MapReduce (1)

Presentazione del Framework

Algoritmi Avanzati
Anno Accademico 2015-2016

Lorenzo Spignoli
Francesco Andreozzi
Viviana Arrigoni

prof.ssa Rossella Petreschi

Contesto

L'aumento della quantità di dati disponibili ha reso cruciale la risoluzione di problemi su grandi quantità di dati.

La velocità dei processori non è l'unico fattore per incrementare le prestazioni di un processo. Una strategia alternativa è quella di parallelizzare le computazioni.

L'abbassamento del costo dell'hardware ha reso accessibili a tutti macchine mediamente potenti.

Introduzione

In questo contesto nasce nel 2004 MapReduce, un sistema progettato da Google, che permette di combinare computazione parallela, distribuita e sequenziale su dati di grande dimensione.

L'idea degli autori è quindi quella di costruire un framework semplice che si possa utilizzare su di un qualsiasi cluster di computers e per un qualsiasi tipo di problema e che liberi gli utenti dal vincolo di occuparsi di aspetti quali parallelizzazione, suddivisione e distribuzione dei task, buckup, comunicazione, fallimenti e recovery.

Framework (1)

Esigenza:

centinaia di computazioni al giorno che processano grandi quantità di dati, come documenti, logs o pagine web per ottenere dati strutturati.

Astrazione:

molte di queste computazioni sono piuttosto semplici. Le analogie riscontrate in queste computazioni hanno portato gli autori a creare un'astrazione inspirata alle primitive funzioni map e reduce del linguaggio funzionale Lisp.

Framework (2)

Una computazione può essere vista come una sequenza finita di round, ognuno dei quali è composto da una funzione di map e una di reduce:

$$< \mu_1 \& \rho_1, \mu_2 \& \rho_2, \ldots, \mu_v \& \rho_v >$$

Le funzioni di map e reduce

map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

Framework (3)

Durante la computazione di ogni round:

1. La funzione di Map prende in input una coppia chiave/valore e restituisce una coppia di valori intermedi chiave/valore.

2. La funzione Shuffle raggruppa tutti i valori intermedi con stessa chiave e la passa in input alla funzione di Reduce.

3. La funzione di Reduce prende in input una coppia di chiave intermedia/lista di valori, fa operazioni su questi valori e restituisce un insieme di valori.

Esempio (1)

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Esempio (2)

Input →
files con un documento per record

"document1", "to be or not to be"

funzione Map → input:
<nome$_{doc}$, contenuto$_{doc}$> e output:
<parola, 1>

funzione Reduce → input:
<parola, lista di 1> *e*
output: <parola, occorrenze$_{\#}$>

Esempio (2)

Input →
files con un documento per record

“document1”, “to be or not to be”

“to”, “1”
“be”, “1”
“or”, “1”
…

funzione Map → input:
<nome$_{doc}$, contenuto$_{doc}$> e output:
<parola, 1>

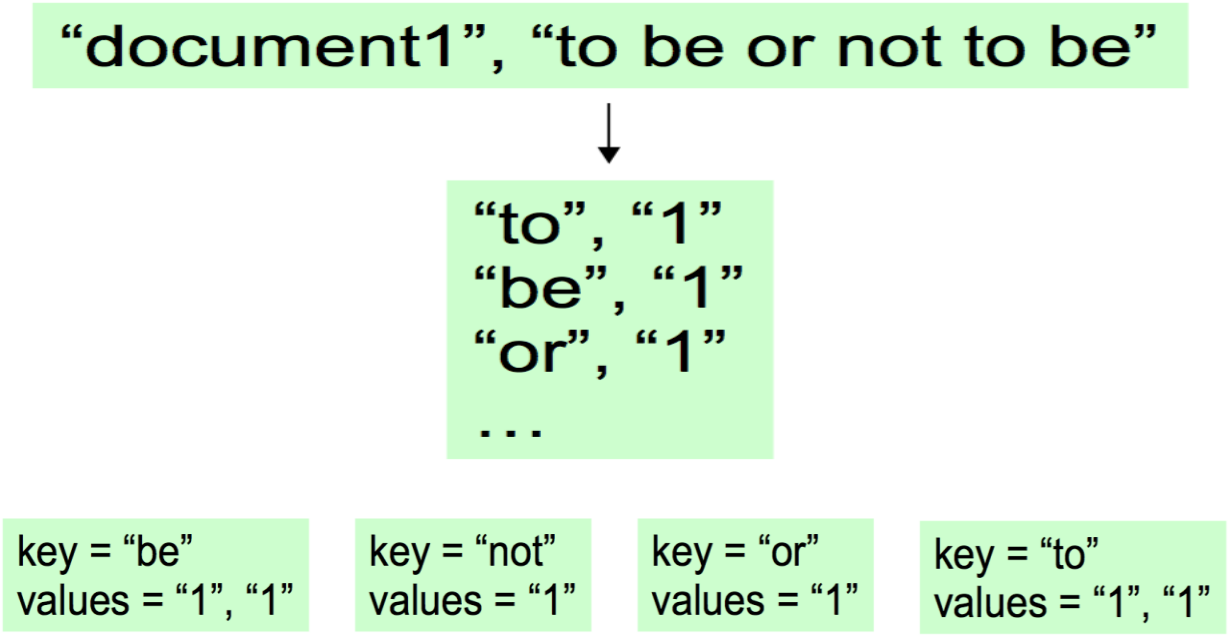
funzione Reduce → input:
<parola, lista di 1> *e*
output: <parola, occorrenze$_{\#}$>

Esempio (2)

Input →
files con un documento per record

funzione Map →
<nome$_{doc}$, contenuto$_{doc}$>
<parola, 1>

input:
e output:



funzione Reduce →
<parola, lista di 1>
output: <parola, occorrenze$_{\#}$>

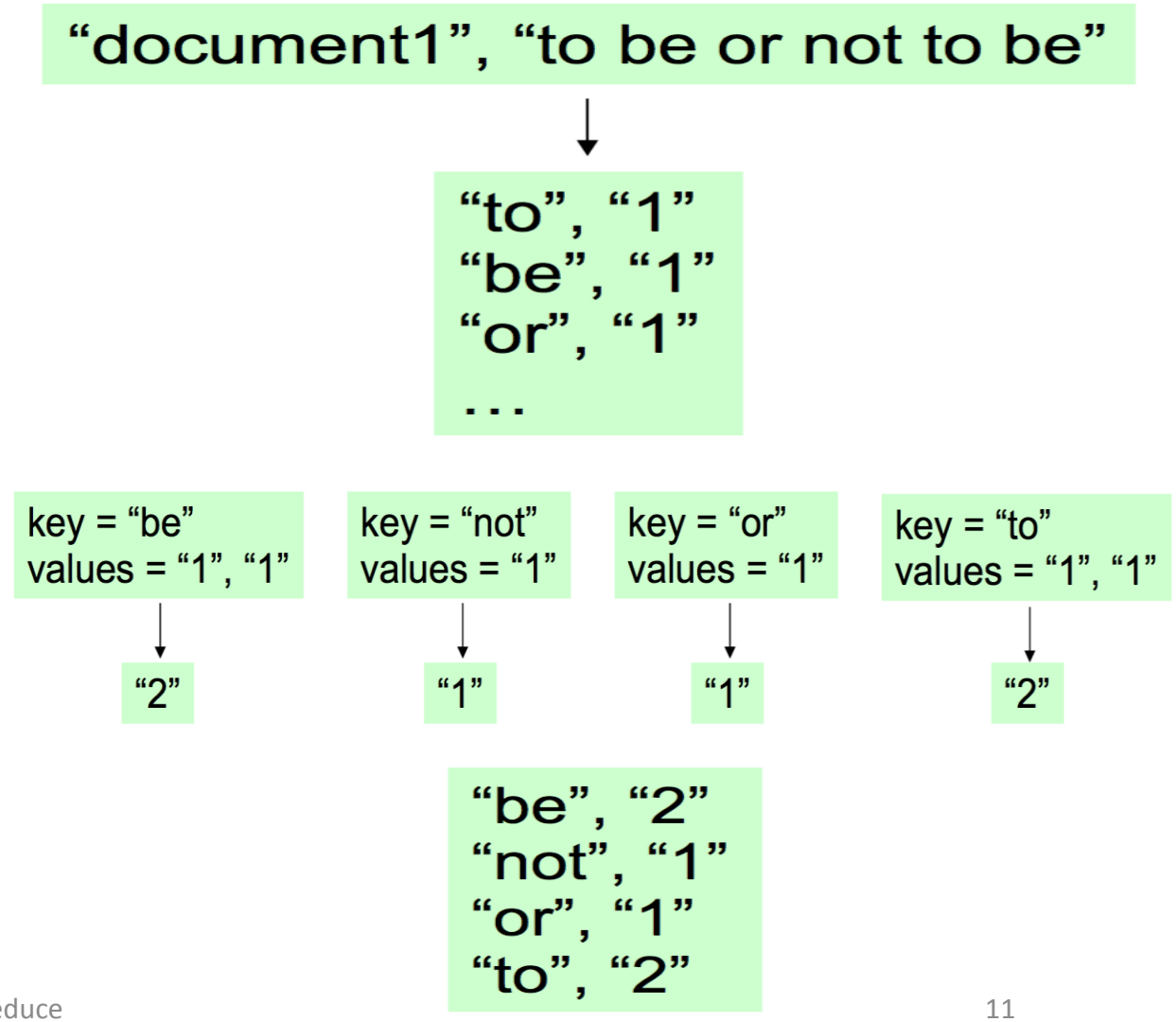
input:
e

Esempio (2)

Input →
files con un documento per record

funzione Map → input:
<nome$_{doc}$, contenuto$_{doc}$> e output:
<parola, 1>

funzione Reduce → input:
<parola, lista di 1> *e*
output: <parola, occorrenze$_{\#}$>



Funzione di combiner (1)

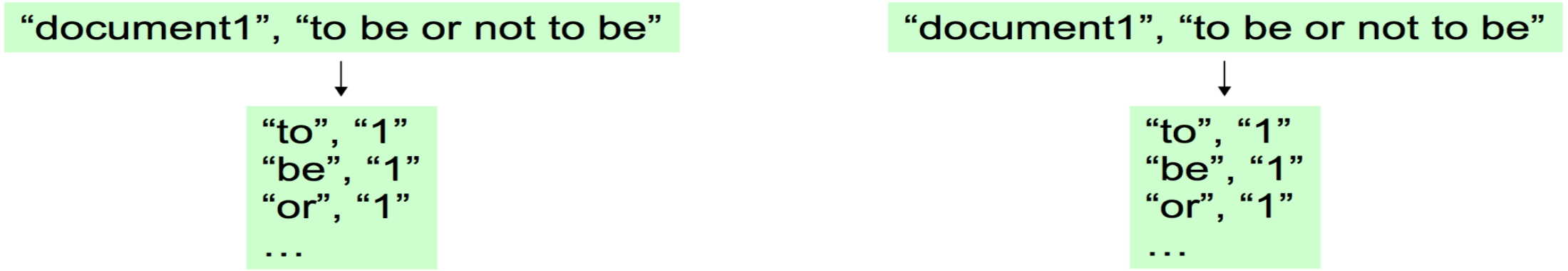
In alcuni casi c'è una pesante quantità di ripetizioni dei valori intermedi per ogni task di map. Per questi casi si è permesso all'utente di scrivere una funzione combiner.

N.B. La funzione di reduce deve essere associativa e commutativa.

Questa è eseguita dopo ogni funzione di map. Tipicamente il codice è lo stesso della funzione reduce, l'unica differenza sta in come la libreria MapReduce maneggia l'output delle due funzioni.

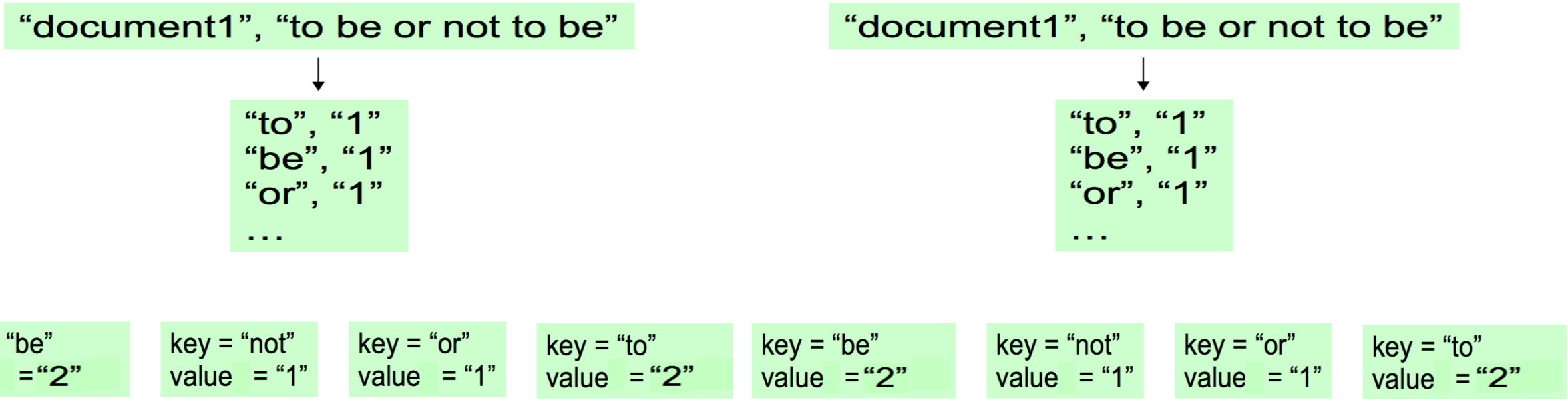
Eseguire la funzione di combiner accelera notevolmente l'esecuzione del processo.

Funzione Combiner (2)



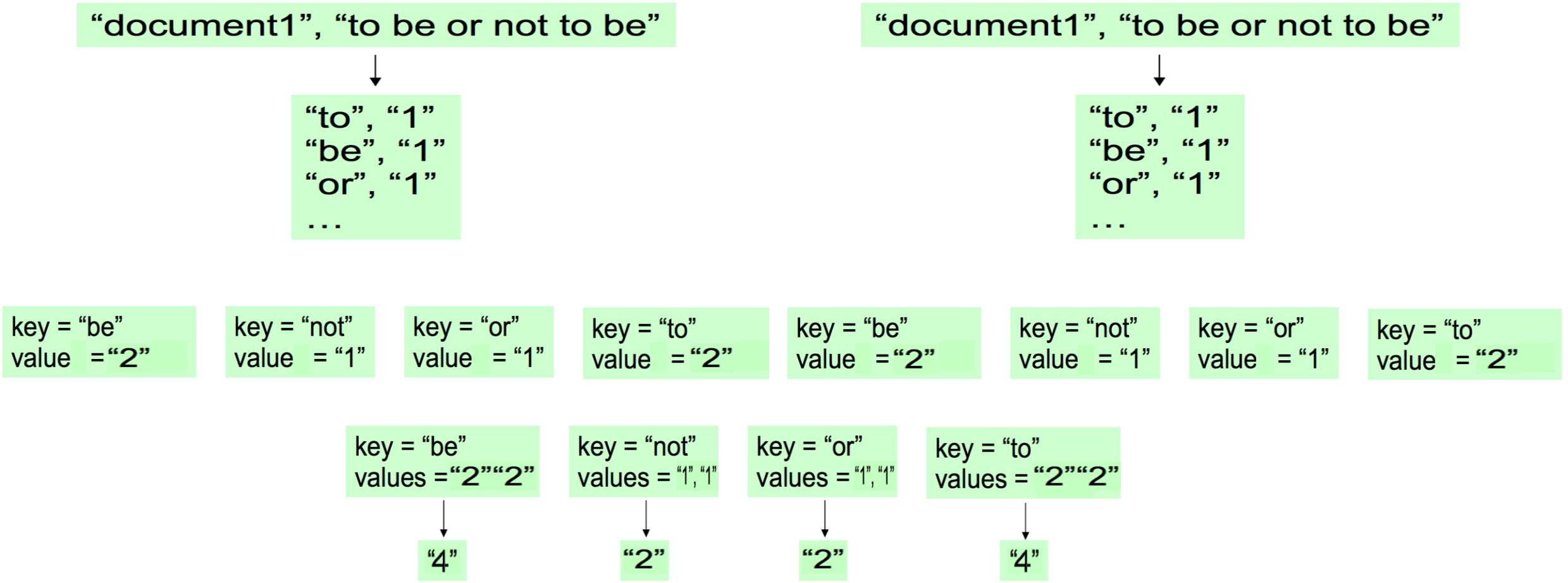
Esecuzione funzione di map

Funzione Combiner (2)



Esecuzione funzione di combiner

Funzione Combiner (2)



Esecuzione funzione di reduce (+ funzione shuffle)

Altri Esempi

Frequenza di accesso di URL:

- Map → processa i log delle pagine web richieste e restituisce la coppia *<URL, 1>*
- Reduce → raggruppa insieme tutte le coppie con stesso URL e restituisce la coppia *<URL, totale>*

Vettore di termini per host:

- Map → processa i documenti e restituisce la coppia *<hostname, vettore di termini>*
- Reduce → raggruppa insieme tutte le coppie con lo stesso hostname, scarta i termini meno frequenti e restituisce la coppia *<hostname, vettore di termini>*

Implementazione (1)

Sono possibili più di un'implementazione del MapReduce e non esiste l'implementazione corretta. Infatti, questa dipende dall'ambiante di lavoro in cui viene utilizzato!!

L'implementazione che è stata proposta è stata presa dall'articolo[*] di Jeffrey Dean e Sanjay Ghemawat e si basa sull'ambiente lavorativo di Google.

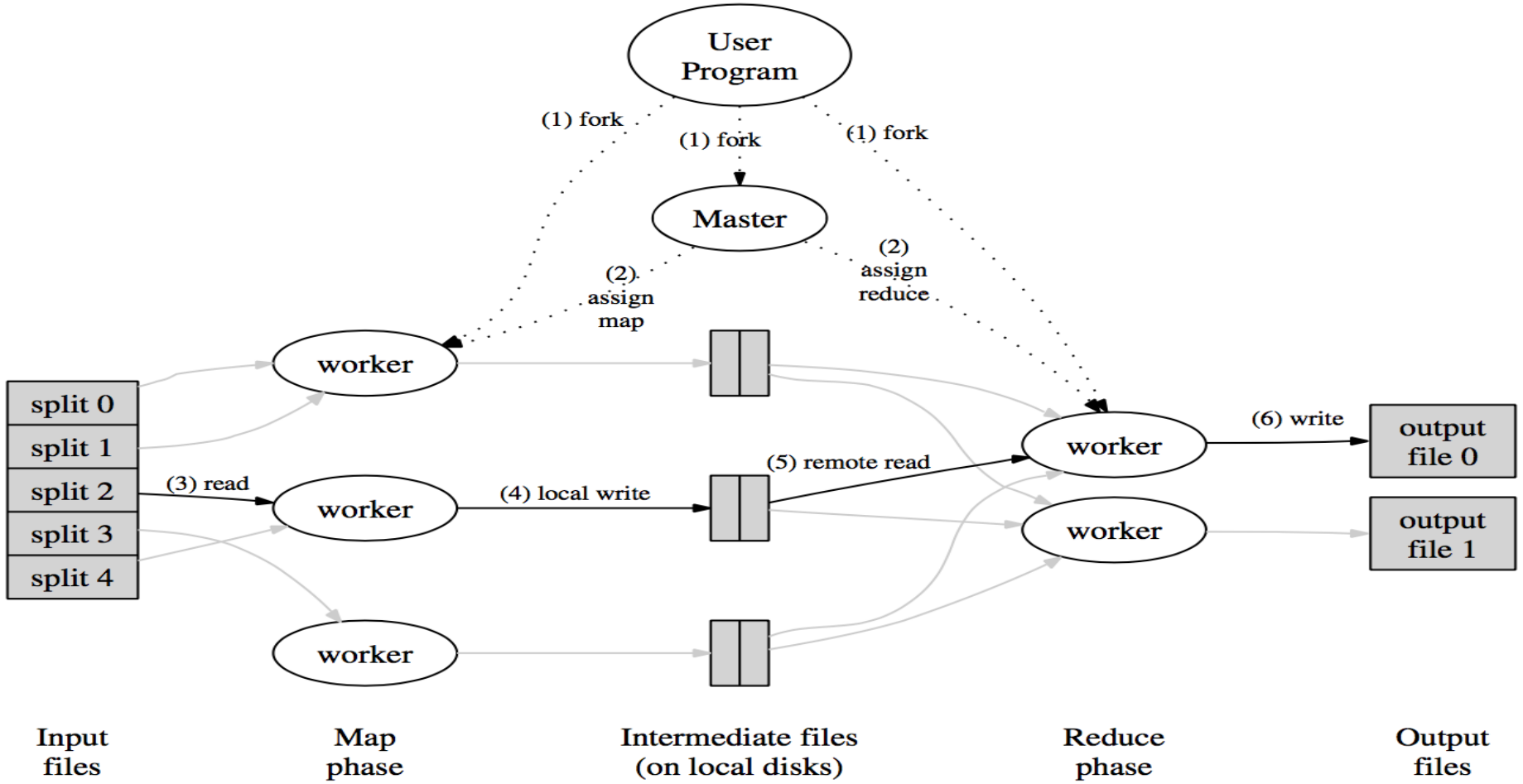
[*] J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, in OSDI'04

Ambiente di lavoro

L'ambiente Google:

- Le macchine hanno processori dual-processor x86 che girano con Linux, aventi 2-4 GB di memoria.
- L'hardware di rete è da 100 megabits/secondo o da 1 gigabit/secondo a livello macchina.
- I clusters sono composti da 100 o 1000 computers.
- Il salvataggio è effettuato con un semplice IDE disk, attaccato direttamente alla macchina.

Implementazione (2)



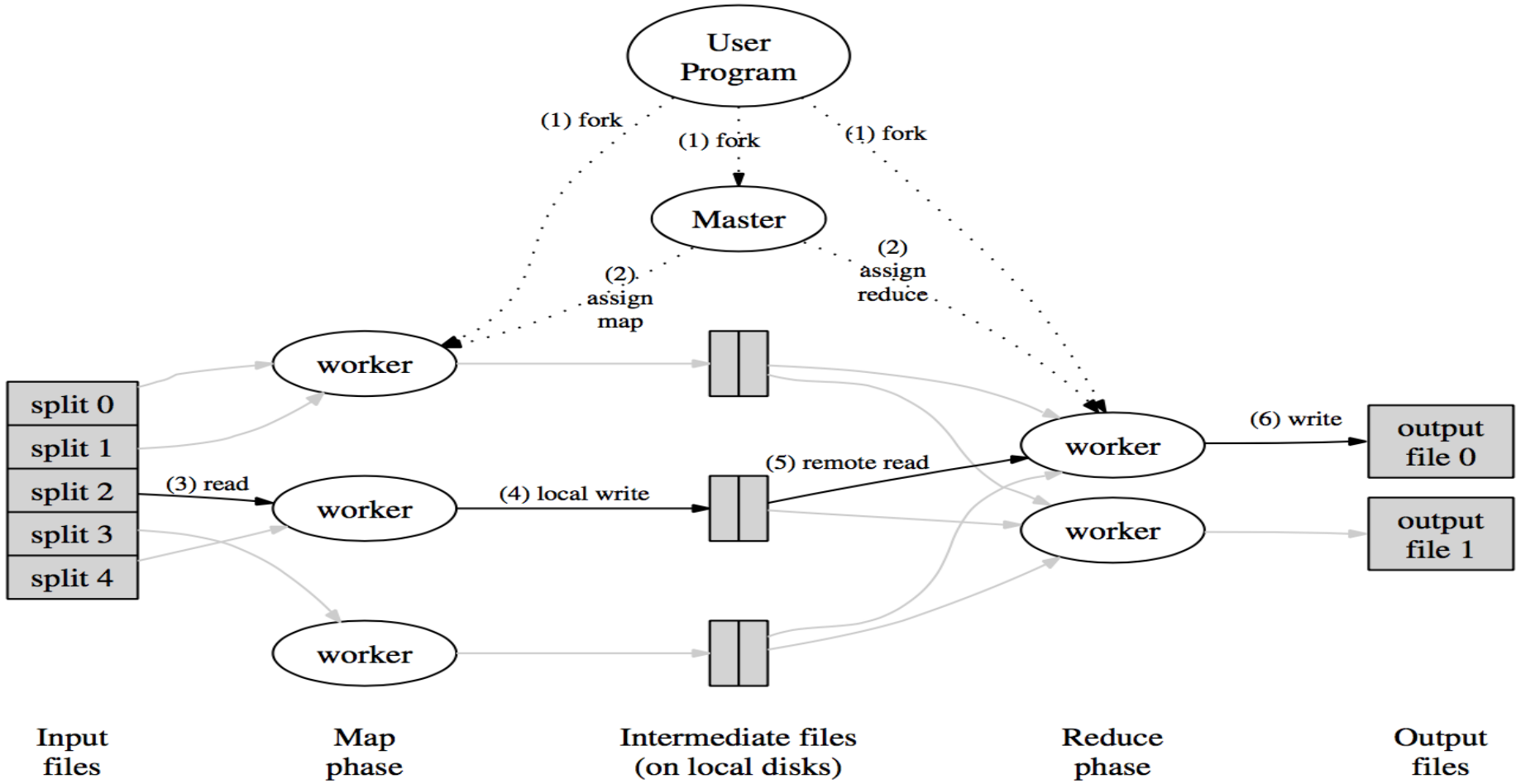
Implementazione (3)

Quando il programma dell'utente chiama la funzione MapReduce() la sequenza di azioni svolta è:

1) La libreria MapReduce divide l'input in M parti (tipicamente da 16 a 64 MB). Poi fa una copia del programma per ogni computer del cluster.

2) Una delle copie del programma è speciale e viene chiamato master, il resto viene chiamato worker. Il master fa da gestore e assegna ad ogni worker inattivo un compito di map o reduce.

Implementazione (3)

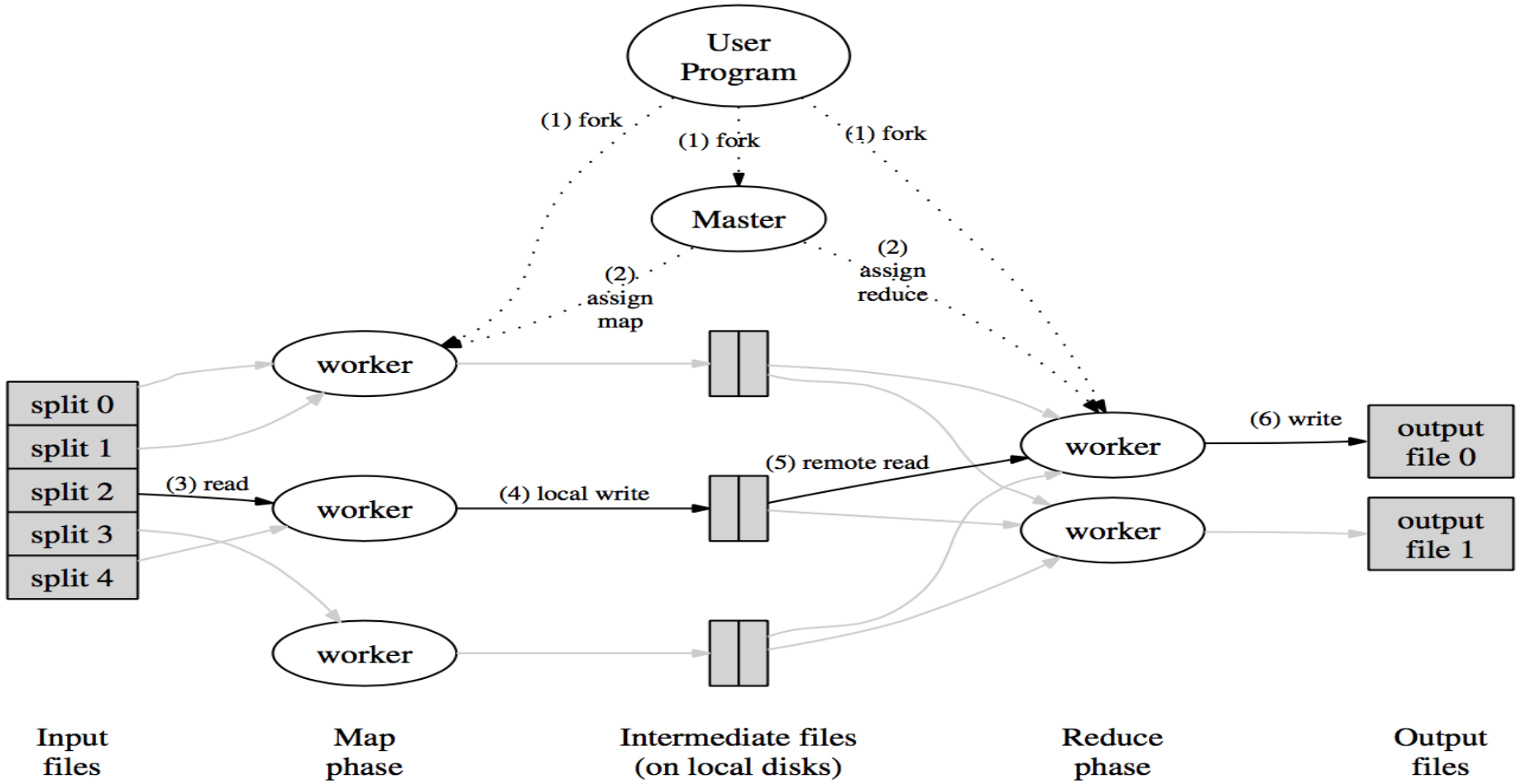


Implementazione (4)

3) Ognuno dei worker di map (mapper) legge il contenuto della corrispondente sotto-parte di input. Estrapola le coppie chiave/valore dall'input e su ognuna di esse si fa girare la funzione map scritta dall'utente. I valori intermedi prodotti sono poi salvati nella memoria del worker.

4) Periodicamente questi valori intermedi vengono scritti in memoria locale, partizionati in una delle R parti. Le locazioni su cui sono stati salvati questi dati vengono comunicate al master, che sarà responsabile di inoltrare tali informazioni ai workers di reduce (reducer).

Implementazione (4)

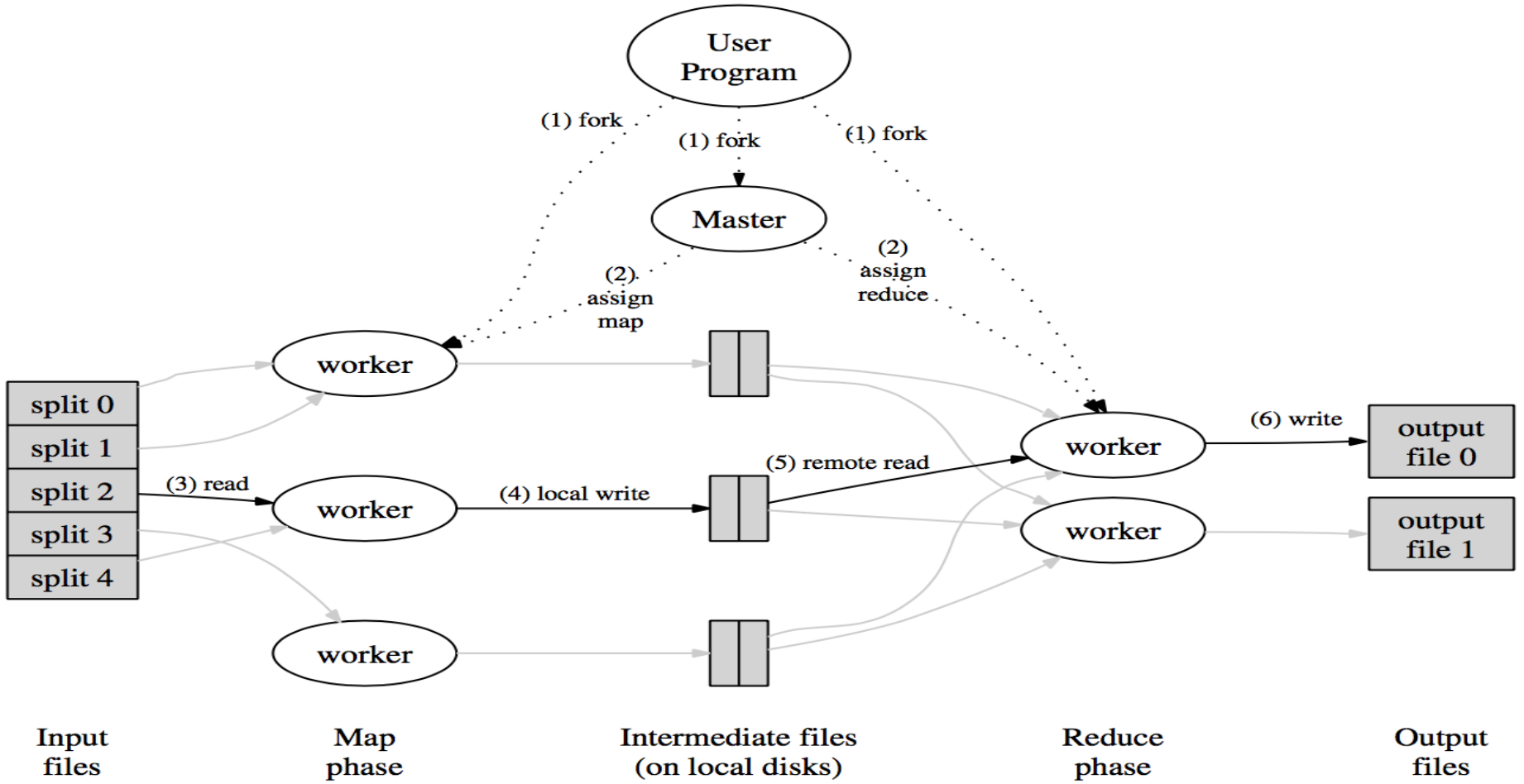


Implementazione (5)

5) Quando ad un reducer viene notificata la locazione dei valori intermedi dal master, esso andrà a leggere le informazioni scritte nelle celle di memori del disco del mapper tramite chiamate di procedure remote. Quando finisce di leggere tutti i valori intermedi affidatigli, ordina questi per chiave intermedia e li raggruppa insieme.

6) Il reducer organizza tutte le coppie intermedie per chiave e per tutte le coppie con stessa chiave forma un input per la funzione di reduce scritta dall'utente e il risultato viene salvato in una parte dedicata del file di output.

Implementazione (5)



Fallimenti

Poiché la libreria di MapReduce è nata con lo scopo di agevolare lo sviluppo di processi su grandi quantità di dati usando centinai o migliaia di computer, deve essere in grado di tollerare il fallimento di una o più macchine.

Il fallimento può provenire o dalla caduta di un worker o dalla caduta del master.

Fallimenti - Worker

Periodicamente il master esegue una funzione di ping su ogni worker e se entro un lasso di tempo questo non invia la risposta, il master considera il worker fallito.

Se il worker sta eseguendo un task, questo viene considerato non eseguito e viene riassegnato ad un altro worker inattivo.

Lo stesso avviene se il worker fallisce dopo aver eseguito il task map, poiché i valori intermedi sono memorizzati sul proprio disco, quindi dopo il suo fallimento tale risultato è da considerarsi irraggiungibile.

Fallimenti - Master

Non è stata prevista nessuna gestione del master. Per svolgere i propri compiti il master 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 disponibili per nessun altro worker, quindi non può essere eletto un nuovo master.

Tuttavia è facilmente implementabile un meccanismo che preveda un salvataggio periodico della memoria del master (checkpoint).

Funzione di back-up

La computazione può essere rallentata dal fatto che una delle macchine impieghi un tempo eccessivo per completare il proprio task.

E’ stato implementato un meccanismo dedicato per 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.

Funzione di partizione

Una configurazione consigliata è M = 200000 e R = 5000 con 2000 worker.

Solitamente M è un numero molto più grande dei worker. R, invece, è un piccolo multiplo dei worker.

D'altra parte le quantità di M e R non possono essere eccessivamente grandi. Il fatto che il master debba tenere informazioni su ogni task crea un bound pratico. Per un *O(M+R)* decisioni corrispondono un *O(M*R)* stati.

Commenti al framework

Complessivamente ben accolto.

Due diverse tipologie di critiche:

1) Critiche avanzate da esperti di architetture e database paralleli → MapReduce viene visto come un sistema per il data warehousing poco strutturato e per niente innovativo.

Commenti al framework

2) Critiche al sistema di computazione parallela/distribuita →
   - 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.
   - 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.

MapReduce (2)

Simulazione in P-RAM

Algoritmi Avanzati
Anno Accademico 2015-2016

Lorenzo Spignoli
Francesco Andreozzi
Viviana Arrigoni

prof.ssa Rossella Petreschi

Classi di Algoritmi in MapReduce (MRC)

Dati $\epsilon, i > 0$, un algoritmo $\mathcal{MRC}^i$ è una sequenza di passi Map e Reduce $< \mu_1, \rho_1, \ldots, \mu_R, \rho_R >$ in cui l'output è corretto con probabilità almeno ¾ e dove:

1) Ogni μ_r usa spazio $O(n^{1-\epsilon})$ e lavora in tempo polinomiale in n.
2) Ogni ρ_r usa spazio $O(n^{1-\epsilon})$ e lavora in tempo polinomiale in n.
3) Il numero di macchine è sublineare rispetto a n.
4) Lo spazio totale $\sum_{<k,v>}(|k| + |v|)$ utilizzato dalle coppie $< key, value >$ è pari a $O(n^{2-2\epsilon})$.
5) Il numero totale dei round R è $O(\log^i n)$

Classi di Algoritmi in MapReduce ($DMRC$)

Nel caso in cui l'algoritmo restituisca l'output corretto deterministicamente , parleremo della classe $\mathcal{DMRC}^i$

Mostriamo come algoritmi della classe $\mathcal{DMRC}^i$ possano simulare algoritmi *PRAM CREW* (lettura concorrente e scrittura esclusiva) e *CRCW* (lettura e scrittura concorrenti)

Simulazione algoritmi *PRAM CREW*

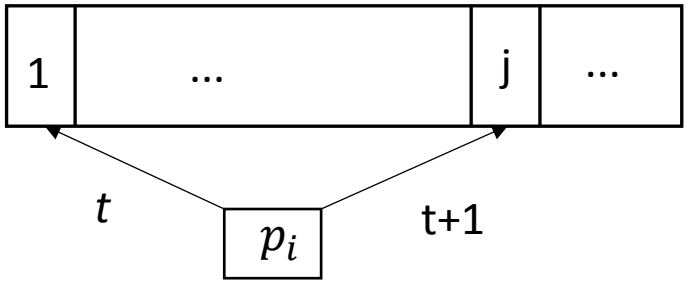
Teorema 1: Un algoritmo *PRAM CREW* che utilizza $O(n^{2-2\epsilon})$ memoria totale e $O(n^{2-2\epsilon})$ processori in un tempo *t=t(n)* può essere simulato da un algoritmo MapReduce in *O(t)* rounds.

Dimostrazione: Utilizzeremo $O(n^{2-2\epsilon})$ reducer per simulare le locazioni di memoria, ed altri $O(n^{2-2\epsilon})$ reducer per simulare i processori.

Dimostriamo che ogni passo t dell'algoritmo *PRAM* può essere simulato in un numero costante di round MapReduce.

Simulazione algoritmi *PRAM CREW (2)*

- Passo *t* dell'algoritmo
  - b_i^t, w_i^t : coppie <*indirizzo a, valore v*>
- Reduce ρ_1:
  - Input <*i;* b_i^t>
  - Output <*i;* r_i^{t+1}, w_i^t>
- Map μ_1 :
  - Input <*i;* r_i^{t+1}, w_i^t>
  - Output <r_i^{t+1}; i> , <a_j; w_i^t>



Il processore p_i (simulato dal reducer ρ_1) legge/scrive la locazione 1 al passo t dell'algoritmo PRAM e richiede di leggere/scrivere la locazione j al passo t+1

Il mapper μ_1 prenderà in input tuple in cui la chiave rappresenta il processore, e i valori rappresentano le locazioni che il processore richiede

Simulazione algoritmi *PRAM CREW (3)*

- Reduce ρ_2:
  - Input <a_j; (a_j, v_j), *i*> , <a_k; *i*>
  - Output <a_k; (a_k, v_k) , *i*>
- Map μ_2 :
  - Input <a_k; (a_k, v_k) , *i*>
  - Output <*i*; a_k, v_k>

- *Il reducer ρ_2 prende in input tuple di due tipi, la cui chiave rappresenta una locazione di memoria*
- *Il primo tipo simula la scrittura del valore v_j da parte del processore i nella locazione a_j*
- *Il secondo tipo simula la richiesta di lettura della locazione a_k da parte del processore i*

- *Il mapper μ_2 infine ottiene la locazione di memoria aggiornata dal reducer ρ_2 ed emette in output tuple per simulare la lettura al passo successivo della locazione di memoria a_k da parte del processore i*

Simulazione algoritmi *PRAM CREW (4)*

- La simulazione è in $\mathcal{DMRC}^i$?
  - Ogni μ_r usa spazio $O(n^{1-\epsilon})$ e lavora in tempo polinomiale in n.
  - Ogni ρ_r usa spazio $O(n^{1-\epsilon})$ e lavora in tempo polinomiale in n
  - Lo spazio totale utilizzato dalle coppie $< key, value >$ è pari a $O(n^{2-2\epsilon})$.
  - Il numero totale dei round R è $O(\log^i n)$.

Simulazione algoritmi *PRAM CRCW*

Teorema 2: Un algoritmo *PRAM CRCW* che utilizza $O(n^{2-2\epsilon})$ memoria totale e *P*=$O(n^{2-2\epsilon})$ processori in un tempo *t=t(n)* può essere simulato da un algoritmo MapReduce in *O(t* $\log_M P$*)* rounds, dove *M* è la memoria dei reducer

In uno stesso passo dell'algoritmo, tutti i processori possono scrivere nella stessa locazione di memoria

Il numero di processori può superare in modo significativo la memoria *M* di ogni reducer.

Simulazione algoritmi *PRAM CRCW (2)*

Nel caso di PRAM con scrittura concorrente, il numero di richieste di scrittura può superare la memoria dei reducer che simulano le locazioni di memoria

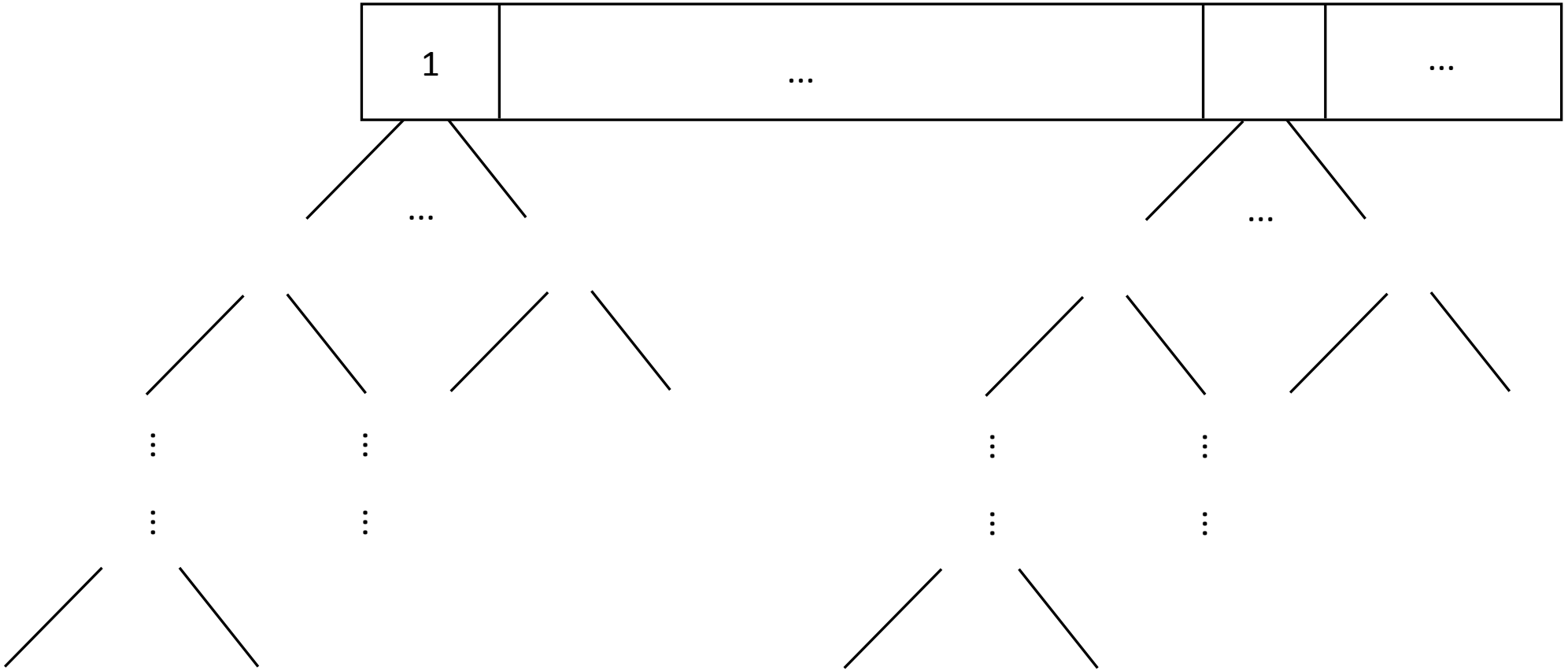
($O(n^{2-2\epsilon}) > O(n^{1-1\epsilon})$)

Utilizziamo la tecnica degli *invisible funnel tree (IFT)*: Un IFT è un albero con radice in ogni locazione di memoria

Non manteniamo l'albero esplicitamente in memoria, poiché questo comporterebbe un aumento della memoria per le coppie <*key,value*>, bensì lo simuliamo attraverso round aggiuntivi di map e reduce

Simulazione algoritmi *PRAM CRCW (3)*

L'IFT avrà fattore di diramazione *M,* dove *M* è la dimensione della memoria di un singolo reducer.



Simulazione algoritmi *PRAM CRCW (4)*

(1) Lettura a due fasi:

- Bottom up : I processori (rappresentati dalle foglie dell'IFT) inviano le richieste di lettura per la locazione in cui l'IFT è radicato. Una sequenza di round *map e reduce* (come nel caso CREW) simula la lettura concorrente da parte dei processori.

- Top down : Il reducer associato ad una locazione riceve le richieste di lettura e le inoltra ai processori che hanno inviato le richieste.

(2)Computazione interna dei processori

(3) Scrittura

- Bottom up : I processori che intendono scrivere su una locazione di memoria, inviano tale richiesta (come nel caso CREW) ma all'interno dell'albero vengono anche eseguiti passi intermedi di Combine. In questo modo, il reducer che simula una locazione riceverà al più *M* valori, non eccedendo la sua memoria interna.

Simulazione algoritmi *PRAM CRCW (4)*

Il numero di operazioni per ogni round di simulazione dell IFT è polinomiale, quindi la complessità rimane invariata.

Lo spazio totale $\sum_{<k,v>}(|k| + |v|)$ delle coppie <*chiave,valore*> rimane invariato.

Il numero di round per la simulazione degli IFT è *O(altezza IFT) = O(*$\log_M P$)

Il numero di mapper e reducer rimane invariato

Simulazione algoritmi *PRAM CRCW (5)*

L'algoritmo è in $\mathcal{DMRC}^i$, quindi il Teorema 2 è dimostrato.

MapReduce (3)

Modello Computazionale e Algoritmi

Algoritmi Avanzati
Anno Accademico 2015-2016

Lorenzo Spignoli
Francesco Andreozzi
Viviana Arrigoni

prof.ssa Rossella Petreschi

Requisiti per un algoritmo in MapReduce

Affinché l'esecuzione di un algoritmo sia efficiente, bisogna garantire i seguenti principî:

Memoria

Ogni mapper e ogni reducer deve riceve un input di dimensione sublineare rispetto alla dimensione dell'intero input.

Macchine

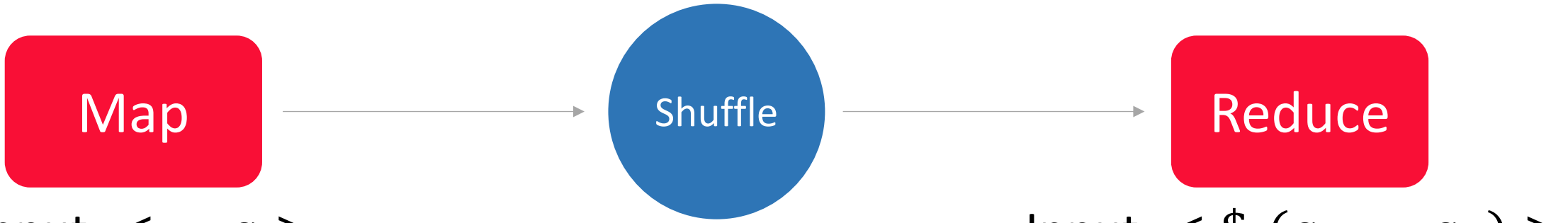
Il numero di macchine deve essere sublineare rispetto alla dimensione del dato iniziale.

Tempo

Ogni mapper e ogni reducer deve lavorare in tempo polinomiale.

Somme di *n* scalari (algoritmo inefficiente)

Task: Dati $< a_1, \dots, a_n >$ calcolare $S = a_1 + \cdots + a_n$



Problemi:

- Non c'è parallelizzazione
- Il Reduce usa spazio lineare

Somme di *n* scalari (algoritmo efficiente)

Task: Dati $< a_1, \dots, a_n >$ calcolare $S = a_1 + \cdots + a_n$

Round 1

Map

Input: $\sqrt{n}$ coppie $< -, a_i >$

Emits: $< \$_{i/\sqrt{n}}, a_i >$

Shuffle

$< \$_{1/\sqrt{(n)}}, \left(a_{\sigma(1)}, \dots, a_{\sigma(\sqrt{n})}\right) >$

…

$< \$_{\sqrt{n}/\sqrt{n}}, \left(a_{\sigma(1)}, \dots, a_{\sigma(\sqrt{n})}\right) >$

Reduce

Input: $< \$_{i/\sqrt{n}}, (a_{\sigma(1)}, \dots, a_{\sigma(\sqrt{n})}) >$

Emits: $< \$_j, S_j = a_{\sigma(1)} + \cdots + a_{\sigma(\sqrt{n})} >$

Round 2

Map

Input: $< \$_j, S_j >$

Emits: $< \$, S_j >$

Shuffle

$< \$, \left(S_{\sigma(1)}, \dots, S_{\sigma(\sqrt{n})}\right) >$

Reduce

Input: $< \$, \left(S_{\sigma(1)}, \dots, S_{\sigma(\sqrt{n})}\right) >$

Emits: $S = S_{\sigma(1)} + \cdots + S_{\sigma(\sqrt{n})}$

MST in MapReduce

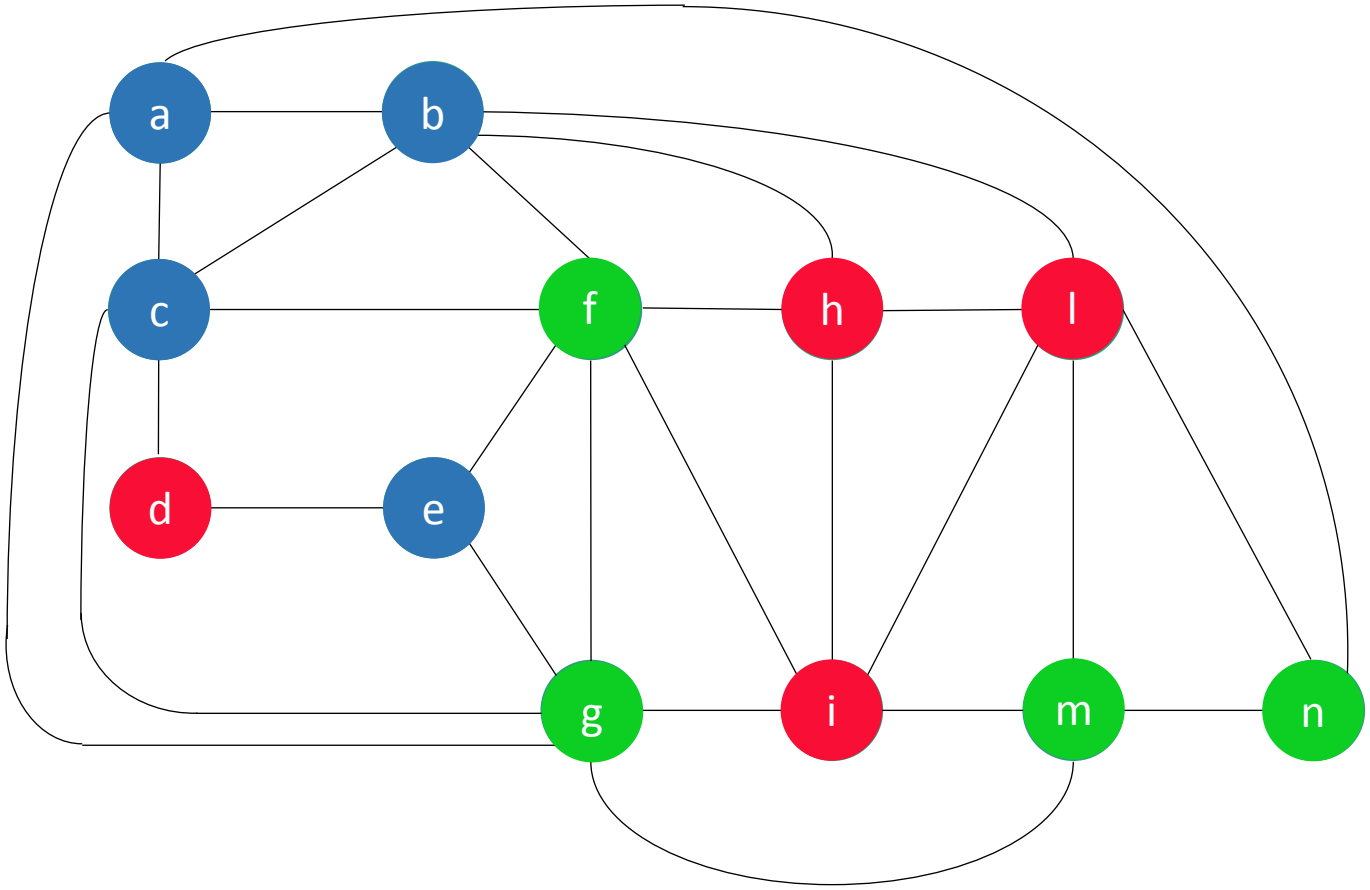
Task: Dato un Grafo connesso $G = (V, E)$, trovare il suo MST

- $|V| = N, |E| = m$.
- Fissato $k \in \mathbb{N}$, sia $\{V\}_{j=1,\dots,k}$ una partizione di V tale che $|V_1| = \cdots = |V_k| = {}^{N}/_{k}$.
- $\forall (i,j), i,j \in \{1,\dots,k\}, i \neq j$, sia $E_{i,j} = \{(u,v) : u,v \in V_i \cup V_j\} \subseteq E$.
- $\forall (i,j), i,j \in \{1,\dots,k\}, i \neq j$, definiamo quindi il grafo indotto $G_{i,j} = \left(V_i \cup V_j, E_{i,j}\right)$.

Osservazione: Il generico $G_{i,j}$ potrebbe non essere connesso.

Esempio

$G = (V, E)$



$N = 12, m = 26$, scegliamo $k = 3$

$$\Rightarrow V = V_1 \cup V_2 \cup V_3, |V_i| = 4$$

Prendiamo ad esempio:

$V_1 = \{a, b, c, e\}$

$V_2 = \{d, h, i, l\}$

$V_3 = \{f, g, m, n\}$

Osserviamo che è stato scelto G denso, ovvero

$$\exists c > 0 \; t.c. \; m \geq N^{1+c}$$

In questo caso infatti:

$$26 \geq 12^{1+c}$$

Essendo:

$$26 \sim 12^{1.3112}$$

MST: idea dell'algoritmo

Per ognuno dei $\binom{k}{2}$ $G_{i,j}$, si calcola l'unico MST, $M_{i,j}$.

- G è considerato, per semplicità, un grafo non pesato. Per garantire l'unicità del MST, nel momento in cui bisogna visitare un nodo u, si sceglie come arco (v, u), dove v è il nodo che ha ordine alfabetico minore rispetto a tutti gli altri nodi vicini a u.
- Poiché il generico $G_{i,j}$ potrebbe non essere connesso, piuttosto che di *minimum spanning tree* sarebbe più corretto parlare di *minimum spanning forest.*

Si definisce $H = \left(V, \bigcup_{i,j} E(M_{i,j})\right)$.

Si calcola il MST di H.

H ha dimensione molto inferiore a *G*, specie se questo è un grafo denso!

Algoritmo per il *MST* (Round 1)

Map1: Input: k, lista di archi (u, v) di G.

Definisci una qualsiasi partizione di V, di N/k vertici ciascuna.

Repeat for all edges $(u, v), u \in V_i, v \in V_j$

- If $i \neq j$ emit $< G_{i,j}, (u, v) >$
- Else emit $< G_{i,h}, (u, v) > \forall h \neq i$

Shuffle:

$\forall i, j$ genera $< G_{i,j}$, lista di archi (u, v) $t.c. u, v \in V_i \cup V_j >$

Reduce1: Input: $< G_{i,j}$, lista di archi $(u, v) >$

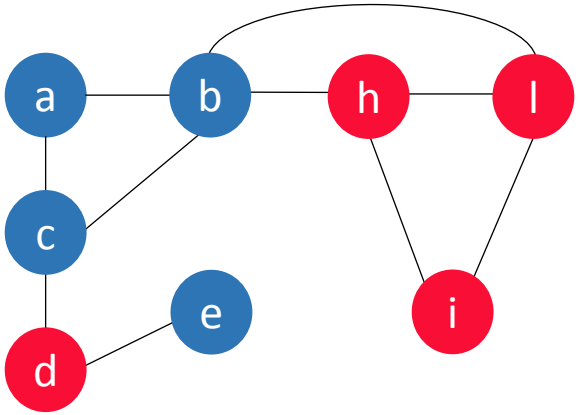
Calcola $M_{i,j}$ tramite un algoritmo qualsiasi (Kruskal, Prim...)

Emit $< G_{i,j}$, lista degli archi di $M_{\mathrm{i,j}} >$

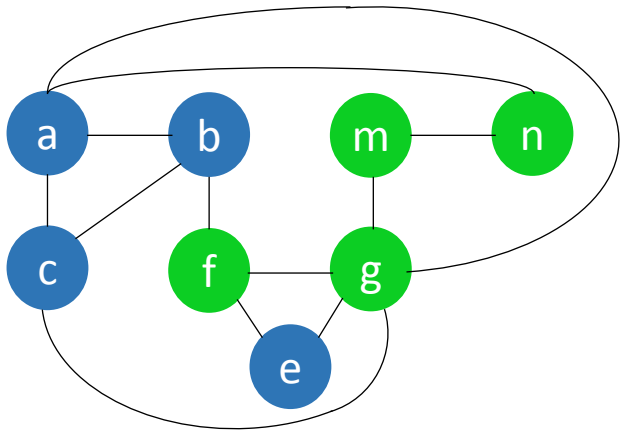
Esempio

Map1:

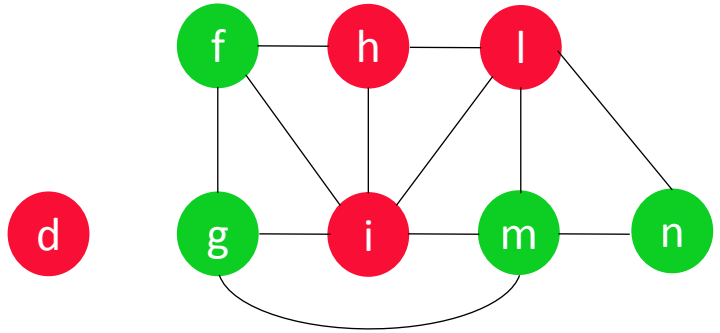
$G_{1,2}$



$G_{1,3}$

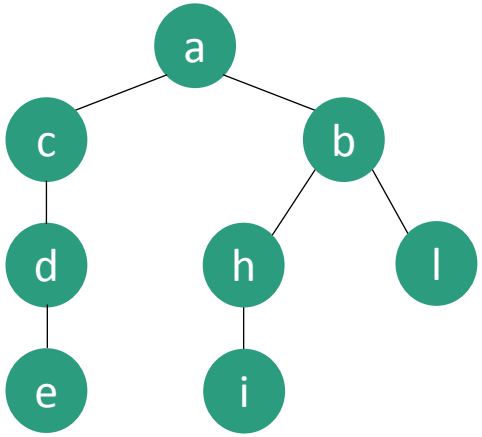


$G_{2,3}$

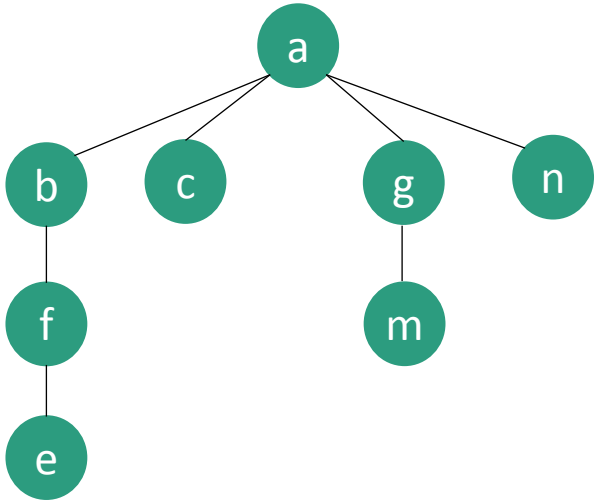


Reduce1:

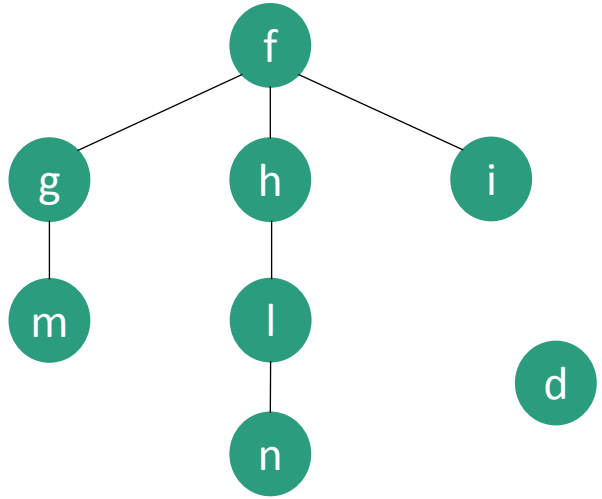
$M_{1,2}$



$M_{1,3}$



$M_{2,3}$



Algoritmo per il *MST* (Round 2)

Map2: Input: $< G_{i,j}$, lista degli archi di $M_{\mathrm{i,j}} >$

Crea il grafo $H = (V_H, E_H) = \left(V, \cup_{i,j} M_{i,j}\right)$.

For ogni arco e in H

- Emit (\$, e)

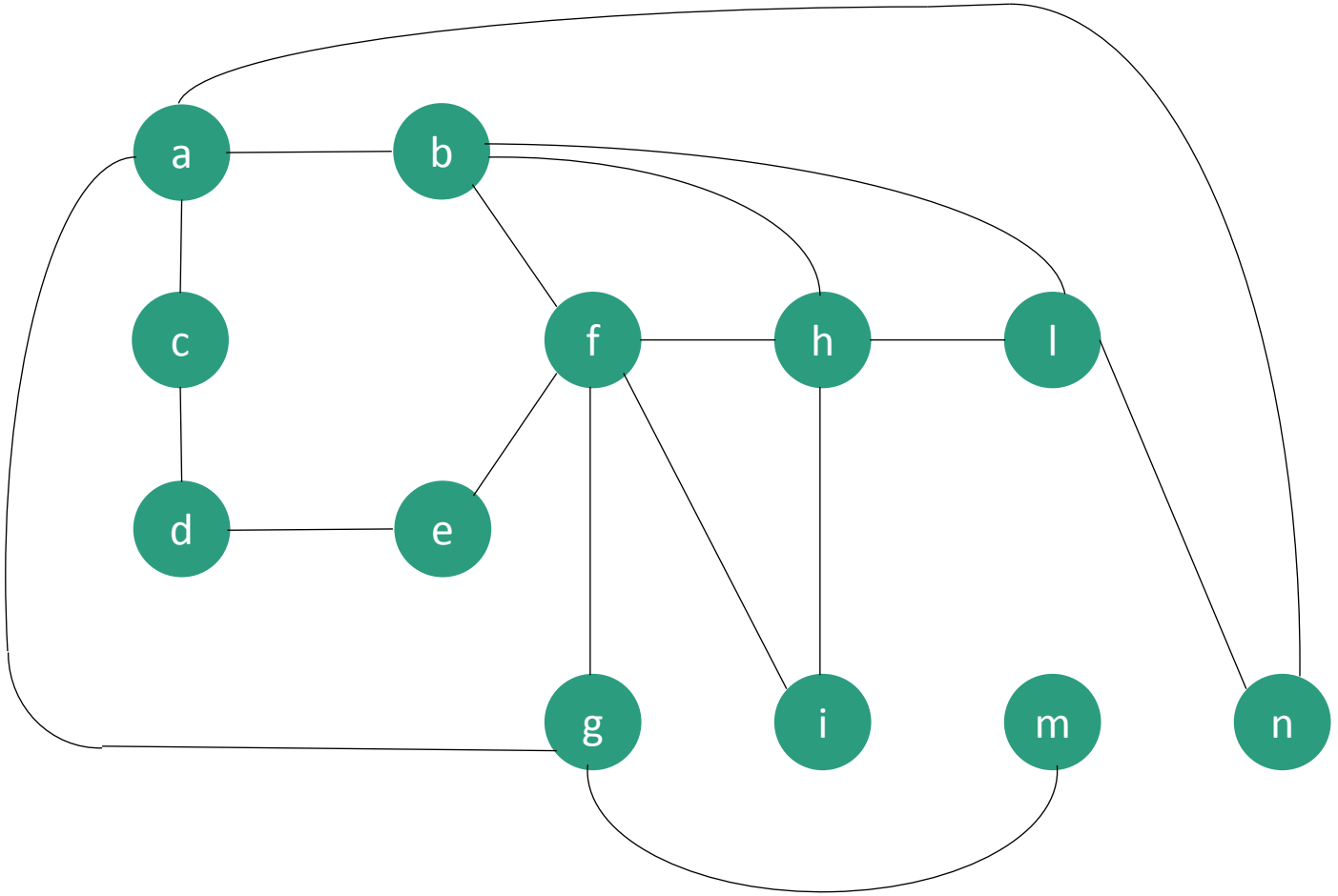
Reduce2: Input: lista degli archi di H

Calcola MST di H (Kruskal, Prim…)

Emit lista di archi del MST di H

Esempio

Map2:

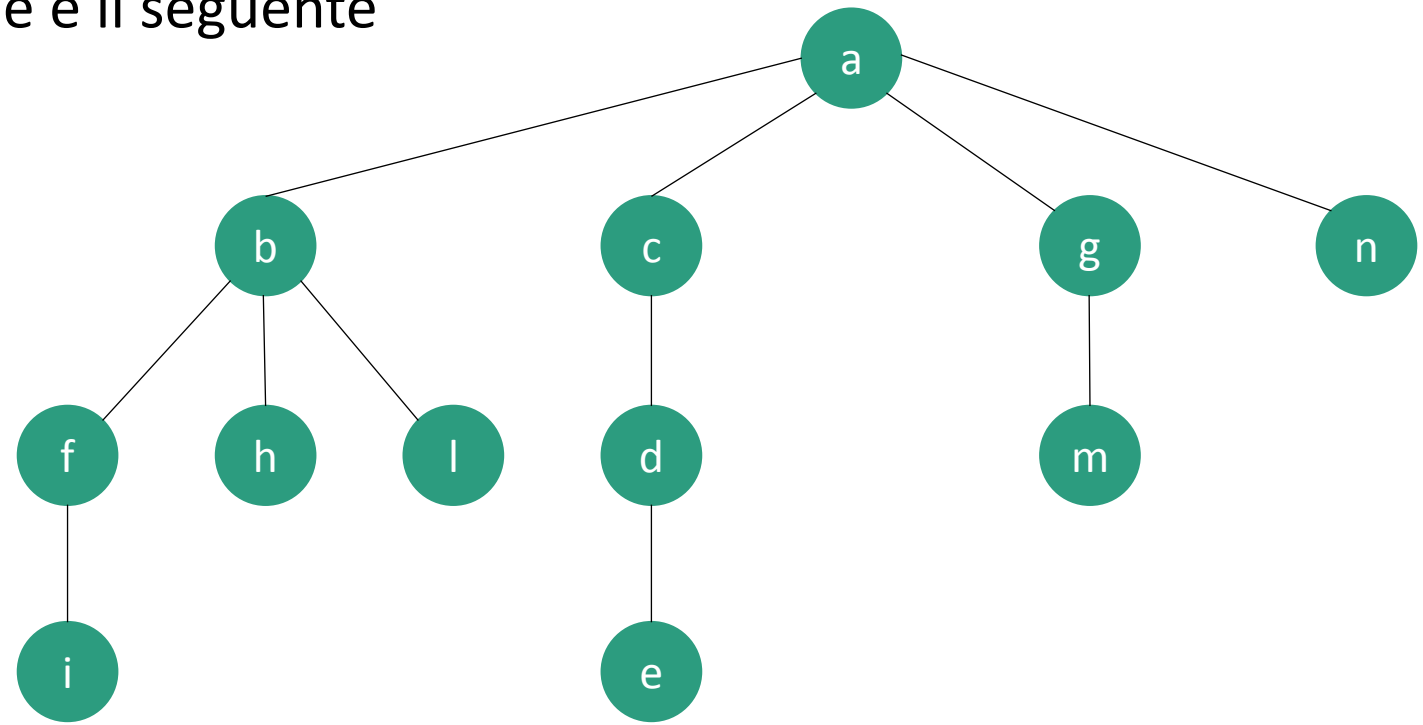


$$H = (V_H, E_H)$$
$$|E_H| = 17$$

Esempio

Reduce2:

Il MST finale è il seguente



Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (1)

Per dimostrare che l'algoritmo è $\mathcal{MRC}^i$ bisogna mostrare i seguenti fatti:

1. L'algoritmo è una sequenza di passi Map e Reduce.

Lo è per costruzione. ✓

2. L'output è corretto.

Vogliamo mostrare che MST(G)=MST(H).

$\nexists(u,v)\ t.c.\ (u,v) \in$MST($G$), $(u,v) \notin$ MST(H).

Sia $(u,v) \in G, (u,v) \notin H$. Allora $(u,v) \notin$MST (H). Ma $\exists\ (i,j)\ t.c.\ (u,v) \in G_{i,j}$.

Tuttavia $(u,v) \notin H$, e questo vuol dire che l'arco è stato scartato durante la costruzione del MST di $G_{i,j}$, ovvero quando si calcola $M_{i,j}$. Ma quindi (u,v) induce un ciclo in $G_{i,j}$, ed è l'arco del ciclo con 'peso' maggiore, e dunque era giusto scartarlo.

$\nexists(u,v)\ t.c.\ (u,v) \in$MST($H$), $(u,v) \notin$ MST(G).
H è un sottografo di G. ✓

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (2)

3. $\exists\ i\ t.c.$ il numero di round è $O(log^i\ m)$.

Il numero di round è 2, quindi costante, e non dipende dalla dimensione dell'input, il numero di round è $O(\log^0 m) = O(1)$.✓

4. Il tempo di ogni map e di ogni reduce è polinomiale.

Map1: Crea partizione, definisce i grafi indotti $G_{i,j} \longrightarrow O(N + m)$

Reduce1: Applica un algoritmo per la costruzione del MST di ogni $G_{i,j} \longrightarrow O(mlogN)$ [Prim], $O(mN)$ [Kruskal].

Map2: Crea il grafo H elencando tutti gli archi di ogni $G_{i,j} \longrightarrow O(m)$.

Reduce2: Applica un algoritmo per la costruzione del MST di H $\longrightarrow O(mlogN)$ [Prim], $O(mN)$ [Kruskal]. ✓

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (2)

5. Lo spazio totale utilizzato è (molto probabilmente) $O(n^{2-2\epsilon})$.

Round 1

Bisogna innanzitutto stimare quanti archi ci sono in ogni $G_{i,j}$, ovvero quanto vale $\mathbb{E}(|E_{i,j}|)$.

Osserviamo che

$$|E_{i,j}| \leq \sum_{v \in V_i} \deg(v) + \sum_{v \in V_j} \deg(v) \qquad (1)$$

Partizioniamo i vertici in gruppi in base al loro grado:

$$W_t = \{v \in V \; t.c. \; 2^{t-1} \leq \deg(v) < 2^t\}$$

Dove $t = 1, \dots, logN$, infatti il grado massimo è $N - 1 = 2^{logN} - 1$.

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (3)

Affermazione1:

$$v \in W_t \Rightarrow \deg(v) = \Theta(2^t)$$

Deriva dalla definizione di W_t. ■

Affermazione2:

$$\Theta\left(\sum_{t=1}^{logN} 2^t |W_t|\right) = \Theta(m)$$

$2m = \sum_{v \in V} \deg(v) = \sum_{t=1}^{logN} \sum_{v \in W_t} \deg(v) =$
$\sum_{t=1}^{logN} \sum_{v \in W_t} \Theta(2^t) = \Theta(\sum_{t=1}^{logN} \sum_{v \in W_t} 2^t) = \Theta(\sum_{t=1}^{logN} |W_t| 2^t)$. ■

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (4)

Ora ci chiediamo quanti dei nodi di W_t vengono mappati in V_i per un qualche i, Ovvero vogliamo studiare quanto vale $\mathbb{E}[|W_t \cap V_i|]$. Sia $v \in W_t$, e sia $X_{v,1}$ una variabile $t.c.$

$$X_{v,i} = \begin{cases} 1 & se\ v \in V_i \\ 0 & altrimenti \end{cases}$$

Supponendo che la distribuzione di probabilità su ogni V_i sia quella uniforme, si ha che:

$$P(X_{v,i} = 1) = \frac{1}{k}$$

E quindi:

$$\mathbb{E}[X_{v,i}] = 1 \cdot \frac{1}{k} + 0 \cdot \left(1 - \frac{1}{k}\right) = \frac{1}{k}$$

Da cui:

$$\mathbb{E}[|W_t \cap V_i|] = \mathbb{E}\left[\sum_{v \in W_t} X_{v,i}\right] = \sum_{v \in W_t} \mathbb{E}[X_{v,i}] = \sum_{v \in W_t} \frac{1}{k} = \frac{|W_t|}{k} \qquad (2)$$

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (5)

Affermazione3:

$$\mathbb{E}\left[\sum_{v \in V_i} \deg(v)\right] = \Theta\left(\frac{m}{k}\right)$$

Infatti: $\mathbb{E}\left[\sum_{v \in V_i} \deg(v)\right] = \mathbb{E}\left[\sum_{t=1}^{logN} \sum_{v \in W_t \cap V_i} \deg(v)\right] \overset{\text{Aff1}}{=} \mathbb{E}\left[\sum_{t=1}^{logN} \Theta(2^t)|W_t \cap V_i|\right] \overset{(2)}{=}$

$= \Theta\left(\sum_{t=1}^{logN} 2^t \frac{|W_t|}{k}\right) = \frac{1}{k}\Theta\left(\sum_{t=1}^{logN} 2^t |W_t|\right) \overset{\text{Aff2}}{=} \Theta\left(\frac{m}{k}\right)$. ■

Spazio per il primo round

$$\Rightarrow |E_{i,j}| \leq \sum_{v \in V_i} \deg(v) + \sum_{v \in V_j} \deg(v) \Rightarrow \mathbb{E}[|E_{i,j}|] = O\left(\frac{m}{k}\right)$$

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (6)

Round 2

Ricordiamo che $H = \left(V, \cup_{1 \leq i \neq j \leq k} M_{i,j}\right)$ e che $|V_i| = \frac{N}{k}, \forall\, i$, allora:

$$\mathbb{E}[|H|] = \mathbb{E}\left[\sum_{i,j} |M_{i,j}|\right] \leq \mathbb{E}\left[\sum_{i,j} |V_i| + |V_j|\right] = \mathbb{E}\left[2\binom{k}{2}\frac{N}{k}\right] \approx \mathbb{E}[kN] = \boxed{kN}$$

Spazio per il secondo round

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (7)

Abbiamo che lo spazio utilizzato per il primo e per il secondo round del Reduce è rispettivamente

$$O\left(\frac{m}{k}\right) \qquad O(kN)$$

Bisogna quindi bilanciare le due quantità:

$$\frac{m}{k} = Nk \Leftrightarrow k^2 = \frac{m}{N} \Leftrightarrow k = \sqrt{\frac{m}{N}}$$

Otteniamo quindi che lo spazio utilizzato per ambo i round del Reduce è:

$$Nk = \frac{m}{k} = N\sqrt{\frac{m}{N}} = \sqrt{mN}$$

Teorema: l'Algoritmo è $\mathcal{MRC}^i$ (8)

È necessario distinguere due casi:

G è sparso → $m \sim N$ → $\sqrt{mN} \sim m$ → Lo spazio utilizzato è lineare: non c'è abbastanza parallelizzazione!

G è denso → $m \geq N^{1+c}$

Affinché lo spazio sia sublineare, deve essere dell'ordine di $n^{1-\epsilon}, \epsilon > 0, n = m + N \approx m$.
Quindi deve essere:

$$\sqrt{mN} \leq n^{1-\epsilon} \approx m^{1-\epsilon} = \frac{m}{m^{\epsilon}} \leq \frac{m}{N^{\epsilon}} \Leftrightarrow \sqrt{m} \geq N^{\frac{1}{2}+\epsilon} \Leftrightarrow m \geq N^{1+2\epsilon}$$

Lo spazio è sublineare solo quando risulta

$m \geq N^{1+2\epsilon}$ ovvero quando $c = 2\epsilon$ ✓

Bibliografia

Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

Goodrich, Michael T. "Simulating parallel algorithms in the mapreduce framework with applications to parallel computational geometry." arXiv preprint arXiv:1004.4708 (2010).

Karloff, Howard, Siddharth Suri, and Sergei Vassilvitskii. "A model of computation for MapReduce." *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010.