COMP 322: Fundamentals of Parallel Programming

Lecture 36: Map Reduce (contd)

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments for Today's Lecture

- Slides from Lectures 1 and 2 in UC Berkeley CS61C course, "Great Ideas in Computer Architecture (Machine Structures), Spring 2012, Instructor: David Patterson
  - http://inst.eecs.berkeley.edu/~cs61c/sp12/
- Slides from MapReduce lecture in Stanford CS 345A course
  - http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt
- Slides from COMP 422 lecture on MapReduce
  - http://www.clear.rice.edu/comp422
- Slides from Google Cluster Computing Faculty Training Workshop
  - Module IV: MapReduce Theory, Implementation, and Algorithms

Outline

- <u>Execution model for Map-Reduce Programs</u>

- Map Reduce Algorithms

Recap of Map-Reduce Model: Operations on Sets of Key-Value Pairs

- Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume that the key and value objects are immutable, and that equality comparison is well defined on all key objects.
- Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1' ,v1'),...(km',vm')}. The kj' keys can be different from ki key in the input of the map function.
  - Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.
- Reduce operation groups together intermediate key-value pairs, {(k', vj')} with the same k', and generates a reduced key-value pair, (k',v''), for each such k', using reduce function g

Summary of MapReduce API

- Programmers must specify:

  map (k, v) → list(<k', v'>)

  reduce (k', list(v')) → <k'', v''>

    All values with the same key are reduced together

Optionally, also:

  partition (k', number of partitions) → partition for k'

    Often a simple hash of the key, e.g., hash(k') mod n

    Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*

    Mini-reducers that run in memory after the map phase

    Used as an optimization to reduce network traffic

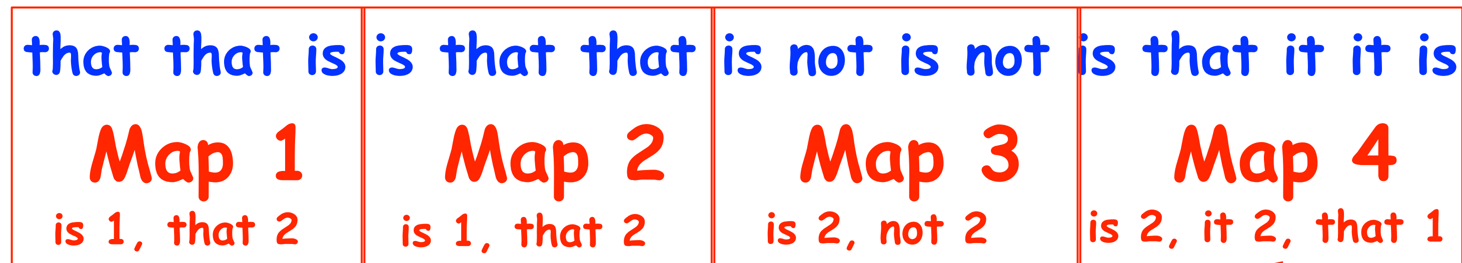
The execution framework handles everything else…

PseudoCode for WordCount

```
1.   map(String input_key, String input_value):
2.     // input_key: document name
3.     // input_value: document contents
4.     for each word w in input_value:
5.       EmitIntermediate(w, "1"); // Produce count of words
6.
7.   reduce(String output_key, Iterator intermediate_values):
8.     // output_key: a word
9.     // intermediate_values: a list of counts
10.     int result = 0;
11.     for each v in intermediate_values:
12.       result += ParseInt(v); // get integer from key-value
13.     Emit(AsString(result));
```

Example Execution of WordCount Program

Distribute



Shuffle

Collect

is 6; it 2; not 2; that 5

Overall schematic for MapReduce framework on a data center cluster

Input key*value pairs

Input key*value pairs

...

Data store 1

map

Data store n

map

(key 1, values...)

(key 2, values...)

(key 3, values...)

(key 1, values...)

(key 2, values...)

(key 3, values...)

== Barrier == : Aggregates intermediate values by output key

key 1, intermediate values

key 2, intermediate values

key 3, intermediate values

reduce

reduce

reduce

final key 1 values

final key 2 values

final key 3 values

MapReduce Processing

User Program

(1) fork

(1) fork

(1) fork

Master

(2) assign map

(2) assign reduce

split 0
split 1
split 2
split 3
split 4

(3) read

worker

worker

worker

(4) local write

(5) remote read

worker

worker

(6) write

output file 0

output file 1

Input files

Map phasr

Intermediate files (on local disks)

Reduce phase

Output files

MapReduce Processing

1. MR 1st splits the input files into M "splits" then starts many copies of program on servers

User Program

(1) fork

(1) fork

(1) fork

Master

(2) assign map

(2) assign reduce

split 0

split 1

split 2

split 3

split 4

(3) read

worker

worker

worker

(4) local write

(5) remote read

worker

worker

(6) write

output file 0

output file 1

Input files

Map phasr

Intermediate files (on local disks)

Reduce phase

Output files

MapReduce Processing

2. One copy—the master—is special. The rest are workers. The master picks idle workers and assigns each 1 of M map tasks or 1 of R reduce tasks.

MapReduce Processing

3. A map worker reads the input split. It parses key/value pairs of the input data and passes each pair to the user-defined map function.

(The intermediate key/value pairs produced by the map function are buffered in memory.)

User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
split 0
split 1
split 2
split 3
split 4
(3) read
worker
worker
worker
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce Processing

4. Periodically, the buffered pairs are written to local disk, partitioned
into R regions by the partitioning function.

User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
split 0
split 1
split 2
split 3
split 4
(3) read
worker
worker
worker
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce Processing

5. When a reduce worker has read all intermediate data for its partition, it bucket sorts using inter-mediate keys so that occur-rences of same keys are grouped together

(The sorting is needed because typically many different keys map to the same reduce task)

User Program

(1) fork

(1) fork

(1) fork

Master

(2) assign map

(2) assign reduce

worker

worker

worker

split 0

split 1

split 2

split 3

split 4

(3) read

(4) local write

(5) remote read

worker

worker

(6) write

output file 0

output file 1

Input files

Map phasr

Intermediate files (on local disks)

Reduce phase

Output files

Shuffle phase

MapReduce Processing

6. Reduce worker iterates over sorted intermediate data and for each unique intermediate key, it passes key and corresponding set of values to the user's reduce function.

The output of the reduce function is appended to a final output file for this reduce partition.

User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1

Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce Processing

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. The MapReduce call in user program returns back to user code.

Output of MR is in R output files (1 per reduce task, with file names specified by user); often passed into another MR job so don't combine

User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phasr
Intermediate files (on local disks)
Reduce phase
Output files

MapReduce is a Data-Parallel form of the "Divide and Conquer" Pattern

- **Map:**
  - Slice data into "shards" or "splits", distribute these to workers, compute sub-problem solutions
  - `map(in_key,in_value)->list(out_key,intermediate value)`
    - Processes input key/value pair
    - Produces set of intermediate pairs
- **Reduce:**
  - Collect and combine sub-problem solutions
  - `reduce(out_key,list(intermediate_value))->list(out_value)`
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values
- Easy to use: focus on problem, let MapReduce library deal with messy details

MapReduce Failure Handling

- **On worker failure:**
  - —Detect failure via periodic heartbeats
  - —Re-execute completed and in-progress map tasks
  - —Re-execute in progress reduce tasks
  - —Task completion committed through master
- **Master failure:**
  - —Could handle, but don't yet (master failure unlikely)
- **Robust: lost 1600 of 1800 machines once, but finished fine**

MapReduce Redundant Execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time
  - 3% more resources, large tasks 30% faster

MapReduce Locality Optimization during Scheduling

- Master scheduling policy:
  - Asks GFS (Google File System) for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect: Thousands of machines read input at local disk speed
- Without this, rack switches limit read rate

Additional Optimization: Combiner Functions

- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Outline

- Execution model for Map-Reduce Programs
- <u>Map Reduce Algorithms</u>

Algorithms for MapReduce

- <u>Sorting</u>
- Searching
- <u>Indexing</u>
- Classification
- <u>TF-IDF</u>
- <u>Breadth-First Search / SSSP</u>
- PageRank
- Clustering

Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered by hash function
- Mapper: Identity function for value

  $(k, v) \rightarrow (v, _)$

- Reducer: Identity function (k', _) -> (k', "")
- Trick: (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
  - Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow hash(k_1) < hash(k_2)$

Inverted Index: Data flow

Foo

This page contains so much text

Foo map output

contains: Foo
much: Foo
page : Foo
so : Foo
text: Foo
This : Foo

Bar

My page contains text too

Bar map output

contains: Bar
My: Bar
page : Bar
text: Bar
too: Bar

Reduced output

contains: Foo, Bar
much: Foo
My: Bar
page : Foo, Bar
so : Foo
text: Foo, Bar
This : Foo
too: Bar

- **Mapper: For each word in (file, words), map to (word, file)**
- **Reducer: Identity function**

TF-IDF

- **Term Frequency – Inverse Document Frequency**
  - **Relevant to text processing**
  - **Common web analysis algorithm**

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- **| *D* | : total number of documents in the corpus**
- $|\{d : t_i \in d\}|$ **: number of documents where the term t_i appears (that is $n_i \neq 0$).**

Information We Need

- Number of times term X appears in a given document
- Number of terms in each document
- Number of documents X appears in
- Total number of documents

Job 1: Word Frequency in Doc

- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), *n*)
- Combiner is same as Reducer

Job 2: Word Counts For Docs

- **Mapper**
  - Input: ((word, docname), *n*)
  - Output: (docname, (word, *n*))
- **Reducer**
  - Sums frequency of individual *n*'s in same doc
  - Feeds original data through
  - Outputs ((word, docname), (*n*, *N*))

Job 3: Word Frequency In Corpus

- **Mapper**
  - Input: ((word, docname), (*n*, *N*))
  - Output: (word, (docname, *n*, *N*, 1))
- **Reducer**
  - Sums counts for word in corpus
  - Outputs ((word, docname), (*n*, *N*, *m*))

Job 4: Calculate TF-IDF

- **Mapper**
  - Input: ((word, docname), (n, N, m))
  - Assume D is known (or, easy MR to find it)
  - Output ((word, docname), TF*IDF)
- **Reducer**
  - Just the identity function

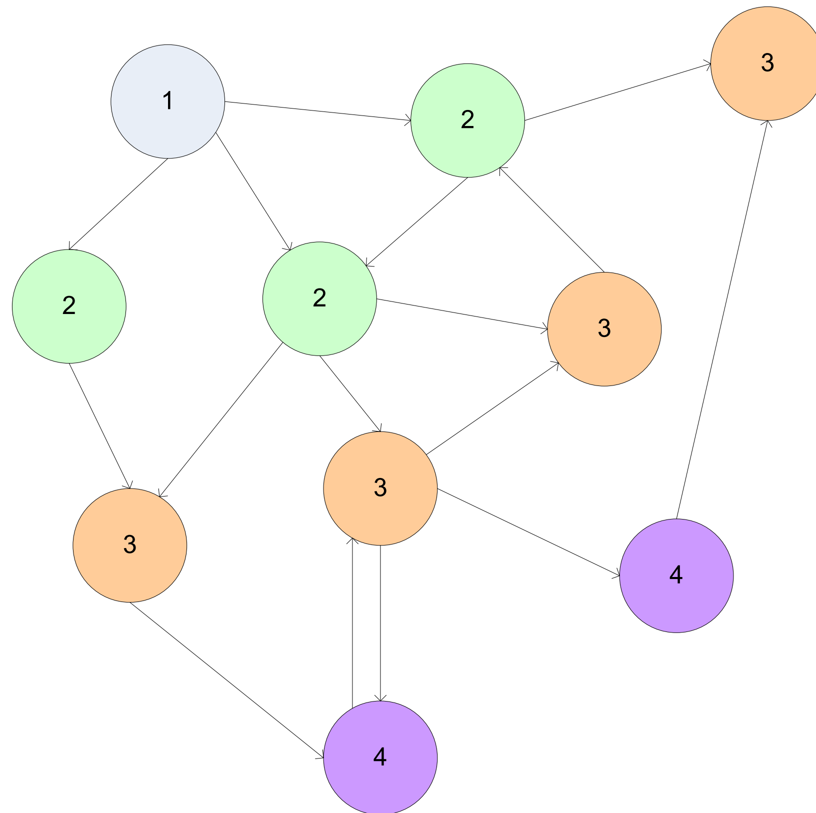
Breadth-First Search (BFS): Motivating Concepts

- **Performing computation on a graph data structure requires processing at each node**
- **Each node contains node-specific data as well as links (edges) to other nodes**
- **Computation must traverse the graph and perform the computation step**

- *How do we traverse a graph in MapReduce? How do we represent the graph for this?*

Breadth-First Search

- **Breadth-First Search is an *iterated* algorithm over graphs**
- **Frontier advances from origin by one level with each pass**



Breadth-First Search & MapReduce

- Problem: This doesn't “fit” into MapReduce
- Solution: Iterated passes through MapReduce – map some nodes, result includes additional nodes which are fed into successive MapReduce passes

Adjacency Matrices

- **Another classic graph representation. M[i][j]= '1' implies a link from node** *i to j.*
- **Naturally encapsulates iteration over nodes**

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	0	1	0

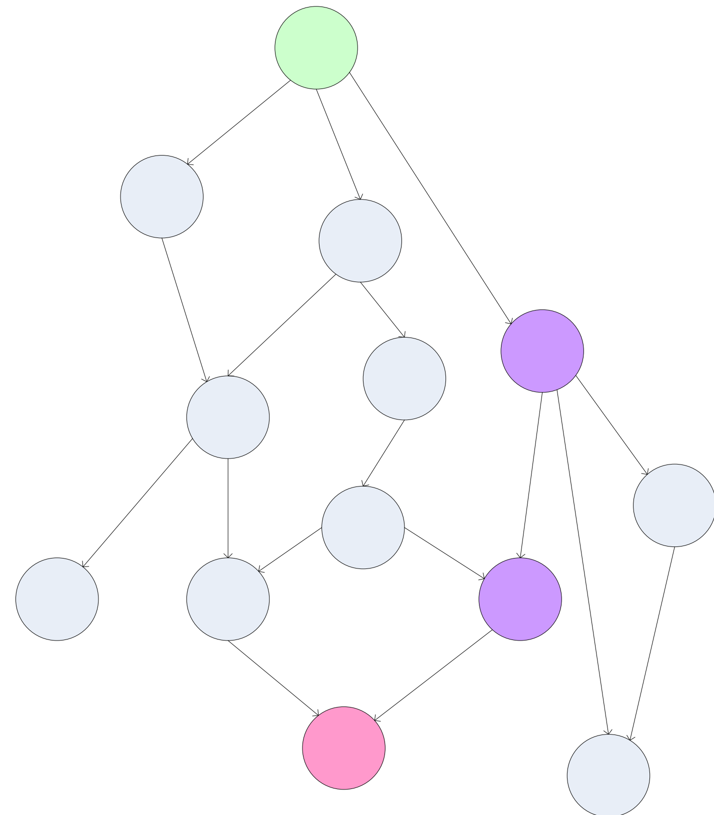
Adjacency Matrices: Sparse Representation

- Adjacency matrix for most large graphs (e.g., the web) will be overwhelmingly full of zeros.
- Each row of the graph is too long to store in a dense manner
- Sparse matrices only include non-zero elements

1: 3, 18, 200
2: 6, 12, 80, 400
3: 1, 14
…

Finding the Shortest Path

- A common graph search application is finding the shortest path from a start node to one or more target nodes
- Commonly done on a single machine with *Dijkstra's Algorithm*
- Can we use BFS to find the shortest path via MapReduce?



This is called the single-source shortest path problem. (a.k.a. SSSP)

Finding the Shortest Path: Intuition

- We can define the solution to this problem inductively:
  - DistanceTo(startNode) = 0
  - For all nodes *n* directly reachable from startNode, DistanceTo(n) = 1
  - For all nodes *n* reachable from some other set of nodes *S*,

    DistanceTo(n) = 1 + min(DistanceTo(m), m $\in$ S)

<u>Algorithm</u>:

- A map task receives a node *n as* a key, and *(D, points-to)* as its value
  - *D* is the distance to the node from the start
  - *points-to* is a list of nodes reachable from *n*
  - $\forall$p $\in$ points-to, emit (p, D+1)
- Reduce task gathers possible distances to a given *p* and selects the minimum one

Termination

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as frontier advances
- Does this ever terminate?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should emit *(n, D)* to ensure that "current distance" is carried into the reducer
- Weighted-edge shortest path is more useful than cost==1 approach
  - Simple change: points-to list in map task includes a weight 'w' for each pointed-to node
    - emit (p, D+w_p) instead of (p, D+1) for each node p
    - Works for positive-weighted graph

Summary of Warehouse Scale Computing and Map Reduce

- Request-Level Parallelism
  - —High request volume, each largely independent of other
  - —Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - —Map: Divide large data set into pieces for independent parallel processing
  - —Reduce: Combine and process intermediate results to obtain final result
- WSC CapEx vs. OpEx
  - —Economies of scale mean WSC can sell computing as a utility
  - —Servers currently dominate capital expense, and power distribution, cooling infrastructure dominate operating expense