frac{N}{H}\right)$$

Rispetto alla versione sequenziale del radix sort, questa risulta migliore per qualsiasi numero di macchine:

$$h \cdot \left(\frac{N}{H}\right) \leq h \cdot N \qquad \Leftrightarrow \qquad H \geq 1$$

Confrontiamo infine l'algoritmo per il radix sort con quello presentato nel paragrafo 5.0.5 per il counting sort. Se consideriamo N interi compresi nell'intervallo $[1, q]$ espressi in notazione decilame, usando la tecnica di radix sort siamo in grado di ordinare tali elementi in tempo sequenziale su H macchine:

$$TSH_{radix} = \log q \cdot \left(\frac{N}{H}\right)$$

Come presentato nel paragrafo 5.0.5 l'algoritmo di counting sort può ordinare lo stesso insieme di elementi in tempo sequenziale:

$$TSH_{counting} = O\left(\max\left\{\frac{N^{2-\epsilon}}{H}, \frac{N^{1+\epsilon}}{H}\right\}\right)$$

Supponendo di scegliere $\epsilon \geq 1/2$ (lo stesso può essere dimostrato per $\epsilon < 1/2$), il radix sort risulta più efficiente anche se $q = O(N)$, in quanto:

$$\frac{N^{1+\epsilon}}{H} > \log N \cdot \left(\frac{N}{H}\right) \Leftrightarrow N^{\epsilon} > \log N$$

Se invece supponiamo di esprimere in base $k = \Theta(N)$ gli N interi compresi nell'intervallo $[1, q]$ le prestazioni delle due tecniche si equivalgono. Infatti in questo caso il radix sort dovrebbe per forza usare l'algoritmo 4 per il counting sort impiegando così un tempo sequenziale su H macchine pari a:

$$TSH_{radix} = \left(\max\left\{\frac{N^{2-\epsilon}}{H}, \frac{N^{1+\epsilon}}{H}\right\}\right) \cdot \log_N q =$$

$$= \left(\max\left\{\frac{N^{2-\epsilon}}{H}, \frac{N^{1+\epsilon}}{H}\right\}\right) \cdot \frac{\log q}{\log N} = O\left(\max\left\{\frac{N^{2-\epsilon}}{H}, \frac{N^{1+\epsilon}}{H}\right\}\right)$$

In questo caso il numero di iterazioni del radix sort diventano costanti ma il tempo sequenziale peggiora.

Capitolo 6

Conclusioni

MapReduce è un framework che permette in modo semplice di progettare soluzioni parallele e distribuite per problemi su dati di grandi dimensioni. La semplicità coniugata alla potenza del framework l'hanno fatto diventare in breve tempo lo standard di fatto sia in campo industriale che in campo accademico. In contrasto col diffuso uso sperimentale, fin ora sono pochi i lavori che hanno tentato di mettere MapReduce sullo stesso piano dei modelli di computazione classici.
In questa tesi si è cercato di analizzare proprio questo aspetto, studiando i modelli, gli algoritmi e le tecniche proposte per MapReduce.

Nel capitolo 2 sono stati prima presentati i modelli classici di computazione parallela come PRAM e BSP, e quindi le caratteristiche dell'implementazione originale del fremework. Abbiamo inoltre analizzato le principali critiche a MapReduce.

Nel capitolo 3 si è cercato di raccogliere i tentativi di modellazione di MapReduce e gli algoritmi progettati usando tali modelli. Tra questi algoritmi troviamo delle simulazioni di PRAM e BSP che rappresentano un ponte tra i modelli classici e il framework di Google.

Tenendo conto di quanto analizzato nei primi due capitoli, è stato proposto un modello che raccoglie le nostre considerazioni sul framework, il modello $\mathcal{MR}$. Tale modello presenta come principale differenza rispetto ai suoi predecessori il fatto di essere stato progettato per l'analisi della complessità sequenziale. Di fondamentale importanza è la distinzione fatta tra le caratteristiche dell'algoritmo e quelle del cluster su cui si utilizzerà tale algoritmo. Come conseguenza di questa distinzione è stata ideata la misura di complessità chiamata *tempo sequenziale su H macchine* (TSH), che permette di esprimere il tempo asintotico dell'algoritmo in funzione dell'input e del numero di macchine del cluster.
Tale misura permette non solo di confrontare algoritmi definiti per modelli differenti, ma anche un confronto asintotico con algoritmi sequenziali. Per dimostrare la bontà dell'approccio proposto vengono presentati due esempi di

analisi con TSH. Per primo abbiamo analizzato una classica soluzione per il *word count*, questo esempio ha evidenziato come è facile violare le limitazioni imposte dal modello $\mathcal{MR}$. Come secondo esempio è stato presentato un innovativo algoritmi per le somme prefisse, il TSH di questo algoritmo è stato confrontato con quello di un algoritmo MapReduce già esistente e con la soluzione sequenziale.

Infine si è pensato di affrontare il problema dell'ordinamento. Sono per questo stati ideati due nuovi algoritmi ispirati ai classici radix sort e counting sort sequenziali. Tali algoritmi riescono ad essere più efficienti delle rispettive soluzioni sequenziali se il cluster su cui vengono utilizzati ha una certa disponibilità di macchine. Anche queste analisi mostrano come il tempo TSH sia una misura semplice da calcolare e potente dal punto di vista del confronto.

Bibliografia

[1] B. Abbott et al. LIGO Scientific Collaboration, *Einstein@Home search for periodic gravitational waves in LIGO S4 data,* Physical Review, 2009, volume 79.

[2] A. Aggarwal, A. K. Chandra, and M. Snir, *On communication latency in PRAM computations,* SPAA 89: 11-21.

[3] A. Aggarwal, A. K. Chandra, and M. Snir, *Communication complexity of PRAMs,* Theoretical Computer Science 71(1):3-28.

[4] J. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[5] J. P. Anderson , S. A. Hoffman , J. Shifman , R. J. Williams, *D825 - a multiple-computer system for command and control,* in Fall joint computer conference (1962): 86-96.

[6] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, Rafael Pasquini, *Incoop: MapReduce for Incremental Computations,* SOCC'11.

[7] R. S. Bird, *Lectures on constructive functional programming,* Constructive Methods in Computer Science (1989): 151-216.

[8] BlueGene/L Team, *An Overview of the BlueGene/L Supercomputer,* Supercomputing ACM/IEEE 2002 Conference: 6-22.

[9] S. Caminiti, I. Finocchi, R. Petreschi, *On coding labeled trees,* in Journal Theoretical Computer Science archive, Volume 382 Issue 2, 2007, Elsevier Science Publishers Ltd.

[10] D. K. G. Campbell, *Towards a unified ed parallel architecture class,* Technical Report RR 294, Department of Computer Science, University of Exeter, 1994.

[11] F. Chierichetti, R. Kumar, A. Tomkins, *Max-Cover in Map-Reduce,* IW3C2, 2010.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms,* The MIT Press, seconda edizione, capitolo 9.

[13] A. Cary, Z. Sun, V. Hristidis, N. Rishe, *Experiences on Processing Spatial Data with MapReduce,* Proceedings in International Conference on Scientific and Statistical Database Management 21(2009).

[14] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, K. Olukotun, *Map-reduce for machine learning on multicore,* In Proceedings NIPS 2006: 281–288.

[15] R. Cole, O. Zajicek, *The APRAM: Incorporating asynchrony into the PRAM model,* ACM Symposium on Parallel Algorithms and Architectures, 1989.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Sub- ramonian, T. Von Eicken, *LogP: Towards a realistic model of parallel computation,* in ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 4 (1993): 1-12.

[17] A. Das, M. Datar, A. Garg, S. Rajaram, *Google news personalization: Scalable online collaborative filtering,* in Proceedings of WWW 2007: 271–280.

[18] J. Dean, S. Ghemawat, *MapReduce: Simplified data processing on large clusetrs,* in OSDI'04: 137-150.

[19] J. Dean, S. Ghemawat, *MapReduce: A flexible data processing tool,* in Communications of the ACM, 2010, 53(1): 72-77.

[20] P. de la Torre, C. P. Kruskal, *Towards a single model of e cient computation in real parallel machines,* Parallel Architectures and Languages Europe(PARLE '91).

[21] D. DeWitt, M. Stonebraker, *MapReduce: A major step backwards,* databasecolumn.com, 2008-08-27.

[22] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. Fox, *Twister: a runtime for iterative MapReduce,* in Proceedings HPDC '10.

[23] T. Elsayed, J. Lin, D. W. Oard, *Pairwise document similarity in large collections with MapReduce,* in Proceedings 46th ACL/HLT 2008: 265–268.

[24] J.Feldman, S. Muthukrishanan, A. Sidiropoulos, C. Stein, Z. Svitkina, *On distributing symmetric streaming computations,* ACM Trans. Algorithms (2010): 6:1-19.

[25] S. Fortune, J. Wyllie, *Parallelism in random access machines,* in ACM Annual Symposium on Theory of Computing, 10(1978): 114-118.

[26] S. Ghemawat, H. Gabioff, S, Leung, *The Google file system,* in Symposium on Operating Systems Principles,19 (2003): 29-43.

[27] M. T. Goodrich, N. Sitchinava, Q. Zhang, *Sorting, searching, and simulation in the MapReduce framework,* articolo non pubblicato (2010).

[28] M. T. Goodrich, *Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry,* MASSIVE 2010.

[29] M. T. Goodrich, M. Mitzenmacher *MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations,* articolo non pubblicato (2010).

[30] Google code - MapReduce tutorial, *http://code.google.com/intl/it-IT/edu/parallel/mapreduce-tutorial.html.*

[31] Gpugrid, *http://www.gpugrid.net/.*

[32] P. Graham, *On Lisp,*Prentice Hall, 1993.

[33] G. Greiner, R. Jacob, *The efficiency of Mapreduce in Parallel External Memory,* S.d. N.d.

[34] Hadoop *Hadoop wiki,* http://wiki.apache.org/hadoop/.

[35] T. Heywood, S. Ranka, *A practical hierarchical model of parallel computation,* Journal of Parallel and Distributed Computing 16(3):212-232.

[36] H. Hollerith, *Art Of Compiling Statistics* US 395781 (A), 1889.

[37] IBM, *IBM Type 704 Manual of operation,* Form 24-66661-1, IBM, 1956.

[38] G. Jorgensen, *Relational Database Experts Jump The MapReduce Shark,* typicalprogrammer.com, 2009-11-11.

[39] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, J. Leskovec, *HADI: Fast diameter estimation and mining in massive graphs with hadoop,* Technical Report CMU-ML-08-117, CMU, 2008.

[40] H. Karloff, S. Suri, S.Vassilvitskii, *A model of computation for MapReduce,* in SODA'10: 938-948.

[41] S. Lattanzi, B. Moseley, S. Suri, S. Vassilvitskii, *Filtering: A method for solving graph problems in MapReduce,* SPAA'11.

[42] J. J. Lin, *Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with MapReduce,* in Proceedings EMNLP 2008: 419-428.

[43] J. Lin, M. Schatz, *Design patterns for efficient graph algorithms in MapReduce,* in Proceeding MLG '10.

[44] R. M. C. McCreadie, C. Macdonald, I. Ounis, *On single-pass indexing with MapReduce,* in Proceedings SIGIR 32(2009): 742-743.

[45] J. M. Nash, P. M. Dew, M. E. Dyer, *Scalable portable computing using the WPRAM model,* Abstract Machine Models for Parallel and Distributed Computing, IOS Press, 1996: 47-62.

[46] E. Nestle, A. Inselberg, *The SYNAPSE $N+1$ System: architectural characteristics and performance data of a tightly-coupled multiprocessor system,* ISCA 1985.

[47] R. Pagh, F. Rodler, *Cuckoo hashing,* Journal of Algorithms (2004), 52:122–144.

[48] S. Papadimitriou, J. Sun, *Disco: Distributed co-clustering with Map-Reduce: A case study towards petabyte-scale end-to-end mining,* in Proceedings of ICDM, 8(2008): 512-521.

[49] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, *A Comparison of Approaches to Large-Scale Data Analysis,* Brown University. 2010-01-11.

[50] R. Petreschi *Dispense del corso di Algoritmi Avanzati,* http://twiki.di.uniroma1.it/twiki/view/AA/WebHome

[51] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, *Evaluating MapReduce for Multi-core and Multiprocessor Systems,* High Performance Computer Architecture, 2007.

[52] R. M. Russell, *The CRAY−1 Computer System,* Communications of the ACM (1978), v.21 n.1: 63-72.

[53] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson, *A new major SETI project based on Project Serendip data and 100,000 personal computers,* IAU Colloq. No. 161.

[54] D. P. Siewiorek, C. Gordon Bell, Allen Newell, *Computer Structures Principles and Examples,* McGraw-Hill computer science series: Cap 20

[55] C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos, *Doulion: Counting triangles in massive graphs with a coin,* in Knowledge Discovery and Data Mining, 2009.

[56] L. G. Valiant, *Bulk-Synchronous Parallel computers,* Technical Report TR-08-89, Aiken Computation Laboratory, Harvard University, 1989.

[57] W. A. Wulf, C. G. Bell, *C.mmp - A multi-mini-processor.* C-MU 1972.