SAPIENZA
UNIVERSITÀ DI ROMA

“Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry”

Michael T. Goodrich

- Facoltà di Ingegneria dell'Informazione, Informatica e Statitica
- Corso di laurea in Informatica
- Dario Frascaria 1244924
- Anno 2011/2012

Introduzione

Il framework 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;
- la gestione della comunicazione tra le macchine e tra i nodi.

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**, e anche la sua versione open source, **Hadoop**, viene largamente utilizzata da grandi compagnie.

La computazione nel framework MapReduce

In questo framework una computazione MapReduce è descritta come una sequenza di tre passi:

- Passo map
- Passo shuffle
- Passo reduce

che operano su un insieme $X = \{x_1, x_2,...,x_n\}$ di valori.

I valori di output da un passo reduce possono essere sia valori finali, cioè inclusi nell'output finale dell'algoritmo, sia valori intermedi, cioè valori utilizzati come input per un altro round di passi map-shuffle-reduce.

La computazione nel framework MapReduce: I 3 passi

- **Un passo map**
  applica una funzione μ ad ogni valore x_i per produrre una coppia chiave-valore (k_i, v_i).
  Per permettere l'esecuzione parallela, il calcolo della funzione $\mu(x_i) \rightarrow (k_i, v_i)$, deve dipendere solo da x_i.
- **Un passo shuffle**
  raccoglie tutte le coppie chiave-valore prodotte nel passo map precedente e produce un insieme di liste $L_k = (k; v_{i1}, v_{i2}, ...)$, dove ogni lista è costituita da tutti i valori v_{ij} tale che $k_{ij} = k$ per una chiave k assegnata in un passo map.
- **Un passo reduce**
  applica una funzione ρ a ciascuna lista, $L_k = (k; v_{i1}, v_{i2}, \ldots)$, formata nel passo Shuffle, per produrre un insieme di valori $y_{j1}, y_{j2},...$
  La funzione ρ può essere definita sequenzialmente su L_k ma dovrebbe essere indipendente dalle altre liste $L_{k'}$ dove $k' \neq k$.

Valutazione degli algoritmi MapReduce

Ci sono diversi parametri che si possono utilizzare per misurare l'efficienza di un algoritmo MapReduce:

- t:
  Il numero di round di map-shuffle-reduce che l'algoritmo utilizza.
- $n_{i,j}$:
  Le dimensioni degli Input e degli Output del mapper/reducer j nel round i.
- M_i:
  La complessità dei messaggi del round i dell'algoritmo, cioè la somma totale delle dimensioni degli input e degli output per tutti i mapper e i reducer nel round i, vale a dire $M_i = \sum_j n_{i,j}$.

  Si può anche definire **la complessità dei messaggi dell'intero algoritmo** come $M = \sum_{i=1}^{t} M_i$.

Valutazione degli algoritmi MapReduce II

- r_i:

  Il tempo di esecuzione interna per il round i, che è il tempo massimo di esecuzione interna utilizzato da un reducer nel round i, dove si assume che $r_i \geq \max_j \{n_{i,j}\}$, dato che un reducer deve avere un tempo di esecuzione che sia almeno uguale alla dimensione dei suoi input e dei suoi output.

  Si può anche definire **il tempo di esecuzione interno dell'intero algoritmo** come $r = \sum_{i=1}^{t} r_i$.

- B:

  La dimensione dei buffer dei reducer, cioè la dimensione massima della memoria di lavoro necessaria ad un reducer per elaborare i suoi input e i suoi output (in aggiunta alla memorizzazione utilizzato per l'input stesso), richiesta in tutti i round t dell'algoritmo.

Valutazione degli algoritmi MapReduce III

- L:

  La latenza della rete del passo Shuffle è il tempo che un reducer deve aspettare fino a quando non riceve il suo primo input in un dato round.

- b:

  La larghezza della banda della rete del passo Shuffle, che è il numero di elementi in una computazione di MapReduce che può essere consegnato dalla rete in ogni unità di tempo.

Tempo di esecuzione degli algoritmi MapReduce

- Dati questi parametri, quindi, il tempo totale T di un'implementazione di un algoritmo MapReduce può essere caratterizzato come segue:

$$T \in O\left(\sum_{i=1}^{t}(r_i + L + M_i/b)\right) = O(r + tL + M/b)$$

dove come abbiamo detto in precedenza:

- r_i è **il tempo di esecuzione interna** per il round i
- L è l**a latenza** della rete del passo Shuffle
- M_i è **la complessità dei messaggi** del round i
- b è **la larghezza della banda** della rete del passo Shuffle
- t è **il numero di round**

Esempio di computazione

Ad esempio, si consideri un algoritmo MapReduce che conta tutte le istanze di parole in un documento.

Dato un documento D si definisce l'insieme dei valori di input X come l'insieme di tutte le n parole nel documento e si procede nel modo seguente:

- **Map**:
  Per ogni parola w nel documento si mappa w in (w, 1).
- **Shuffle**:
  Si raccolgono tutte le coppie (w,1) per ogni parola producendo una lista (w; 1, 1, ..., 1), osservando che il numero di 1 in ogni lista è uguale al numero di volte in cui w appare nel documento.
- **Reduce**:
  Scansiona ogni lista (w; 1, 1, ..., 1) sommando il numero di 1 in ciascuna lista e si restituisce in output una coppia (w, n_w) come valore di output finale, dove n_w è il numero di 1 nella lista di w.

Esempio di computazione II

Tale algoritmo ha nel caso peggiore:

- t =1
- M = O(n)
- r = O(n);

di conseguenza il suo tempo di esecuzione nel caso peggiore è O(n).

Purtroppo tali prestazioni potrebbero essere molto comuni.

Infatti il tempo di esecuzione dell'algoritmo di conteggio è proporzionale al numero di occorrenze della parola più frequente ed alcune parole nel linguaggio naturale tendono ad comparire molto frequentemente.

Valutazione deli algoritmi MapReduce: conclusioni

Da questo dunque si deduce che concentrarsi esclusivamente sul numero di round in un algoritmo MapReduce può effettivamente portare ad algoritmi inefficienti.

Per esempio concentrarsi solo sul numero di round t può portare ad algoritmi che mappano tutti gli input in una singola chiave e quindi il reducer di questa chiave non fa altro che eseguire un algoritmo standard sequenziale per risolvere il problema, non usando nessun parallelismo.

Quindi si avrebbe un algoritmo efficiente come il miglior algoritmo sequenziale esistente.

Algoritmi Memory-Bound MapReduce

In questo articolo si fanno un pò di ipotesi semplificative che consentono agli algoritmi MapReduce di essere più concisi.

In particolare si suppone che si può definire la dimensione del buffer dei reducer (B) in modo tale che la dimensioni degli I/O dei reducer sia sempre O(B).

Così il tempo di esecuzione di MapReduce visto precedentemente

$O(\sum_{i=1}^{t}(r_i + L + M_i/b))$ dove $\mathbf{r_i \geq \max_j \{n_{i,j}\}}$.

diventa : $\mathbf{O(tB + tL + M/b)}$

Da questo deriva il nome **algoritmo memory-bound MapReduce.**

Considerando che L e b sono dipendenti dal cluster di macchine, solo la complessità dei messaggi ed il numero dei round vengono presi in considerazione nell'analisi dell'efficenza degli algoritmi.

Modello BSP

Il Bulk Synchronous Parallel (BSP) è un modello di transizione per la progettazione di algoritmi paralleli sviluppato da Leslie Valiant nel 1980. L'articolo definitivo è stata pubblicato nel 1990.

Un modello di transizione "è inteso né come hardware, né un modello di programmazione, ma una via di mezzo" .

Serve uno scopo simile a quello del modello Parallel Random Access Machine(PRAM) ma differisce da questo non dando per scontato comunicazione e sincronizzazione.

Infatti una parte importante dell'analisi di un algoritmo BSP poggia sulla quantificazione necessaria della sincronizzazione e della comunicazione.

Modello BSP II

Ogni processore ha una memoria locale veloce e può eseguire diversi thread di calcolo.

L'input di dimensione n è assegnato ai p processori (connessi da una rete di comunicazione) in modo tale che ogni processore sia assegnato al massimo ad $m = \lceil n/p \rceil$ elementi di input.

Un calcolo BSP procede in una serie di supepassi, ognuno dei quali prevede che ciascun processore esegua una computazione interna e poi invii un insieme di m messaggi agli altri processori .

Un superpasso si compone di tre fasi ordinate:

- Computazione concorrente;
- Comunicazione;
- Barriera di sincronizzazione;

Modello BSP III

- Computazione concorrente:

  Calcoli diversi si svolgono su ogni processore, il quale utilizza solo I valori memorizzati nella sua memoria locale.

  Le computazioni sono indipendenti.

- Comunicazione:

  In questa fase,i processori inviano messaggi tra loro ma questi non saranno ricevuti prima del prossimo superpasso.

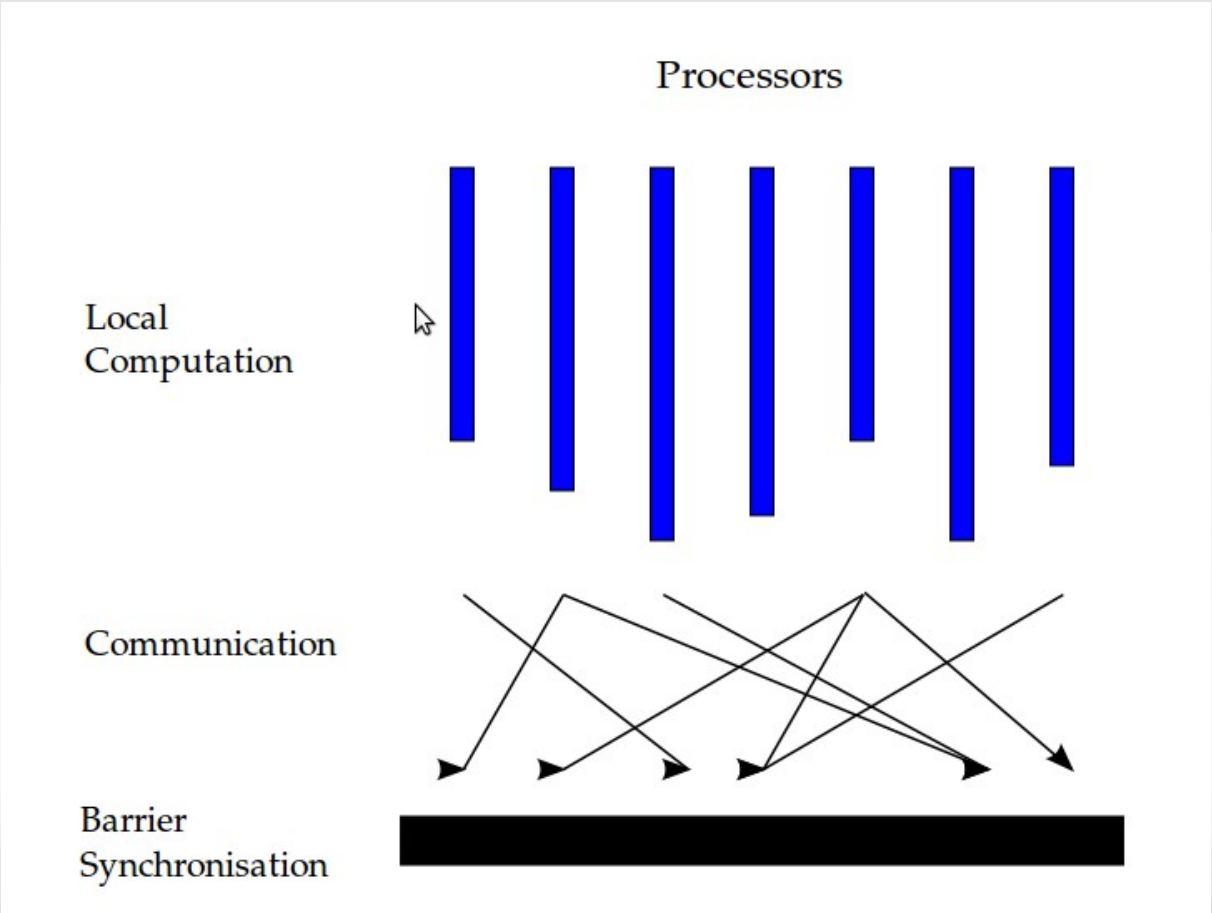
- Barriera di sincronizzazione:

  Quando un processore arriva a questa fase aspetta fino a quando tutti gli altri hanno terminato le loro azioni di comunicazione.

  Quando tutti i processori raggiungono questa fase i messaggi inviati nella fase di comunicazione vengono resi disponibili nella memoria dei processori destinatari.

Modello BSP IV

I processi non sono considerati come aventi un determinato ordine lineare (da sinistra a destra o altrimenti), e possono essere mappati in processori in qualsiasi modo.



Risultati

Nell'articolo sono state fornite simulazioni che mostrano che ogni algoritmo BSP eseguito in **t superpassi** con una **memoria di dimensione N e P ≤ N processori** può essere simulato, con elevata probabilità, da un algoritmo MapReduce memory-bound:

- Che impiega t round
- Con una complessità dei messaggi pari a O(tN)
- Con di dimensione dei buffer dei reducer B= O(N/ P).

In aggiunta, per facilitare tale simulazione, si è fornita una soluzione al problema di indicizzazione di un insieme di input per MapReduce che comporta di identificare ciascun input x_i con l'indice i, il che non è, in generale, incluso nel formato di input per un algoritmo MapReduce.

Tale soluzione usa la tecnica del **“B-tree invisibile”.**

Risultati II

Sono state fornite simulazioni che mostrano che qualsiasi algoritmo CRCW PRAM eseguito in **t passi con P processori su una memoria di dimensione N** può essere simulato, con elevata probabilità, con un algoritmo memory-bound MapReduce randomizzato

- In $O(t \log_B P)$ rounds
- Con una complessità dei messaggi pari a $O(t (N + P) \log_B N)$.

Anche in questo caso si usa la tecnica del B-tree invisibile.

Risultati III

Oltre a fornire questi risultati, si mostra come l'approccio risolve diversi problemi di geometria computazionale parallela tra cui:

- L'ordinamento:
- Il problema di tutti i vicini più vicini monodimensionale (all-nearest neighbors)
- il problema dell' involucro convesso in 2 e 3 dimensioni
- problemi di programmazione lineare per dimensioni fissate

Dove per input di dimensione N:

- Si eseguono in $O(\log_B N)$ round
- Si ha una complessità di messaggi pari a $O(N \log_B N)$.

Risultati ottenuti IV

Così, per il caso in cui B sia $\Theta(N^{\varepsilon})$, per una piccola costante $\varepsilon > 0$, questi algoritmi paralleli di geometria computazionale sono eseguiti in un numero costante di round con complessità dei messaggi lineare nel modello memory-bound MapReduce.

Si noti che, nella maggior parte delle applicazioni, sarebbe infatti aspettato che la dimensione del buffer dei reducer sia $O(N^{\varepsilon})$ in quanto il framework MapReduce è progettato per risolvere problemi che sono troppo grandi per entrare nello spazio di memoria di un singolo computer, ma non così grande da essere esponenzialmente maggiore dello spazio di memoria di un singolo computer.

In tali casi gli algoritmi di geometria computazionale presentati sono entro fattori costanti dall'ottimo.

Il problema di tutti i vicini più vicini:

Il problema dei vicini più vicini (nearest-neighbors NN) monodimensionale è un problema di geometria computazionale che chiede di identificare il luogo dei punti che giacciono più vicini ad un punto detto punto di query.

Il problema di tutti i vicini più vicini monodimensionale è un NN applicato ad ogni punto del piano.

Questo potrebbe chiaramente essere risolto eseguendo un NN per ogni punto, ma causerebbe ridondanza di informazioni tra diverse query.
Come semplice esempio si consideri il caso in cui sia trovata la distanza dal punto x al punto y:

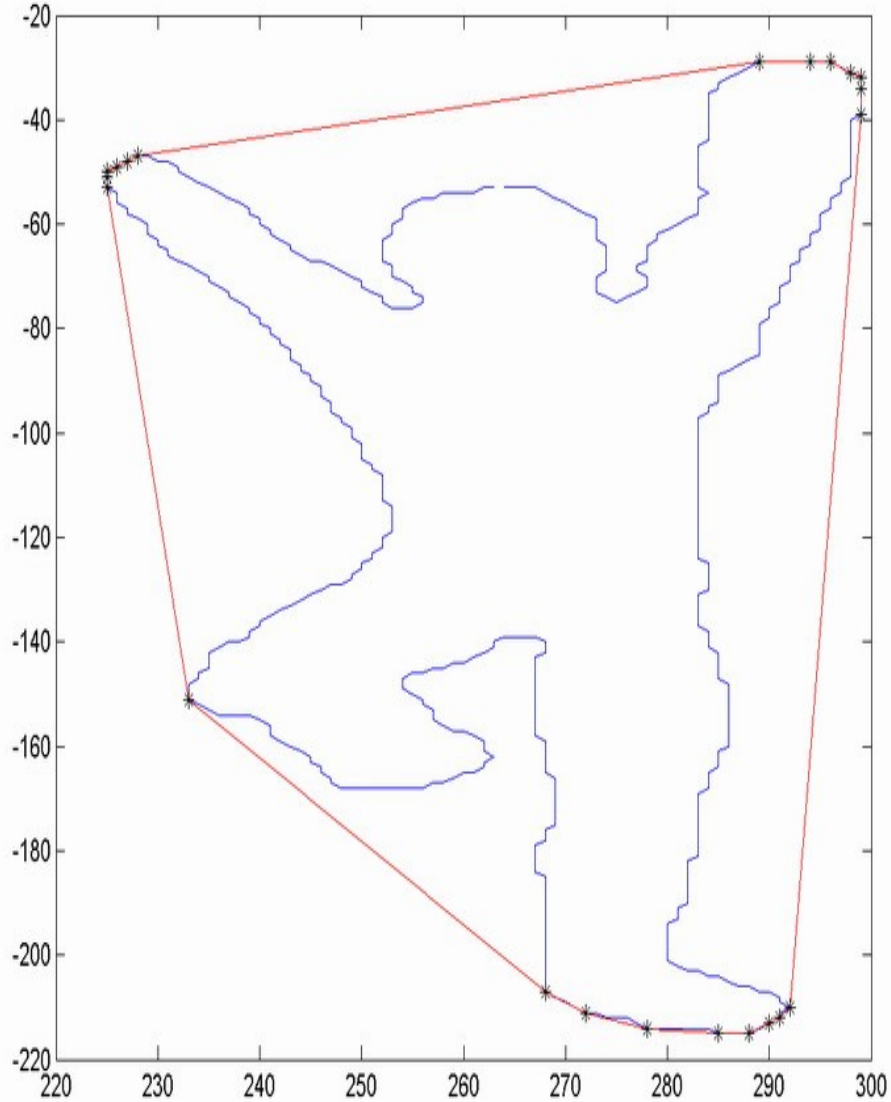
questa ci dice anche la distanza dal punto di y al punto x, per cui lo stesso calcolo può essere riutilizzato in due query diverse.

Il problema dell' involucro convesso:

Il problema di trovare l'involucro convesso di un insieme finito di punti su un piano o su uno spazio euclideo a bassa dimensionalità è uno dei problemi fondamentali della geometria computazionale.

In matematica, involucro convesso per un insieme di punti S nel piano euclideo o nello spazio euclideo è il minimo insieme convesso contenente S.

Per esempio, quando S è un sottoinsieme limitato del piano, l'involucro convesso può essere visualizzato come la forma formata da un elastico tirato attorno ad S.



B-tree:

Un B-tree di grado minimo t è un albero radicato con le seguenti proprietà:

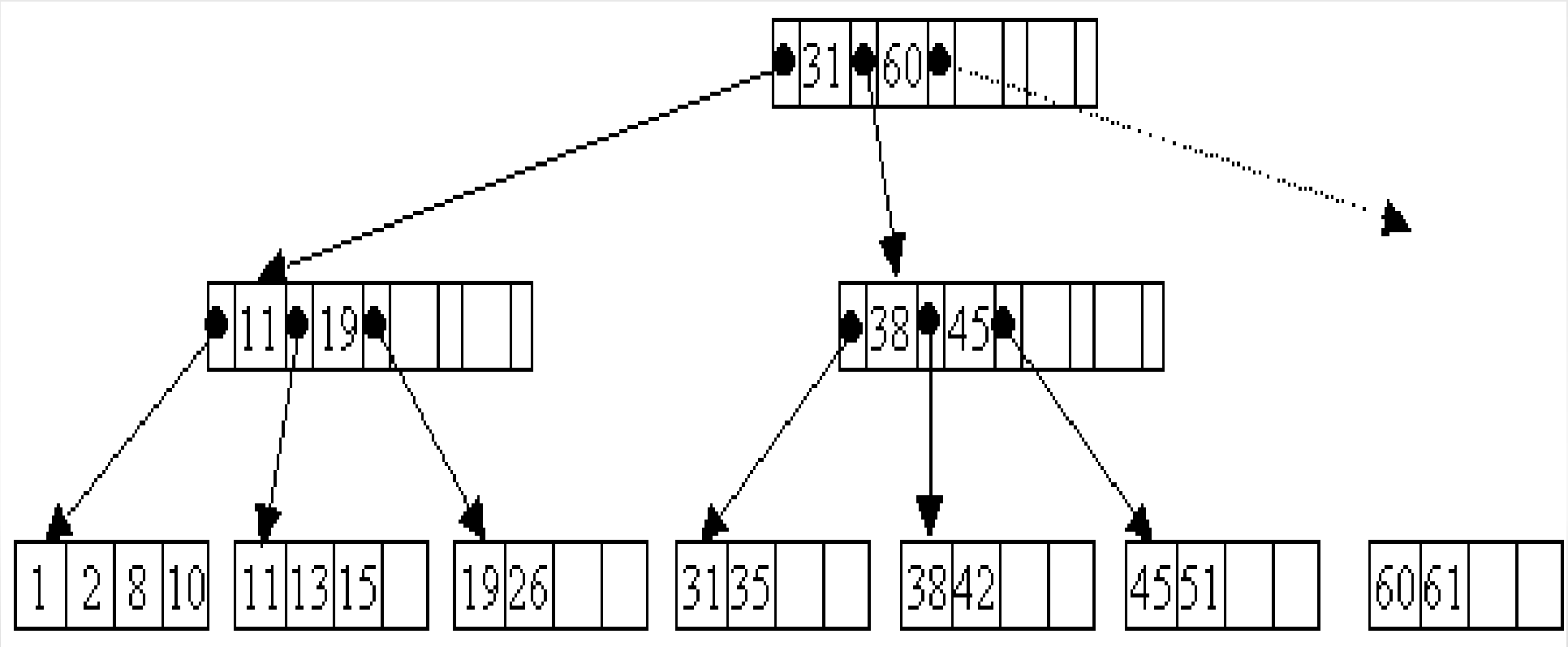
- Tutte le foglie hanno la stessa profondità;
- Ogni nodo v diverso dalla radice mantiene k(v) chiavi ordinate con

  $t-1 \leq k \leq 2t-1$;
- La radice r mantiene k(r) chiavi ordinate con $1 \leq k \leq 2t-1$;
- Ogni nodo interno ha almeno k(v)+1 figli;
- Le chiavi soddisfano la proprietà di ricerca estesa;

B-tree II

La loro struttura ne garantisce il bilanciamento.

Massimizzando il grado minimo t l'altezza dell'albero si riduce e l'operazione di bilanciamento è necessaria meno spesso aumentando così l'efficienza.



Il metodo dei b-tree invisibili

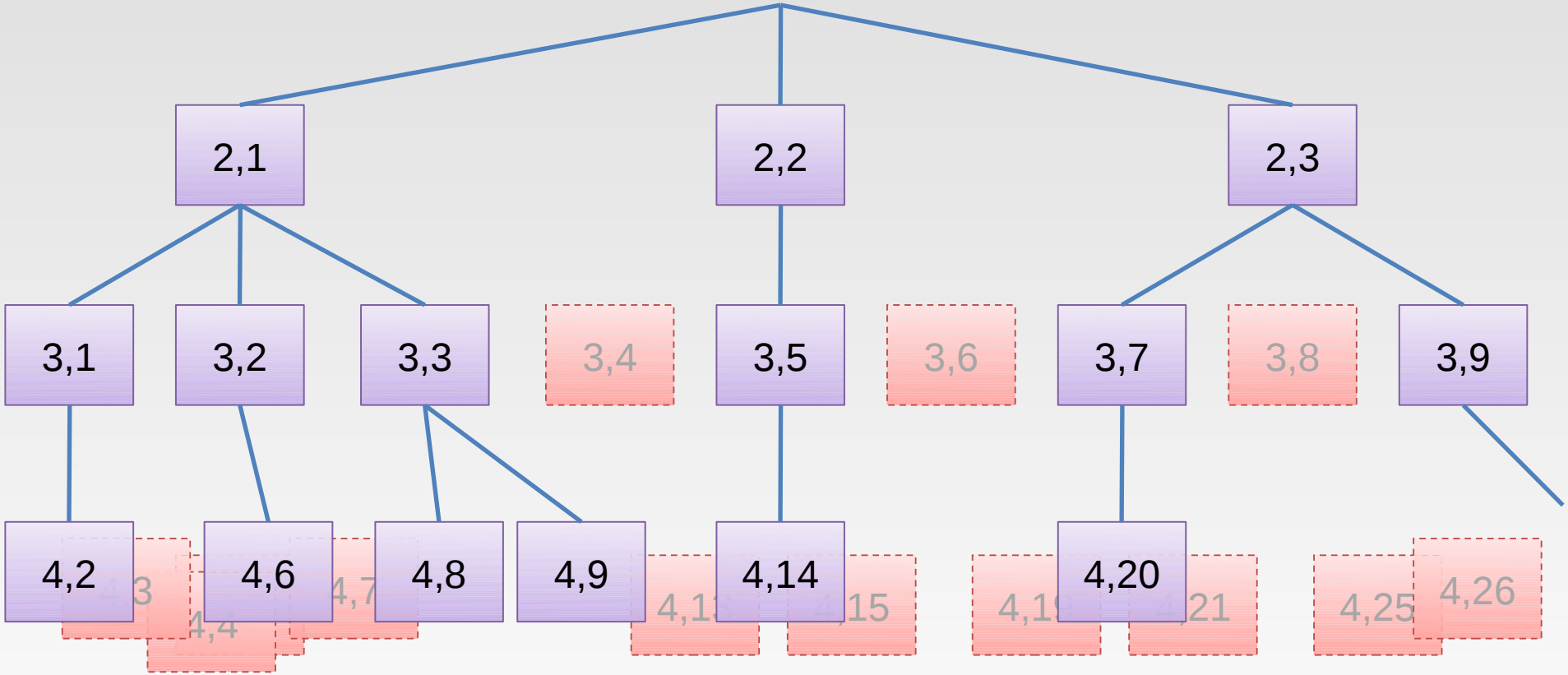
Si immagina di avere un B-tree T di altezza L completo.

Si etichettano i nodi in T in modo che l'i-esimo nodo sul livello l sia etichettato con [l, i].
Così facendo il padre di un nodo, che non sia la radice, etichettato [l, i], è etichettato con [l - 1, $\lfloor i/B \rfloor$], dove B è il numero di figli di ogni nodo.

In altre parole, dato un nodo v in T, si possono identificare sia suo padre sia i suoi figli basandosi unicamente sulla sua etichetta.

Il metodo dei b-tree invisibili II

Consente alle computazioni di utilizzare solo i nodi necessari.



Indicizzare l'Input

L'input di un algoritmo BSP o PRAM è implicitamente indicizzato, in modo che, per ogni input x_i, si conosce il valore di i.

L'input di un computazione MapReduce, invece, non lo è.

Quindi, un primo passo necessario per essere in grado di simulare un algoritmo BSP o PRAM nel framework MapReduce è quello di indicizzare l'input.

Il metodo non richiede che tutti i mappers debbano conoscere il valore di N (la dimensione dell'input), ma si assume che vi è un valore N' tale che N' sia noto a tutti i mappers e che $N \leq N' \leq bN^c$, per qualche costante $b \geq 1$ e $c \geq 1$.

Indicizzare l'Input: Teorema 2.1

Si utilizza un algoritmo che calcola le somme prefisse di tutti i valori iniziali associati agli input x_i.

Impostando tutti questi valori associati ad 1, l'algoritmo calcola l'indice i di ogni x_i.

- Teorema 2.1:

  Dato un insieme X di N valori, si può calcolare,con elevata probabilità, un indicizzazione casuale dei valori in X da 1 a N, o un ordine casuale degli elementi di X e un insieme di somme prefisse basato su questo ordine, nel framework memory-bound MapReduce in $O(\log_B N)$ round e con una complessità dei messaggi di $O(N \log_B N)$.

Indicizzare l'Input: il metodo

Si usa la tecnica del B-tree invisibile utilizzando un B-tree T di altezza $L=\lceil 3\log_B N' \rceil$, in modo che vi siano N'^{3} foglie in T.

Si eseguono 3 fasi:

- La fase di inizializzazione;
- La fase bottom-up;
- La fase top-down;

Indicizzare l'Input: il metodo II

- Fase d'inizializzazione:

  Per ogni valore in input x_i si mappa x_i nella tupla (v_i, x_i, **1**), dove $\mathbf{v_i}$ è un nodo scelto a caso tra le foglie di T.

  Cioè si sceglie per x_i un nodo v_i che è etichettato [L, r_i], dove r_i è un numero casuale scelto uniformemente nell'intervallo [1, N'^3].

- La fase bottom-up:

  Si esegue la somma sui nodi dei B-tree.

  Ogni foglia invia il proprio valore al padre.

  Ogni nodo non foglia somma i valori ottenuti dai figli e invia quest'ultimo valore al padre fino ad arrivare alla radice del B-tree.

Indicizzare l'Input: il metodo III

- Fase di top-down:

  Si eseguono le somme prefisse.

  La radice invia la somma dei valori ricevuti dai figli 0,....,i-1 al figlio i e termina.

  Per ogni livello l ad eccezione dell'ultimo, ogni nodo v riceve un valore sp dal padre, lo somma ai valori ricevuti dai figli 0,....,i-1 al passo precedente e lo invia al figlio i.

  Le foglie riceveranno un valore dal padre che sommeranno al proprio per ottenere la somma prefissa finale.

Teorema 2.1 implicazioni

Nella versione MapReduce del problema di ordinamento classico viene dato un insieme X di elementi confrontabili e viene chiesto di calcolare per ogni elemento x in X il numero di elementi di X che sono minori o uguali ad x.

Dato che si può calcolare un'indicizzazione su X, si può assumere senza perdita di generalità che gli elementi di X siano tutti distinti e si può così risolvere il problema dell'ordinamento.

Con questo metodo si può anche risolvere il problema di tutti i vicini più vicini monodimensionale, in cui, nella versione MapReduce, per ogni x in X, viene chiesto di trovare il più piccolo elemento in X che è più grande di x.

In entrambi i problemi si utilizzano

- $O(\log_B N)$ rounds e complessità dei messaggi di $O(N \log_B N)$.

Questa risultato è ottimale fino a fattori costanti per questo modello.

Simulare algoritmi BSP

L’algoritmo utilizza P reducer ognuno dei quali ha bisogno di memoria $B=\lceil N/P \rceil$.

Il motivo di tale necessità è che la memoria del BSP (di dimensione N) è formata dalle P memorie locali dei processori, ed ognuno di essi è simulato da un reducer che quindi deve avere spazio almeno $B=\lceil N/P \rceil$.

Come sappiamo ogni processore del BSP esegue una certa computazione e al termine del superstep invia al più M messaggi alla barriera.

Simulare algoritmi BSP II

- Fase d'inizializzazione dell'input:

  Si crea una tupla per ogni cella di memoria e per ogni processore.

- Map:

  Serve per inviare le richieste ai reducer destinatari.
  In pratica simula la barriera di sincronizzazione che riceve i messaggi e li invia ai corrispondenti reducer destinatari mappando ogni messaggio nell'etichetta del reducer di destinazione.

- Reduce:

  I reducer si utilizzando per eseguire le computazioni parallele dei singoli processori PRAM.

  Si esegue un passo del processore P, dando in output il proprio insieme di celle di memoria in modo da poterlo ricevere nuovamente al round successivo dato che non possono salvare i dati in memoria permanentemente per più di un round.

Simulare algoritmi BSP implicazioni:

Goodrich, fornendo un algoritmo BSP ottimo per il problema dell'involucro convesso, mostra che si può calcolare una rappresentazione dell'involucro convesso di un insieme di N punti nel piano (o nello spazio), con elevata probabilità, nel modello di memory-bound MapReduce con:

- complessità temporale $O(\log_B N)$;
- complessità dei messagi di $O(N \log_B N)$;

Simulare algoritmi PRAM CRCW tramite B-tree invisibili

In particolare è stata proposta una simulazione di algoritmi PRAM f-CRCW, dove scritture concorrenti a stesse locazioni in memoria vengono risolte tramite un operatore commutativo f .

Si consideri ora come si può simulare uno di tali algoritmi, chiamato A.

L'ostacolo principale nella simulazione dell'algoritmo A è che vi possono essere fino a **P letture e P scritture** sulla stessa cella di memoria in ogni passo.

Le richieste di scrittura dovranno essere valutate efficientemente nei reducer e ognuno di essi ha $B \leq P$ spazio di memoria a disposizione, dove P può essere significativamente maggiore di B.

Per orientare le richieste di lettura e di scrittura si usa la tecnica del B-tree invisibile, dove si immagina che vi è un diverso B-tree invisibile di altezza L in ciascuna cella di memoria e che ha l'insieme dei processori come foglie.

Simulare algoritmi PRAM CRCW tramite B-tree invisibili II

La simulazione di un singolo passo dell'algoritmo A di fatto prevede la computazione del passo e il routing delle richieste di lettura e scrittura negli N B-trees di altezza L.

In dettaglio:

- Fase di lettura bottom-up;
- Fase di lettura top-down;
- Computazione interna;
- Fase di scrittura, bottom-up;

Simulare algoritmi PRAM CRCW tramite B-tree invisibili III

- Fase di lettura 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 B-tree, ad esempio la richiesta (j, L, i).

  Per ogni livello l da 1 a L − 1, ogni nodo v invia la richiesta di lettura ricevuta al ruond precedente al padre p(v).

- Fase di lettura 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 − 1, ogni nodo che riceve un valore dal padre, invia tale valore ai figli che nella fase bottom-up l'avevano richiesto.

Simulare algoritmi PRAM CRCW tramite B-tree invisibili IV

- 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.

- Fase di scrittura bottom-up:

  Per ogni livello l da L − 1 a 1, ogni nodo v che riceve k richieste di scrittura dai suoi figli applica la funzione f all’input z_1, ..., z_k .

  Se il nodo v non è la radice, invia tale risultato al padre, altrimenti modifica il valore corrente in memoria.

Teorema 5.1:

- Teorema 5.1
  Dato un algoritmo A nel modello CRCW PRAM con i conflitti di scrittura risolti tramite un'operatore commutativo f tale che A si esegue in t passi usando P processori ed N celle di memoria, si può simulare A nel framework memory-bound MapReduce in $O(t \log_B P)$ round e con complessità dei messaggi di
  $O(t (N + P) \log_B P)$.

Corollario 5.2:

- Corollario 5.2:

  Dato un insieme S di N vincoli lineari in R^d, per una costante fissa d≥1, ed una funzione di ottimizzazione lineare, si può risolvere, con elevata probabilità, questo problema di programmazione lineare n nel framework memory-bound MapReduce in $O(\log_B N)$ round e con complessità dei messaggi di $O(N \log_B N)$

- Dimostrazione:

  Alon e Megiddo forniscono un algoritmo CRCW PRAM di programmazione lineare che si esegue, con una probabilità molto alta, in O(1) tempo per dimensioni fissate, utilizzando N processori e O(N) celle di memoria.

  Applicando il Teorema 2.1 per indicizzare i punti di input e simulando questo algoritmo come determinato dal Teorema 5.1 si ottiene il risultato richiesto.

Conclusioni

Sono state date simulazioni efficienti di algoritmi paralleli BSP e PRAM nel framework memory-bound MapReduce.

Inoltre, tali simulazioni portano ad algoritmi efficienti di MapReduce per:

- L'ordinamento
- Il problema di tutti i vicini più vicini monodimensionale (all-nearest neighbors)
- il problema dell' involucro convesso in 2 e 3 dimensioni
- E problemi di programmazione lineare per dimensioni fissate

Dichiaratamente, gli algoritmi utilizzati sono asintoticamente efficienti però non sono tra i più semplici degli algoritmi.

Conclusioni II

Così un problema aperto è quello di progettare algoritmi efficienti, ma semplici, nel modello memory-bound MapReduce per i problemi che sono stati trattati nell'articolo.

Inoltre, gli algoritmi memory-bound MapReduce che sono stati discussi in questo articolo fanno uso esplicito della conoscenza della dimensione del buffer dei reducer B.

Sarebbe interessante trovare un modo per esprimere efficienti algoritmi memory-bound MapReduce che non conoscono questo valore, più o meno nello stesso modo in cui, nel contesto di memoria esterna, algoritmi cache-oblivious non conoscono la dimensione dei blocchi.

Bibliografia

- M. T. Goodrich, *Simulating Parallel Algorithms in the MapReduce Framework with Applications to Parallel Computational Geometry,* MASSIVE 2010.
- J.Dean, S.Ghemawat *MapReduce: Simplified Data Processing on Large Clusters,* in OSDI'04: 137-150.
- G. Crisafulli. *MapReduce: Modelli e Algoritmi,* Università degli Studi di Roma "La Sapienza" .
- L. G. Valiant, *Bulk-Synchronous Parallel computers*, Technical Report TR-08-89, Aiken Computation Laboratory, Harvard University, 1989.
- DJ O'Neil, *Nearest neighbors problem*, Department of Computer Science and Engineering, University of Minnesota.

Bibliografia II

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduzione agli algoritmi e strutture dati*, 3/ed, McGraw-Hill, capitolo 33.
- C. Demetrescu, I. Finocchi, G. F. Italiano. *Algoritmi e strutture dati*, 2/ed, McGraw-Hill, capitolo 6.