Design Patterns for Efficient Graph Algorithms in MapReduce

Jimmy Lin and Michael Schatz
University of Maryland

Tuesday, June 29, 2010
Talk Outline

- Graph algorithms
- Graph algorithms in MapReduce
- Making it efficient
- Experimental results

Punch line: per-iteration running time -69% on 1.4b link webgraph!
What’s a graph?

- \( G = (V, E) \), where
  - \( V \) represents the set of vertices (nodes)
  - \( E \) represents the set of edges (links)
  - Both vertices and edges may contain additional information

- Graphs are everywhere:
  - E.g., hyperlink structure of the web, interstate highway system, social networks, etc.

- Graph problems are everywhere:
  - E.g., random walks, shortest paths, MST, max flow, bipartite matching, clustering, etc.
Source: Wikipedia (Königsberg)
Graph Representation

- \( G = (V, E) \)
- Typically represented as adjacency lists:
  - Each node is associated with its neighbors (via outgoing edges)

```
1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3
```
“Message Passing” Graph Algorithms

- Large class of iterative algorithms on sparse, directed graphs
- At each iteration:
  - Computations at each vertex
  - Partial results (“messages”) passed (usually) along directed edges
  - Computations at each vertex: messages aggregate to alter state
- Iterate until convergence
A Few Examples...

- Parallel breadth-first search (SSSP)
  - Messages are distances from source
  - Each node emits current distance + 1
  - Aggregation = MIN

- PageRank
  - Messages are partial PageRank mass
  - Each node evenly distributes mass to neighbors
  - Aggregation = SUM

- DNA Sequence assembly
  - Michael Schatz’s dissertation
PageRank in a nutshell....

- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
  - With some probability, user randomly jumps around

- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
PageRank: Defined

Given page x with inlinks \( t_1, t_n \), where

- \( C(t) \) is the out-degree of \( t \)
- \( \alpha \) is probability of random jump
- \( N \) is the total number of nodes in the graph

\[
PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}
\]
Sample PageRank Iteration (1)
Sample PageRank Iteration (2)

Iteration 2

$$n_1 (0.066) \rightarrow n_4 (0.3) \rightarrow n_5 (0.3) \rightarrow n_3 (0.166) \rightarrow n_2 (0.166)$$

$$n_1 (0.033) \rightarrow n_4 (0.3) \rightarrow n_5 (0.3) \rightarrow n_3 (0.166) \rightarrow n_2 (0.166)$$

$$n_1 (0.1) \rightarrow n_4 (0.2) \rightarrow n_5 (0.383) \rightarrow n_3 (0.183) \rightarrow n_2 (0.133)$$
PageRank in MapReduce
PageRank Pseudo-Code

1: class Mapper
2:   method MAP(nid n, node N)
3:     p ← N.PAGERANK/|N.ADJACENCYLIST|
4:     Emit(nid n, N)  \[\triangleright\] Pass along graph structure
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       Emit(nid m, p)  \[\triangleright\] Pass PageRank mass to neighbors

1: class Reducer
2:   method REDUCE(nid m, [p_1, p_2, \ldots])
3:     M ← \emptyset
4:     for all p ∈ counts [p_1, p_2, \ldots] do
5:       if IsNode(p) then
6:         M ← p
7:       else
8:         s ← s + p  \[\triangleright\] Sums incoming PageRank contributions
9:     M.PAGERANK ← s
10:    Emit(nid m, node M)

\[\triangleright\] Recover graph structure
Why don’t distributed algorithms scale?
Three Design Patterns

- In-mapper combining: efficient local aggregation
- Smarter partitioning: create more opportunities
- Schimmy: avoid shuffling the graph
In-Mapper Combining

- Use combiners
  - Perform local aggregation on map output
  - Downside: intermediate data is still materialized

- Better: in-mapper combining
  - Preserve state across multiple map calls, aggregate messages in buffer, emit buffer contents at end
  - Downside: requires memory management
Better Partitioning

- Default: hash partitioning
  - Randomly assign nodes to partitions
- Observation: many graphs exhibit local structure
  - E.g., communities in social networks
  - Better partitioning creates more opportunities for local aggregation
- Unfortunately, partitioning is hard!
  - Sometimes, chick-and-egg
  - But in some domains (e.g., webgraphs) take advantage of cheap heuristics
  - For webgraphs: range partition on domain-sorted URLs
**Schimmy Design Pattern**

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)

- Schimmy: separate the two data flows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages

Both relations consistently partitioned and sorted by join key

![Diagram showing the Schimmy Design Pattern](image-url)
Do the Schimmy!

- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS
Experiments

- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3

- Dataset:
  - First English segment of ClueWeb09 collection
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 billion links, 7.0 GB
  - Dataset arranged in crawl order

- Setup:
  - Measured per-iteration running time (5 iterations)
  - 100 partitions
Results

“Best Practices”
Results

[Bar chart showing per-iteration running time (seconds) for different conditions: -Combining, Baseline, +IMC, range partitioning, +Schimmy, with +18% increase from 1.4b to 674m]
Results

![Bar chart showing per-iteration running time (seconds) for different configurations: -Combining, Baseline, +IMC, +range partitioning, +Schimmy. The bars indicate +18% for 1.4b and -15% for 674m.](image)
Results

Per-iteration Running Time (seconds)

- Combining: 1.4b +18%
- Baseline: 674m
- +IMC: -15%
- +range partitioning: 86m -60%
- +Schimmy
Results

![Bar Chart showing per-iteration running time (seconds) for different scenarios: Combining, Baseline, +IMC, +range partitioning, and +Schimmy. The chart includes labels for the running time in seconds and percentage changes. The bars are labeled with values such as +18% for Combining, 674m for Baseline, -15% for +IMC, -60% for +range partitioning, and -69% for +Schimmy.](image-url)
Take-Away Messages

- Lots of interesting graph problems!
  - Social network analysis
  - Bioinformatics

- Reducing intermediate data is key
  - Local aggregation
  - Better partitioning
  - Less bookkeeping

http://mapreduce.me/

Source code available in Cloud⁹

http://cloud9lib.org/

@lintool