Design Patterns for Efficient Graph Algorithms in MapReduce

Jimmy Lin and **Michael Schatz**
University of Maryland

Tuesday, June 29, 2010

@lintool

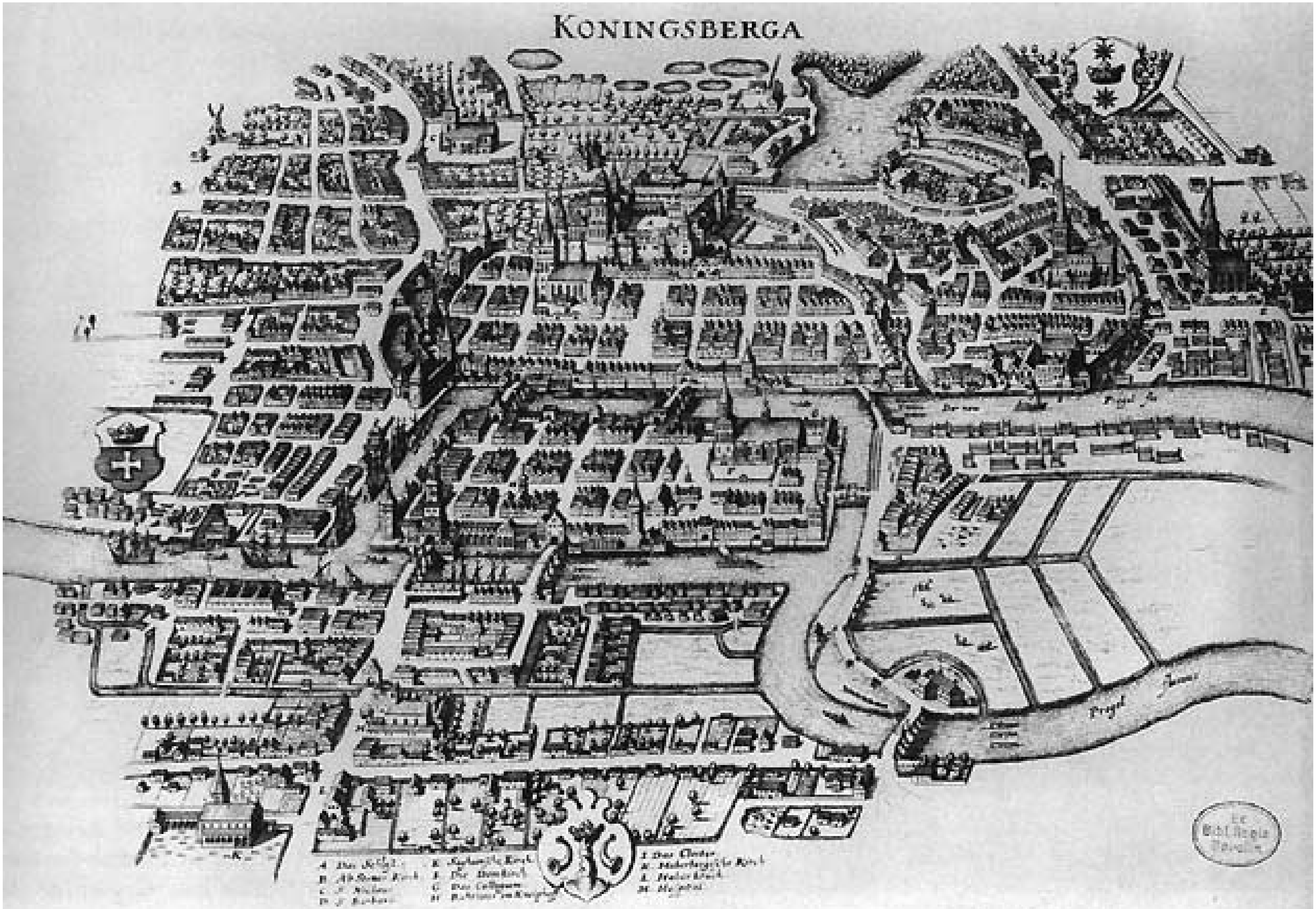
Talk Outline

- 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?

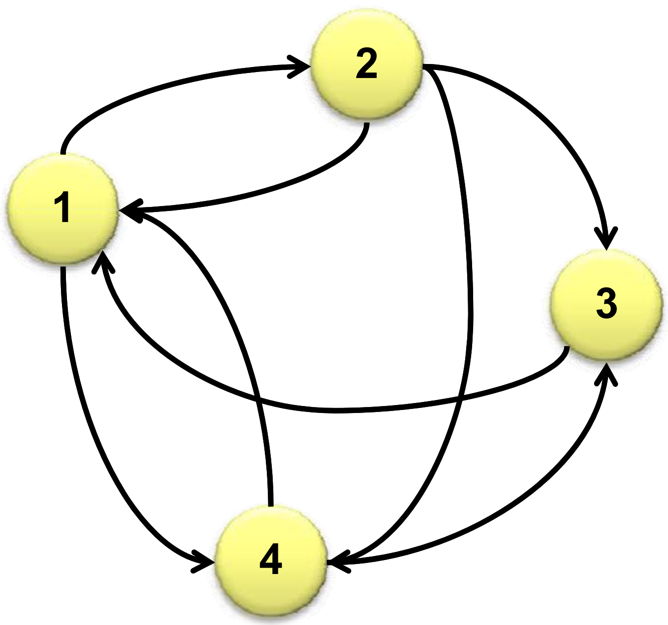
- G = (V, E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Graphs are everywhere:
  - E.g., hyperlink structure of the web, interstate highway system, social networks, etc.
- Graph problems 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

Boring!

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

Still boring!

- DNA Sequence assembly
  - Michael Schatz's dissertation

PageRank in a nutshell….

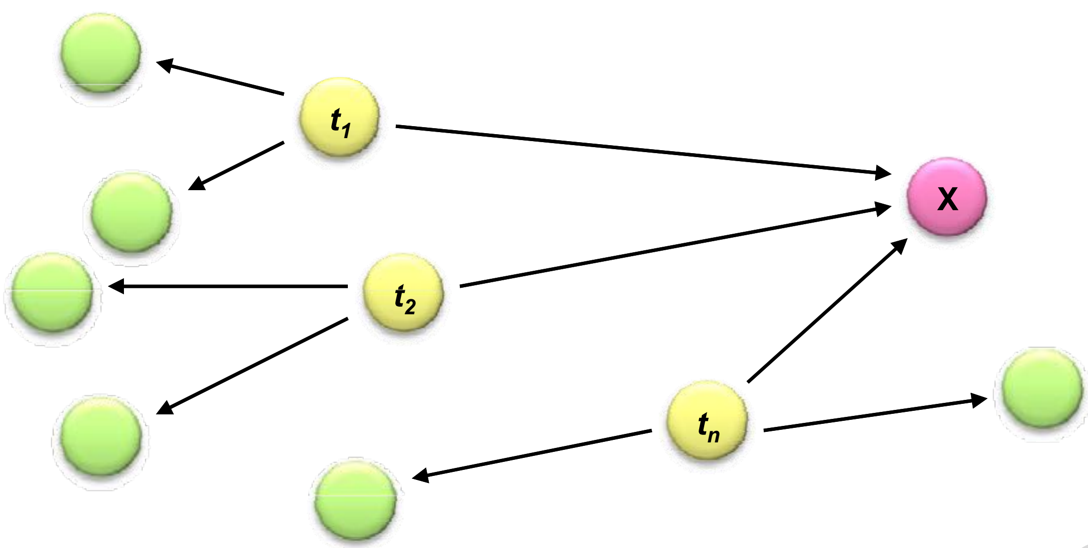
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
  - With some probability, user randomly jumps around
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages

PageRank: Defined

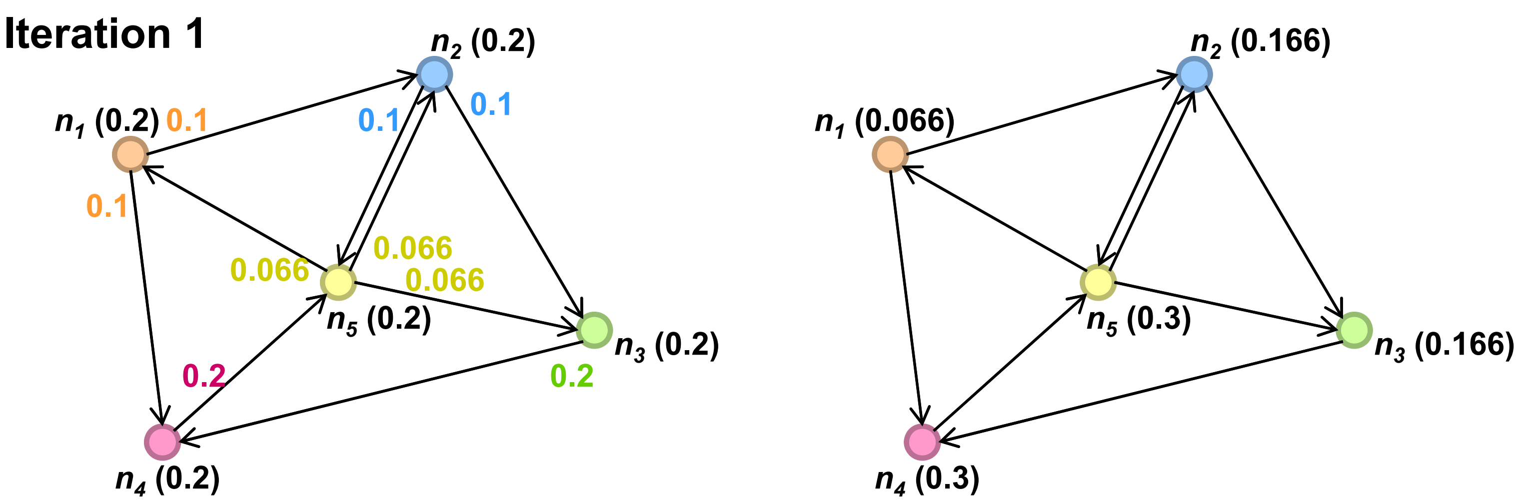
Given page *x* with inlinks $t_1 \ldots t_n$, where

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

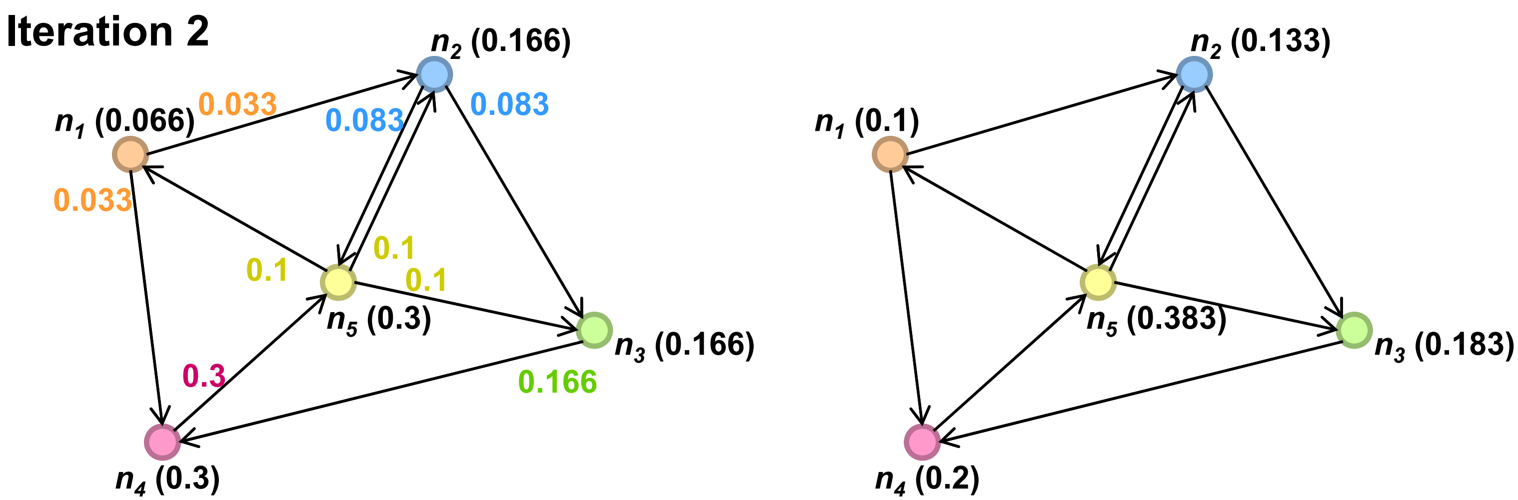
$$PR(x) = \alpha\left(\frac{1}{N}\right) + (1-\alpha)\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$



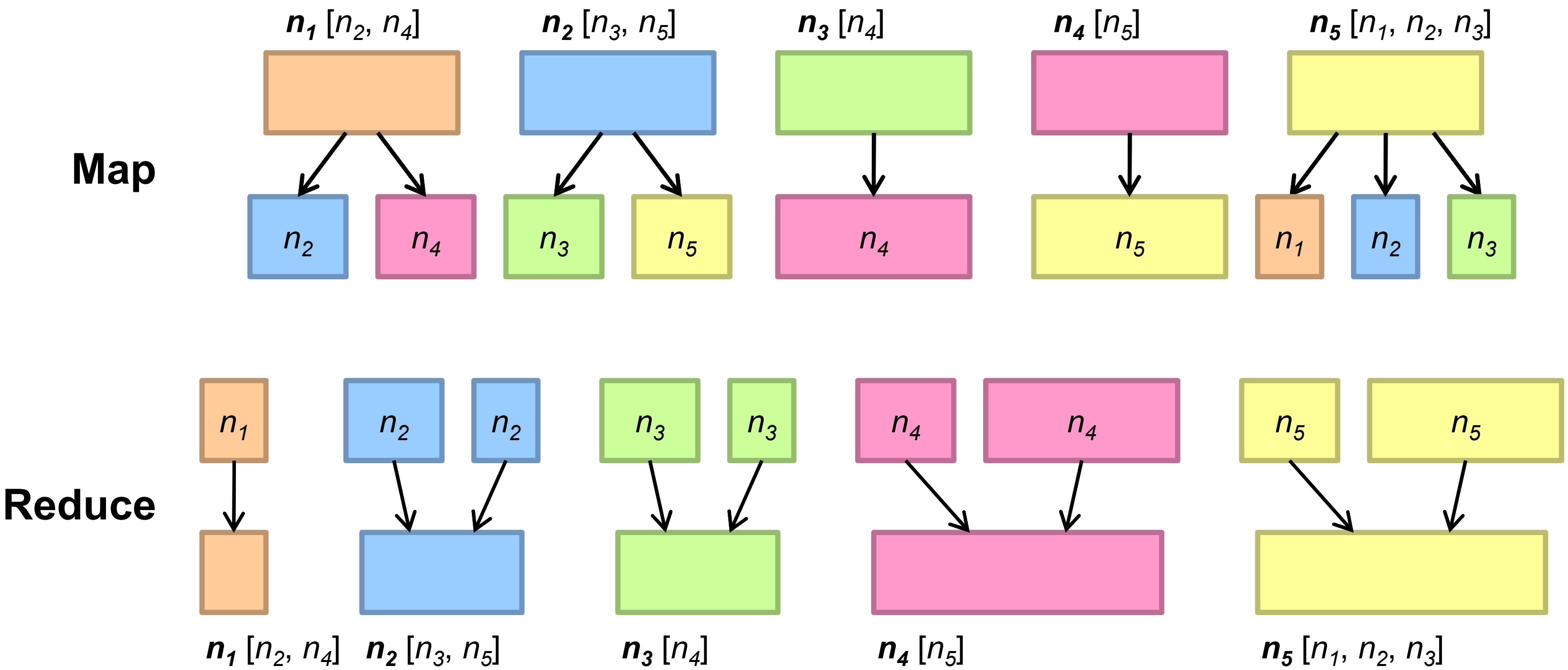
Sample PageRank Iteration (1)



Sample PageRank Iteration (2)



PageRank in MapReduce

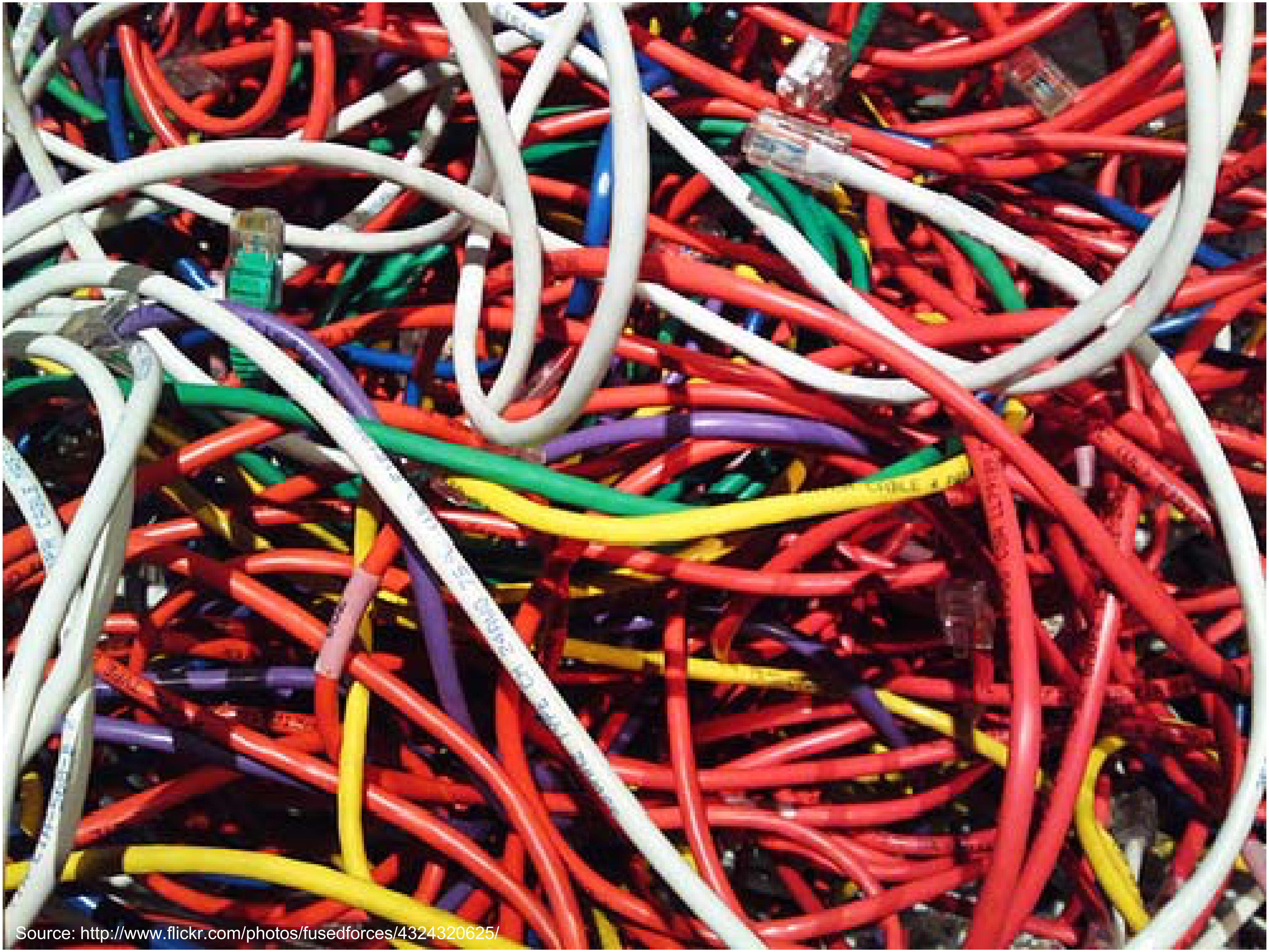


PageRank Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors

class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

Why don’t distributed algorithms scale?



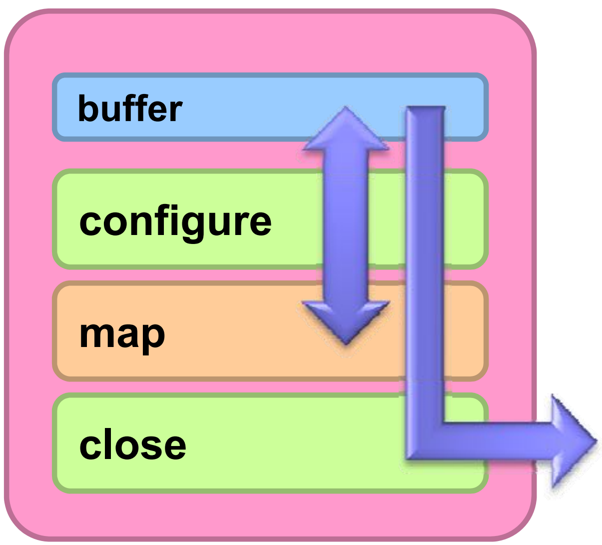
Source: http://www.flickr.com/photos/fusedforces/4324320625/

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



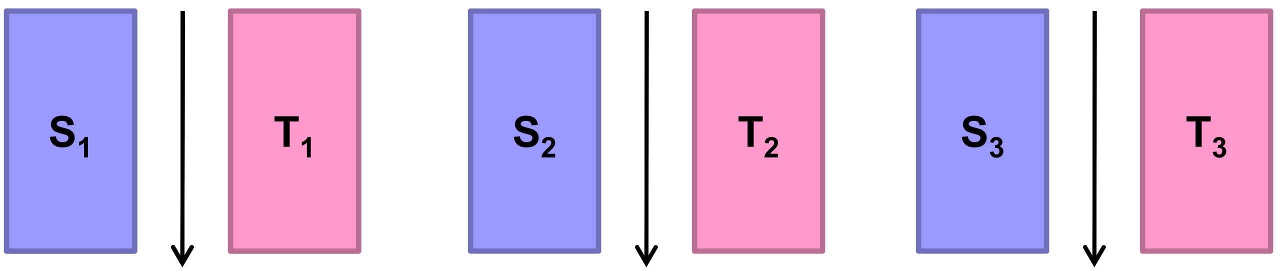
Better Partitioning

- 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

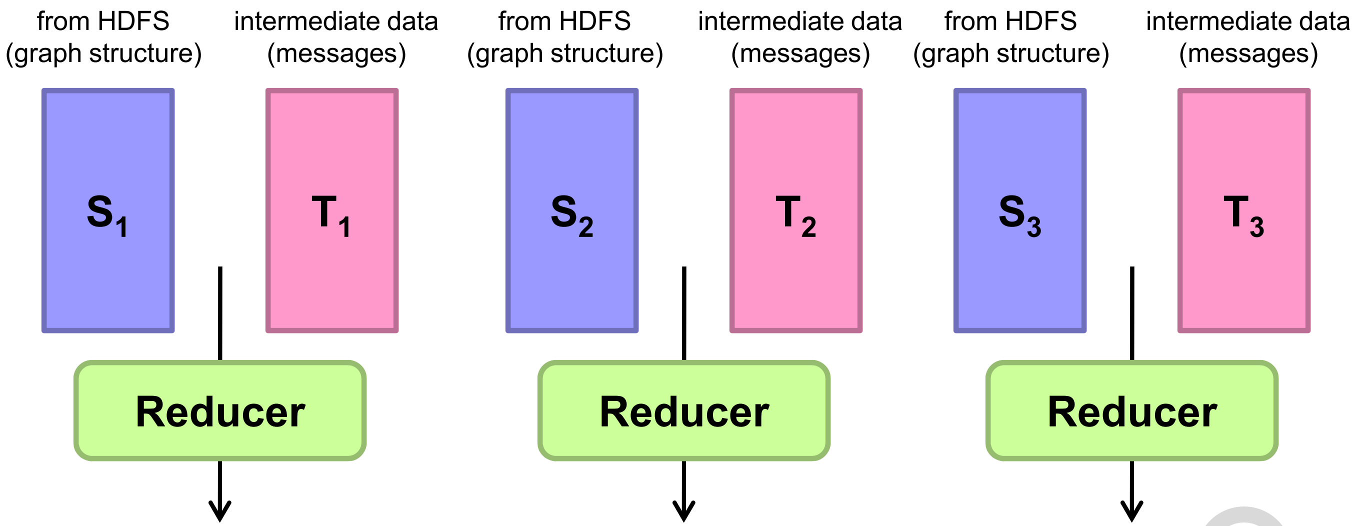
- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two data flows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages

both relations consistently partitioned and sorted by join key



Do the Schimmy!

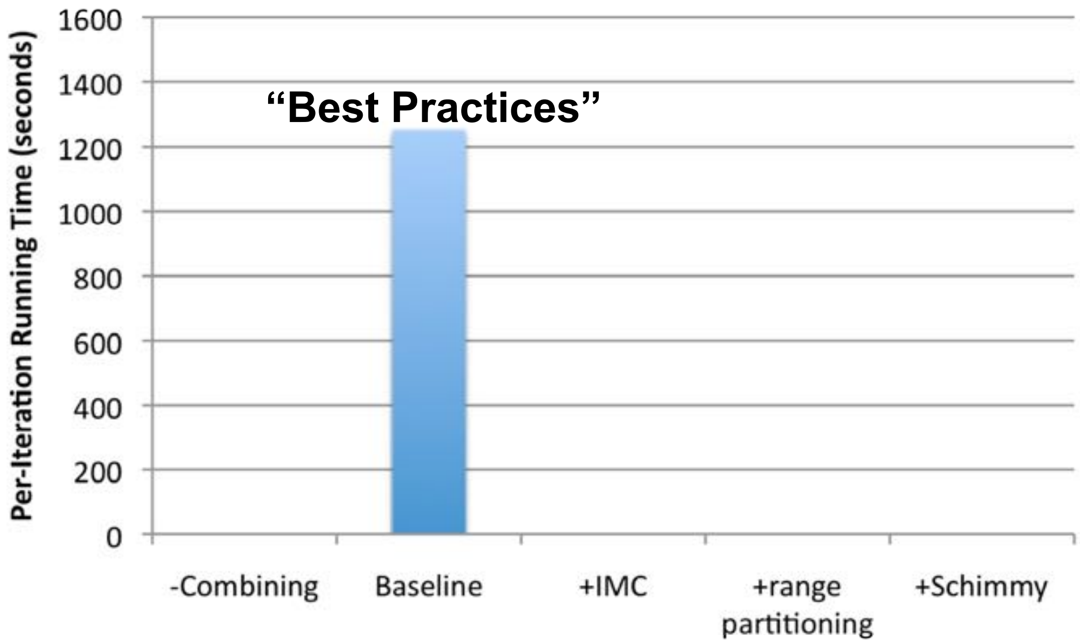
- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS



Experiments

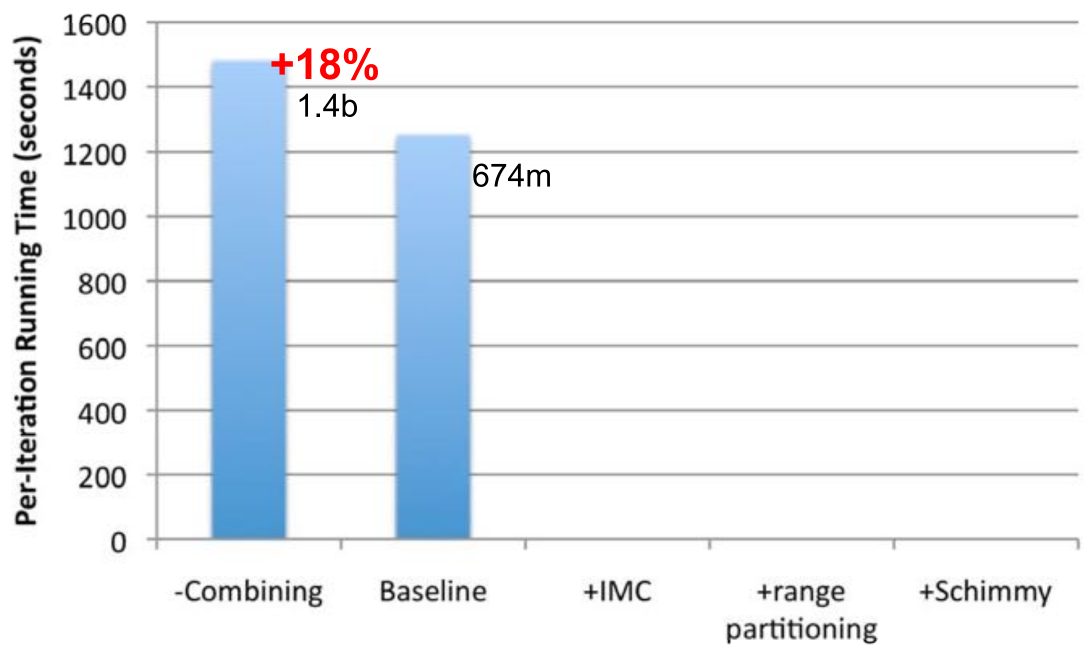
- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3
- Dataset:
  - First English segment of ClueWeb09 collection
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 billion links, 7.0 GB
  - Dataset arranged in crawl order
- Setup:
  - Measured per-iteration running time (5 iterations)
  - 100 partitions

Results



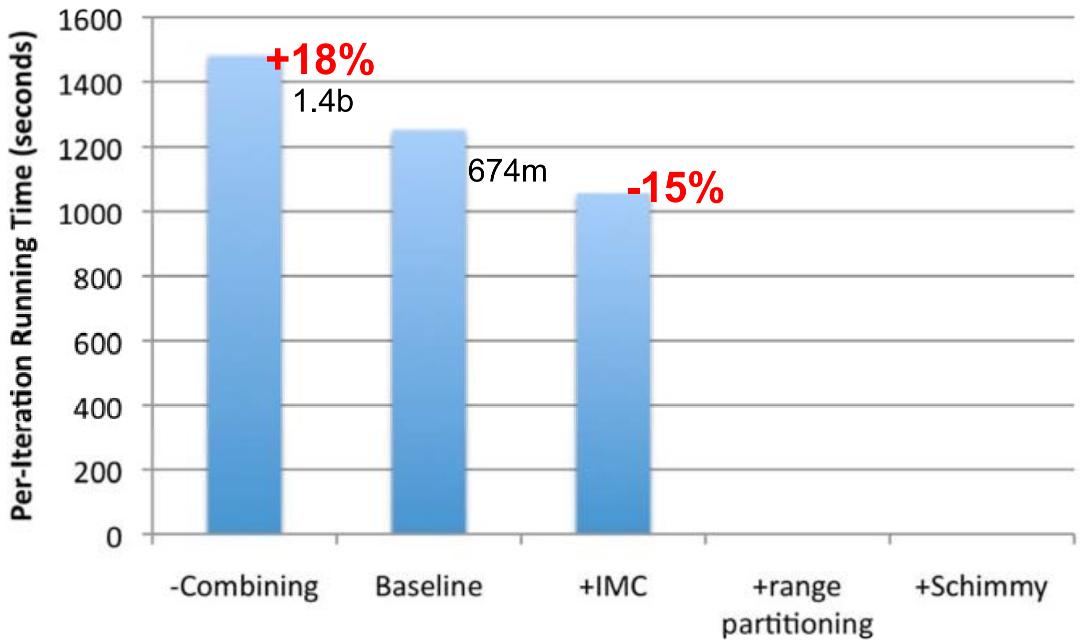
“Best Practices”
Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results

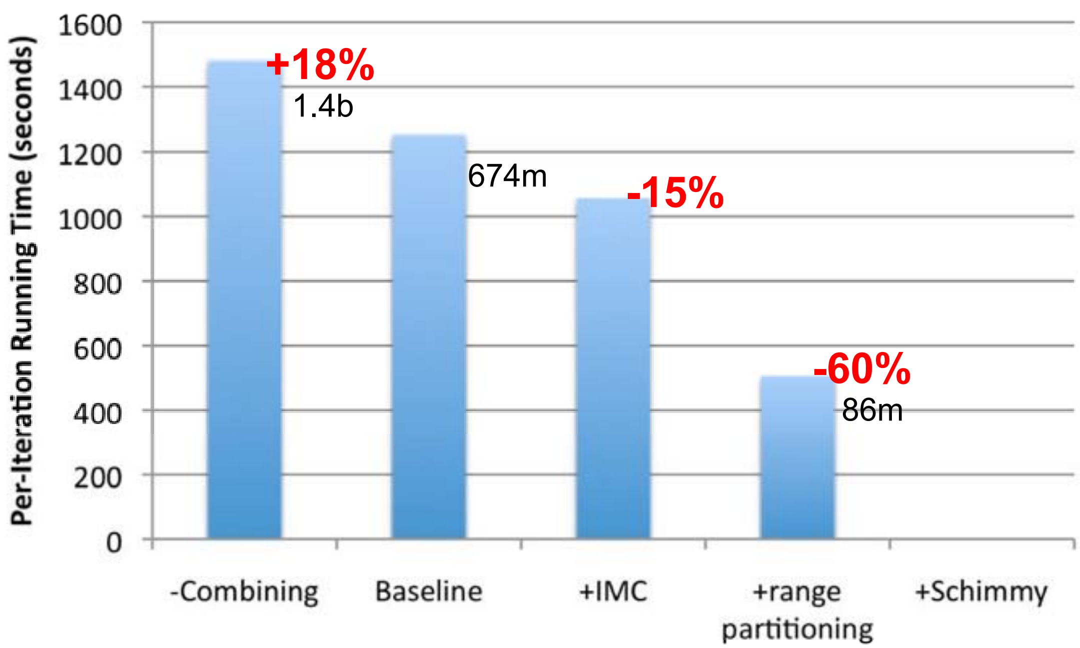


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results

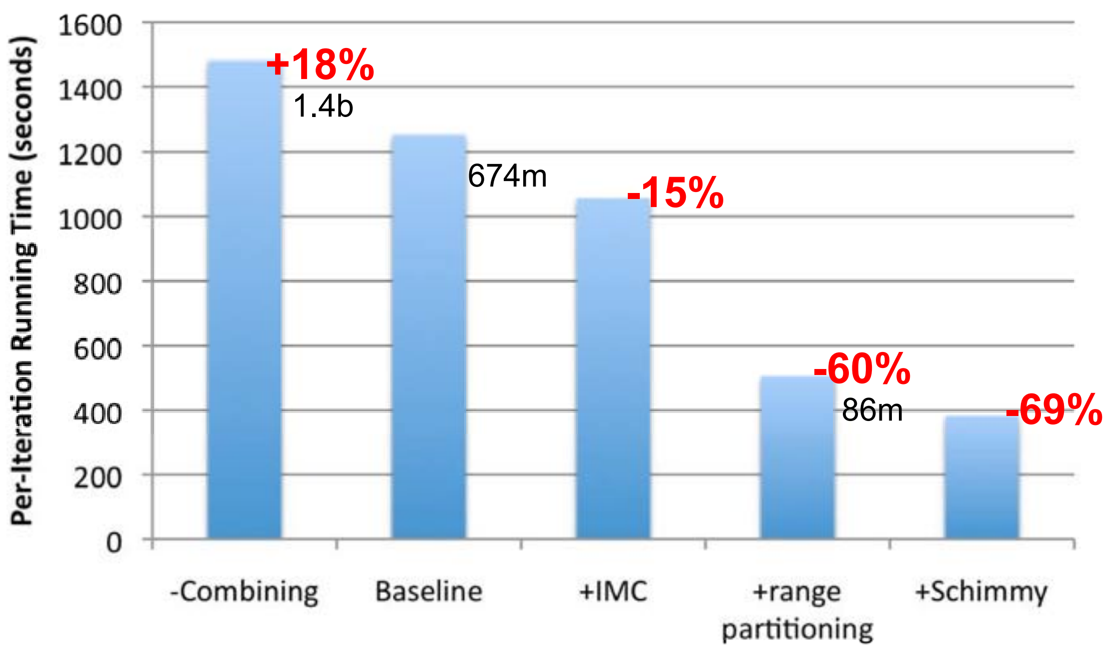

Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results



Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

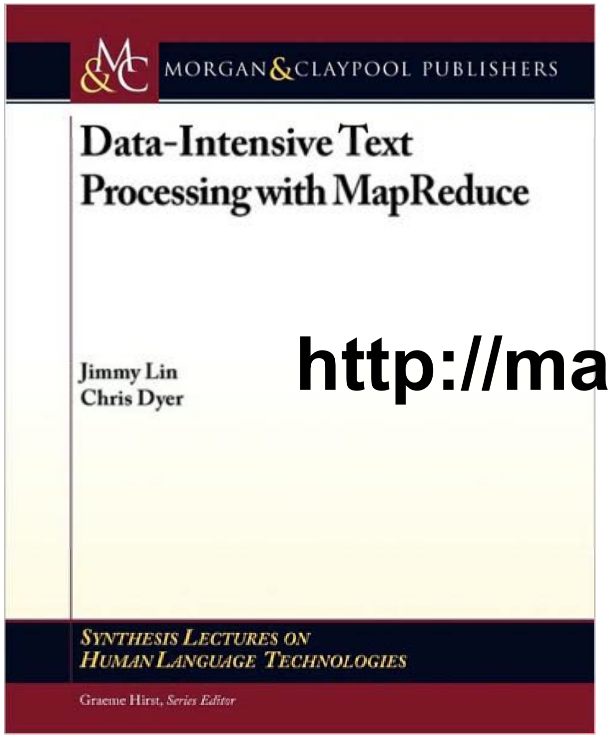
Results


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-69%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Take-Away Messages

- Lots of interesting graph problems!
  - Social network analysis
  - Bioinformatics
- Reducing intermediate data is key
  - Local aggregation
  - Better partitioning
  - Less bookkeeping

Complete details in Jimmy Lin and Michael Schatz. **Design Patterns for Efficient Graph Algorithms in MapReduce.** *Proceedings of the 2010 Workshop on Mining and Learning with Graphs Workshop (MLG-2010)*, July 2010, Washington, D.C.



http://mapreduce.me/

Source code available in Cloud^9
http://cloud9lib.org/

@lintool