Graphs (Part II)

Shannon Quinn

(with thanks to William Cohen and Aapo Kyrola of CMU, and J. Leskovec, A. Rajaraman, and J. Ullman of Stanford University)

Parallel Graph Computation

- Distributed computation and/or multicore parallelism
  - Sometimes confusing. We will talk mostly about **distributed** computation.
- Are classic graph algorithms parallelizable? What about distributed?
  - Depth-first search?
  - Breadth-first search?
  - Priority-queue based traversals (Djikstra's, Prim's algorithms)

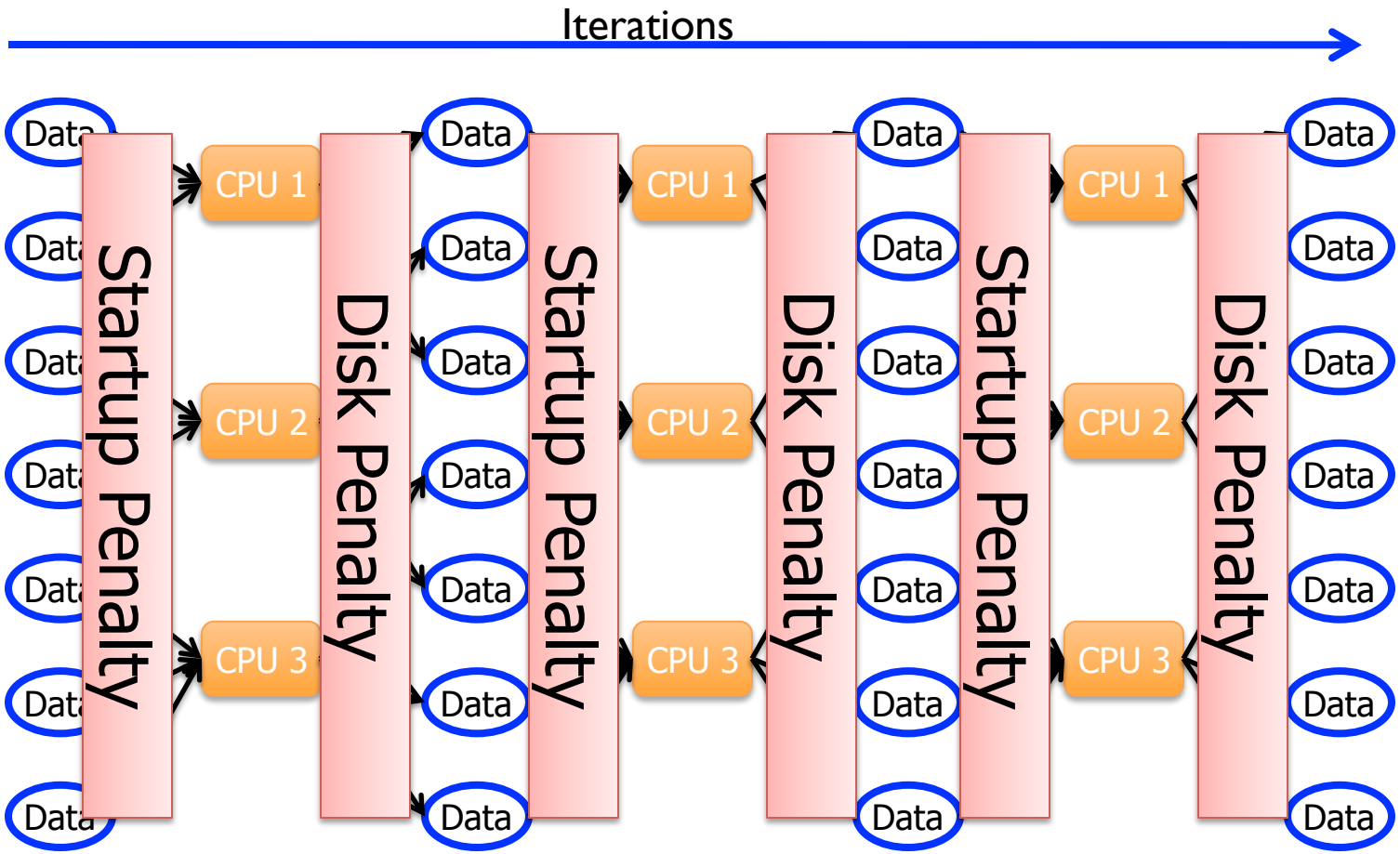
MapReduce for Graphs

- Graph computation almost always **iterative**

- MapReduce ends up shipping the whole graph on each iteration over the network (map->reduce->map->reduce->...)
  - Mappers and reducers are *stateless*

Iterative Computation is Difficult

- System is not optimized for iteration:



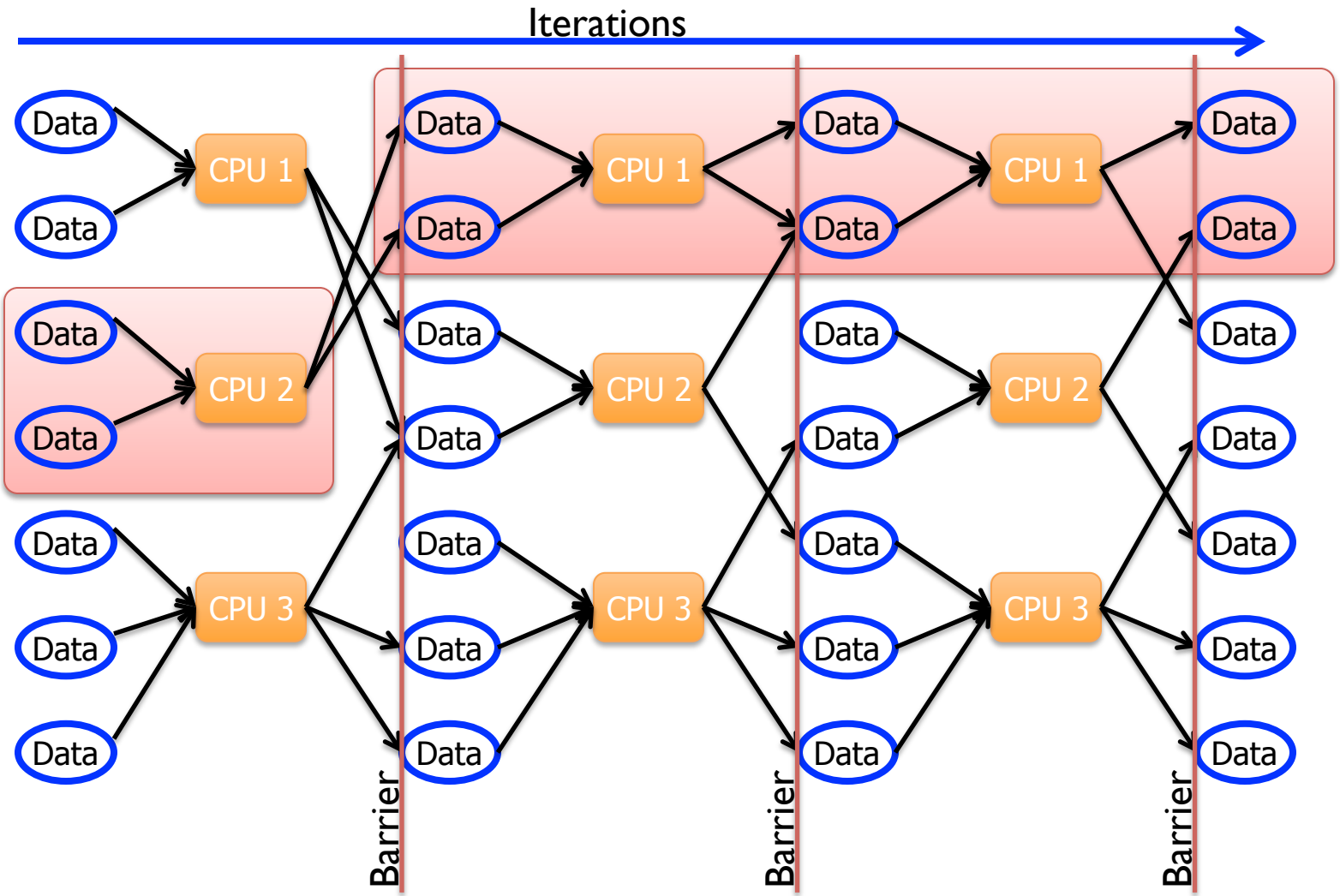
MapReduce and Partitioning

- Map-Reduce splits the keys randomly between mappers/reducers
- But on natural graphs, high-degree vertices (keys) may have million-times more edges than the average

  →Extremely uneven distribution

  →Time of iteration = time of slowest job.

Curse of the Slow Job

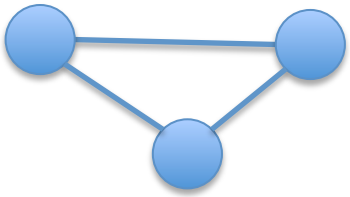


http://www.www2011india.com/proceeding/proceedings/p607.pdf

Map-Reduce is Bulk-Synchronous Parallel

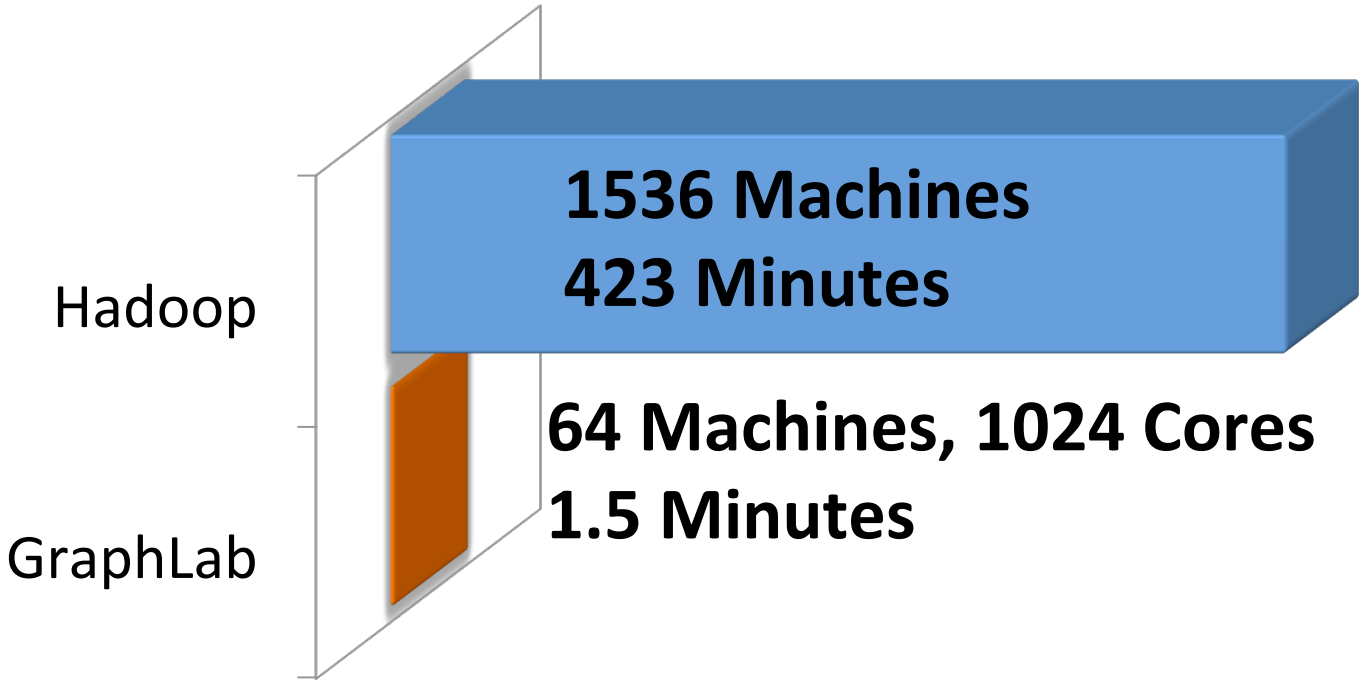
- Bulk-Synchronous Parallel = BSP (Valiant, 80s)
  - Each iteration sees only the values of previous iteration.
  - In linear systems literature: *Jacobi iterations*
- Pros:
  - Simple to program
  - Maximum parallelism
  - Simple fault-tolerance
- Cons:
  - Slower convergence
  - Iteration time = time taken by the slowest node

Triangle Counting in Twitter Graph



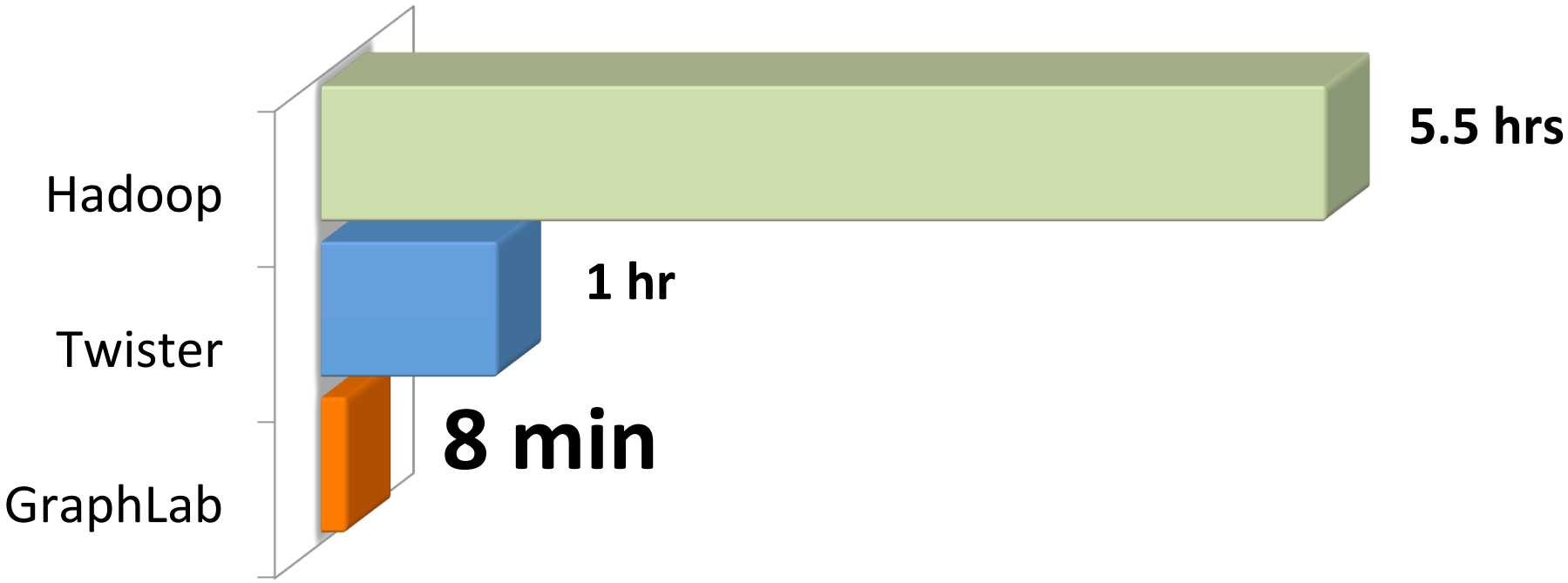
40M Users
1.2B Edges

Total:
34.8 Billion Triangles



Hadoop results from [Suri & Vassilvitskii '11]

PageRank



40M Webpages, 1.4 Billion Links (100 iterations)

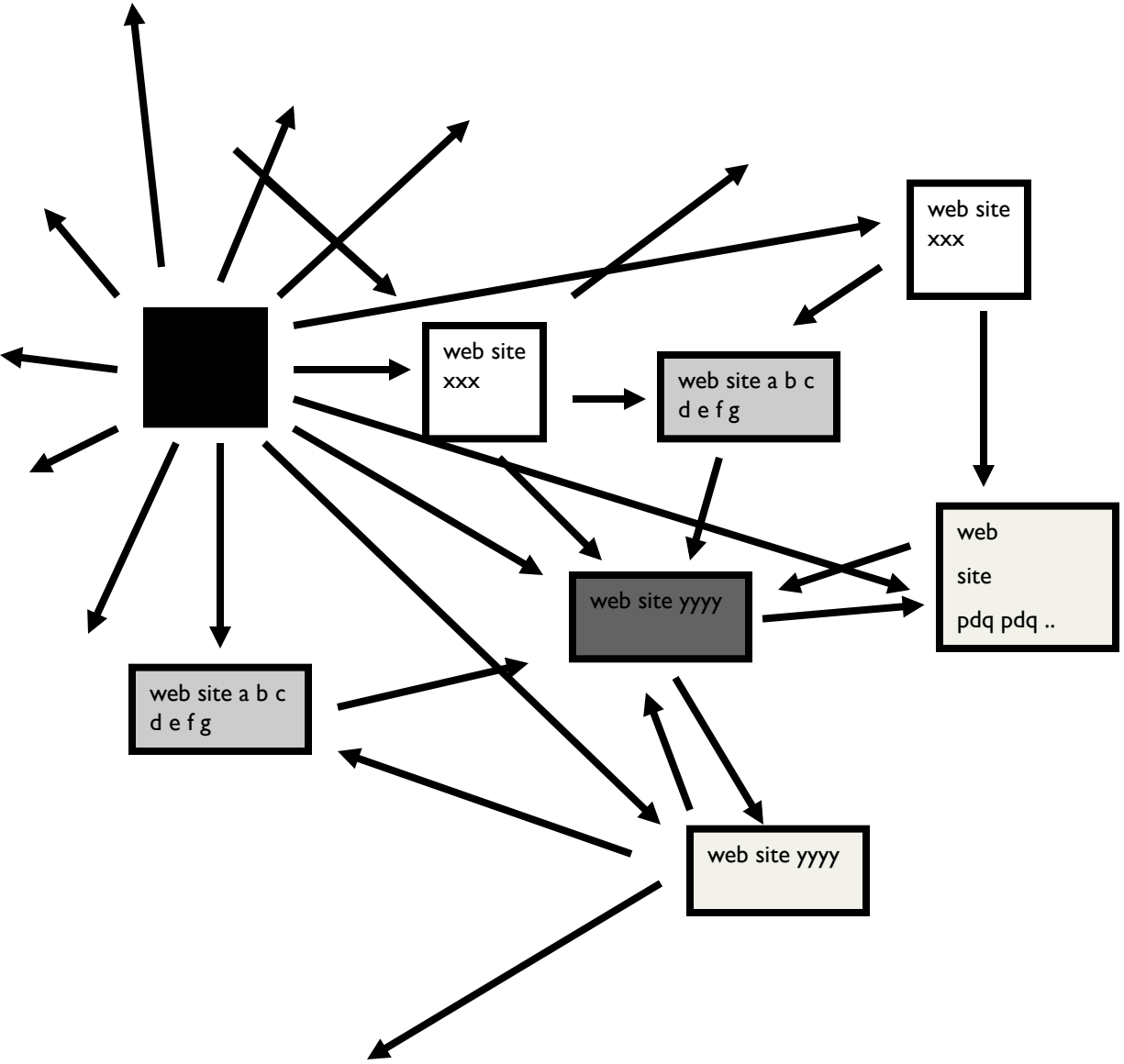
Hadoop results from [Kang et al. '11]
Twister (in-memory MapReduce) [Ekanayake et al. '10]

Graph algorithms

- PageRank implementations
  - in memory
  - streaming, node list in memory
  - streaming, no memory
  - map-reduce

- A little like Naïve Bayes variants
  - data in memory
  - word counts in memory
  - stream-and-sort
  - map-reduce

Google's PageRank



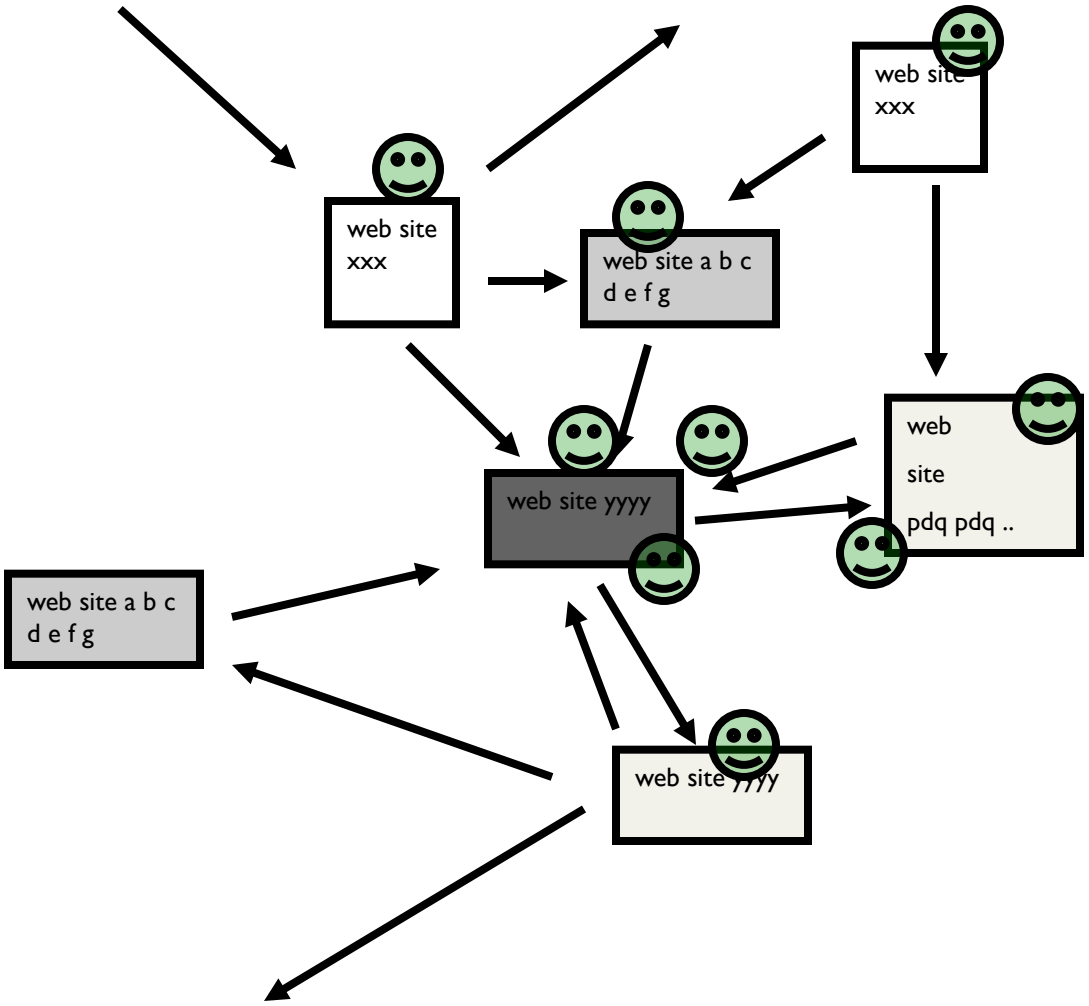
Inlinks are "good" (recommendations)

Inlinks from a "good" site are better than inlinks from a "bad" site

but inlinks from sites with many outlinks are not as "good"...

"Good" and "bad" are relative.

Google's PageRank

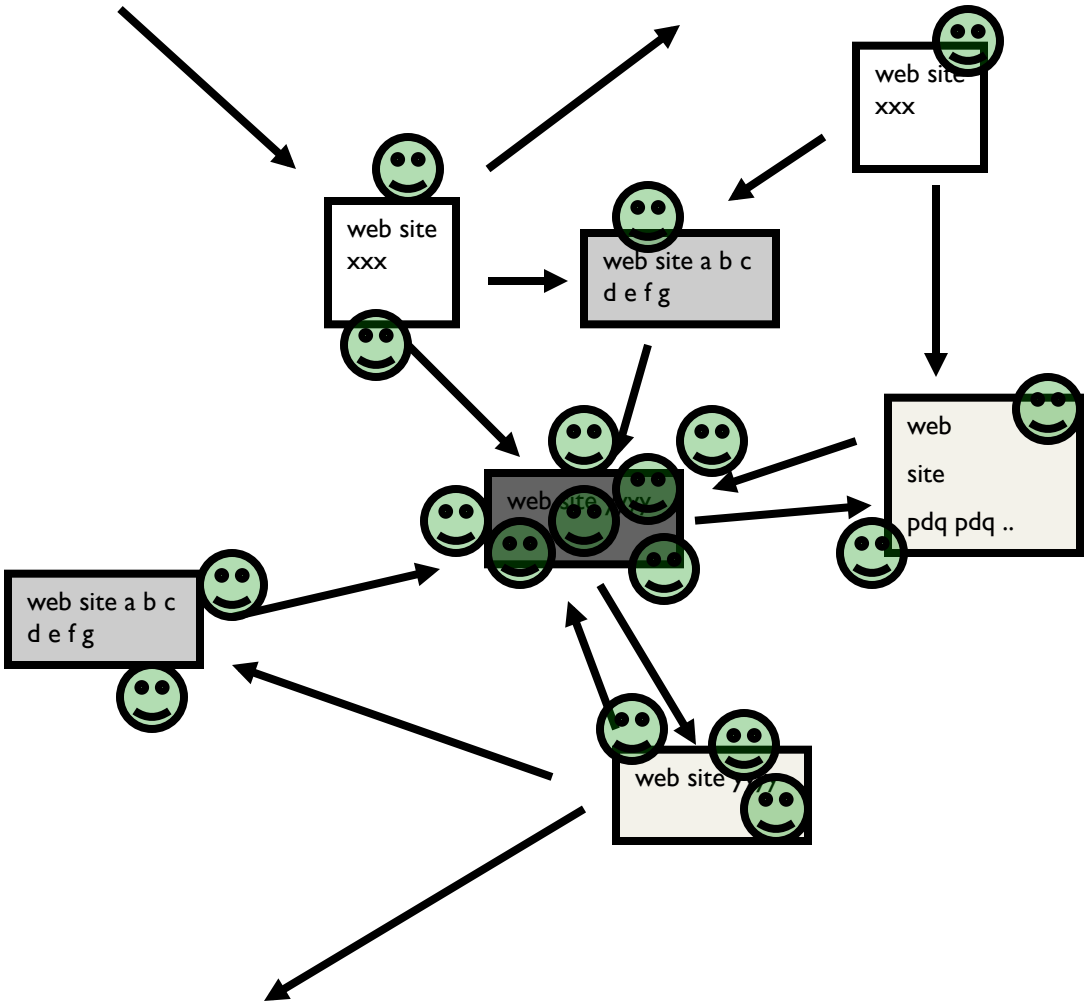


Imagine a "pagehopper" that always either

- follows a random link, or
- jumps to random page

Google's PageRank

(Brin & Page, http://www-db.stanford.edu/~backrub/google.html)



Imagine a "pagehopper" that always either

- follows a random link, or
- jumps to random page

PageRank ranks pages by the amount of time the pagehopper spends on a page:

- or, if there were many pagehoppers, PageRank is the expected "crowd size"

Random Walks

G : a graph

P : transition probability matrix

$$P(u,v) = \begin{cases} \dfrac{1}{d_u} & \text{if } u : v, \\ 0 & \text{otherwise.} \end{cases}$$

d_u := the degree of u.

A lazy walk:

$$W = \frac{I + P}{2}$$

avoids messy "dead ends"….

Random Walks: PageRank

A (bored) surfer

- either surf a random webpage
  with probability α
- or surf a linked webpage
  with probability $1-\alpha$

α : the jumping constant

$$p = \alpha(\tfrac{1}{n}, \tfrac{1}{n}, \ldots., \tfrac{1}{n}) + (1-\alpha)\, pW$$

Random Walks: PageRank

Two equivalent ways to define PageRank $p=pr(\alpha,s)$

(1) $$p = \alpha s + (1-\alpha) pW$$

(2) $$p = \alpha \sum_{t=0}^{\infty} (1-\alpha)^t (sW^t)$$

$s = (\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ ⟹ the (original) PageRank

$s =$ some "seed", e.g., $(1, 0, \ldots, 0)$

⟹ personalized PageRank

Graph = Matrix

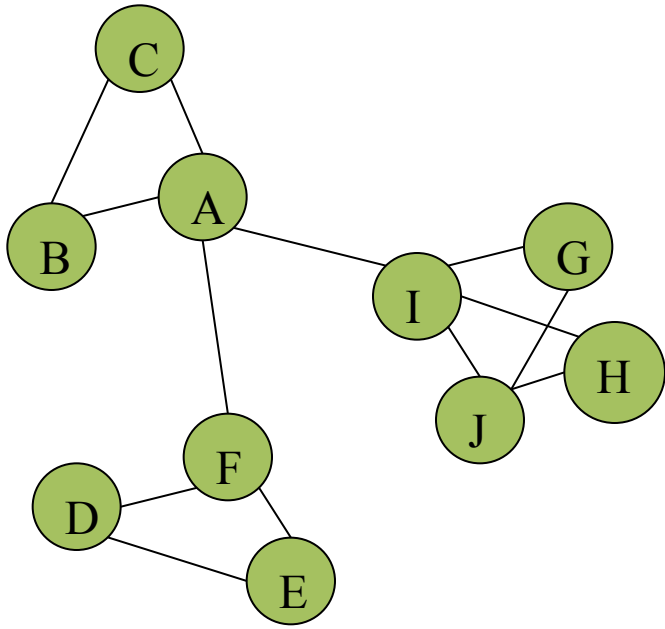
Vector = Node → Weight

M

	A	B	C	D	E	F	G	H	I	J
A	_	I	I			I				
B	I	_	I							
C	I	I	_							
D				_	I	I				
E				I	_	I				
F	I			I	I	_				
G							_		I	I
H								_	I	I
I							I	I	_	I
J							I	I	I	_

v

	A
A	3
B	2
C	3
D	
E	
F	
G	
H	
I	
J	



M

PageRank in Memory

- Let $\mathbf{u} = (1/N, \ldots, 1/N)$
  - dimension = #nodes N
- Let A = adjacency matrix: $[a_{ij}=1 \Leftrightarrow$ i links to j$]$
- Let $W = [w_{ij} = a_{ij}/\text{outdegree}(i)]$
  - w_{ij} is probability of jump from i to j
- Let $\mathbf{v}^0 = (1,1,\ldots,1)$
  - or anything else you want
- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$
    - c is probability of jumping "anywhere randomly"

Streaming PageRank

- Assume we can store **v** but not **W** in memory
- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^{t}$

- Store **A** as a row matrix: each line is
  - i $j_{i,1},\ldots,j_{i,d}$ [the neighbors of i]
- Store **v'** and **v** in memory: **v'** starts out as c**u**
- For each line "i $j_{i,1},\ldots,j_{i,d}$"
  - For each j in $j_{i,1},\ldots,j_{i,d}$
    - **v'**[j] += (1-c)**v[i]**/d

Everything needed for update is right there in row….

Streaming PageRank: with some long rows

- Repeat until converged:
  - Let $\mathbf{v}^{t+1} = c\mathbf{u} + (1\text{-}c)\mathbf{W}\mathbf{v}^t$

- Store **A** as a list of edges: each line is: "i d(i) j"
- Store **v'** and **v** in memory: **v'** starts out as c**u**
- For each line "i d j"
  - **v'**[j] += (1-c)**v[i]**/d

We need to get the degree of *i* and store it locally

Streaming PageRank: preprocessing

- Original encoding is edges (i,j)
- Mapper replaces i,j with i,1
- Reducer is a SumReducer
- Result is pairs (i,d(i))

- Then: join this back with edges (i,j)
- For each i,j pair:
  - send j as a message to node i in the degree table
    - messages always sorted after non-messages
  - the reducer for the degree table sees i,d(i) first
    - then j1, j2, ….
    - can output the key,value pairs with key=i, value=d(i), j

Preprocessing Control Flow: 1

I	J
i1	j1,1
i1	j1,2
…	…
i1	j1,k1
i2	j2,1
…	…
i3	j3,1
…	…

MAP

I	
i1	1
i1	1
…	…
i1	1
i2	1
…	…
i3	1
…	…

SORT

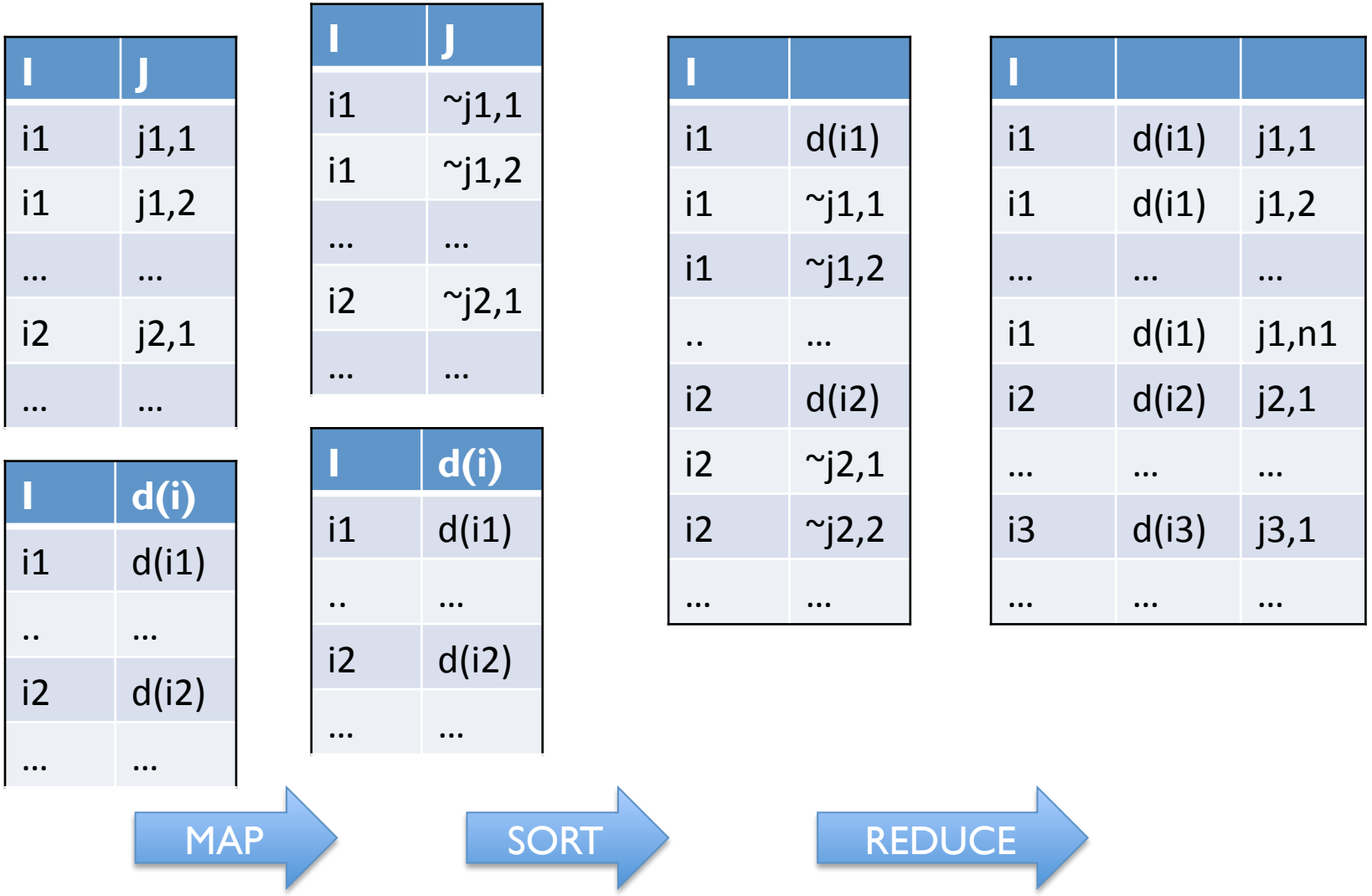
I	
i1	1
i1	1
…	…
i1	1
i2	1
…	…
i3	1
…	…

REDUCE

Summing values

I	d(i)
i1	d(i1)
..	…
i2	d(i2)
…	…
i3	d)i3)
…	…

Preprocessing Control Flow: 2



Control Flow: Streaming PR

I	J
i1	j1,1
i1	j1,2
…	…
i2	j2,1
…	…

I	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

I	d/v
i1	d(i1),v(i1)
i1	~j1,1
i1	~j1,2
..	…
i2	d(i2),v(i2)
i2	~j2,1
i2	~j2,2
…	…

to	delta
i1	c
j1,1	(1-c)v(i1)/d(i1)
…	…
j1,n1	i
i2	c
j2,1	…
…	…
i3	c

I	delta
i1	c
i1	(1-c)v(…)….
i1	(1-c)…
..	…
i2	c
i2	(1-c)…
i2	….
…	…

MAP → SORT

copy or convert to messages

REDUCE → MAP → SORT

send "pageRank updates " to outlinks

Control Flow: Streaming PR

to	delta
i1	c
j1,1	(1-c)v(i1)/d(i1)
…	…
j1,n1	i
i2	c
j2,1	…
…	…
i3	c

l	delta
i1	c
i1	(1-c)v(…)….
i1	(1-c)…
..	…
i2	c
i2	(1-c)…
i2	….
…	…

l	v’
i1	~v’(i1)
i2	~v’(i2)
…	…

l	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

l	d/v
i1	d(i1),v’(i1)
i2	d(i2),v’(i2)
…	…

Summing values

Replace v with v’

Control Flow: Streaming PR

I	J
i1	j1,1
i1	j1,2
…	…
i2	j2,1
…	…

I	d/v
i1	d(i1),v(i1)
i2	d(i2),v(i2)
…	…

copy or convert to messages

and back around for next iteration….

PageRank in MapReduce

```
1: class Mapper
2:     method Map(id n, vertex N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(id n, vertex N)
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(id m, value p)

1: class Reducer
2:     method Reduce(id m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ [p1, p2, ...] do
5:             if IsVertex(p) then
6:                 M ← p
7:             else
8:                 s ← s + p
9:         M.PageRank ← s
10:        Emit(id m, vertex M)
```

More on graph algorithms

- PageRank is a one simple example of a graph algorithm
  - but an important one
  - *personalized* PageRank (aka "random walk with restart") is an important operation in machine learning/data analysis settings
- PageRank is typical in some ways
  - Trivial when graph fits in memory
  - Easy when node weights fit in memory
  - More complex to do with constant memory
  - A major expense is scanning through the graph many times
    - … same as with SGD/Logistic regression
    - disk-based streaming is *much* more expensive than memory-based approaches
- Locality of access is very important!
    - gains if you can pre-cluster the graph even approximately
    - avoid sending messages across the network – keep them local

Design Patterns for Efficient Graph Algorithms in MapReduce

Jimmy Lin and Michael Schatz
University of Maryland, College Park
{jimmylin,mschatz}@umd.edu

Machine Learning in Graphs - 2010

Some ideas

- Combiners are helpful
  - Store outgoing *incrementVBy* messages and aggregate them
  - This is great for high indegree pages
- Hadoop's combiners are suboptimal
  - Messages get emitted before being combined
  - Hadoop makes weak guarantees about combiner usage

This slide again!

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

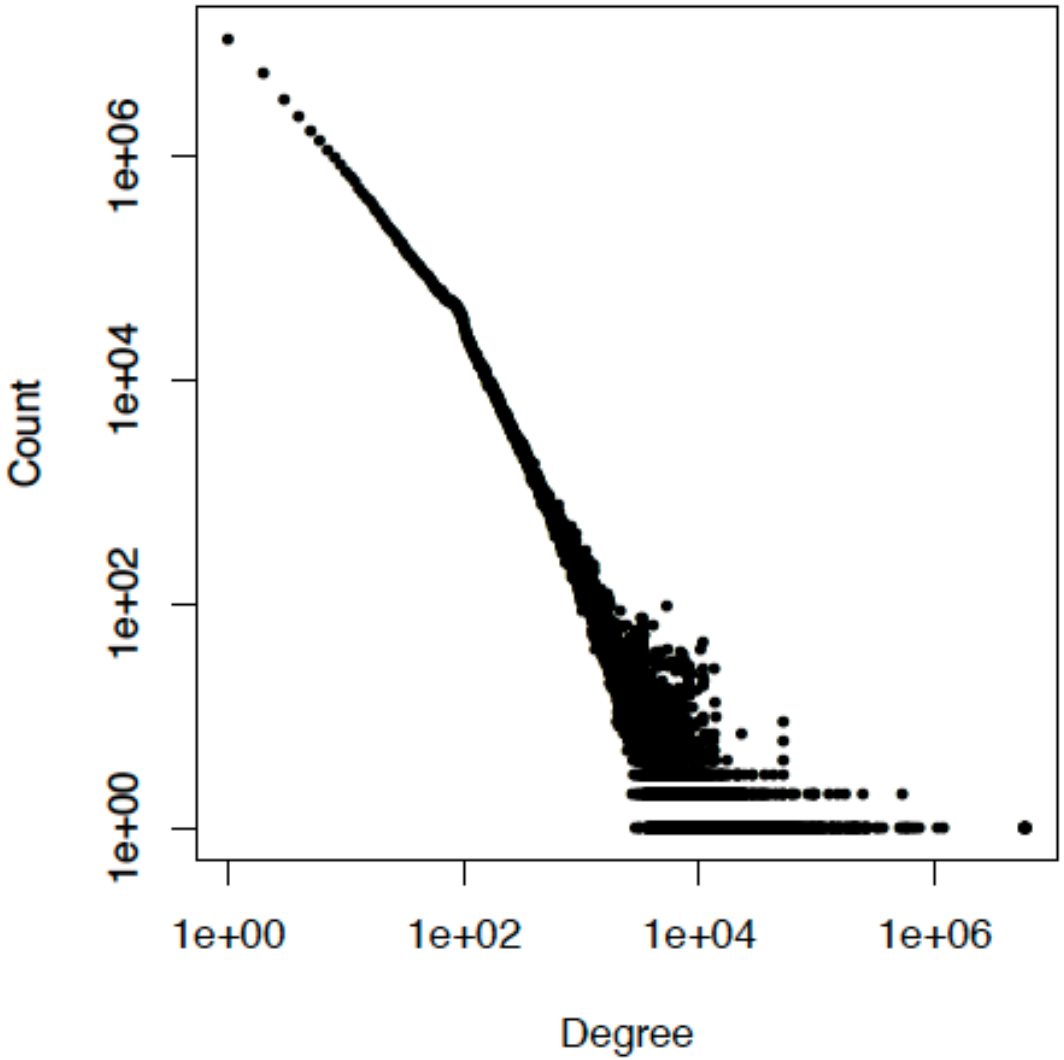


Figure 6: ClueWeb09 in-degree distribution. Most pages have relatively few predecessors, but a significant fraction have more than 100, and 15 pages have more than 1 million.

```
1: class Combiner
2:     method Combine(id m, [p_1, p_2, ...])
3:         M ← ∅
4:         for all p ∈ [p_1, p_2, ...] do
5:             if IsVertex(p) then
6:                 Emit(id m, vertex p)
7:             else
8:                 s ← s + p
9:         Emit(nid m, value p)
```

Figure 3: Combiner pseudo-code for PageRank in MapReduce. The combiner aggregates partial PageRank contributions by destination vertex and passes the graph structure along.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(id n, vertex N)
5:         p ← N.PageRank/|N.AdjacencyList|
6:         Emit(id n, vertex N)
7:         for all id m ∈ N.AdjacencyList do
8:             H{m} ← H{m} + p
9:     method Close
10:        for all id n ∈ H do
11:            Emit(id n, value H{n})
```

Figure 4: Mapper pseudo-code for PageRank in MapReduce that implements the in-mapper combining design pattern.

Some ideas

- Most hyperlinks are within a domain
  - If we keep domains on the same machine this will mean more messages are local
  - To do this, build a custom partitioner that knows about the domain of each nodeId and keeps nodes on the same domain together
  - Assign node id's so that nodes in the same domain are together – partition node ids by range
  - Change Hadoop's *Partitioner* for this

Some ideas

- Repeatedly shuffling the graph is expensive
  - We should separate the messages about the graph structure (fixed over time) from messages about pageRank weights (variable)
  - compute and distribute the edges *once*
  - read them in incrementally in the reducer
    - not easy to do in Hadoop!
  - call this the "Schimmy" pattern

Schimmy

In the initialization API hook, the reducer opens the file containing the graph partition corresponding to the intermediate keys that are to be processed by the reducer. As the reducer is processing messages passed to each vertex in the REDUCE method, it advances the file stream in the graph structure until the corresponding vertex's structure is found. Once the reduce computation is complete (a simple sum), the vertex's state is updated with the revised PageRank value and written back to disk. The partitioner ensures consistent partitioning of the graph structure from iteration to iteration.

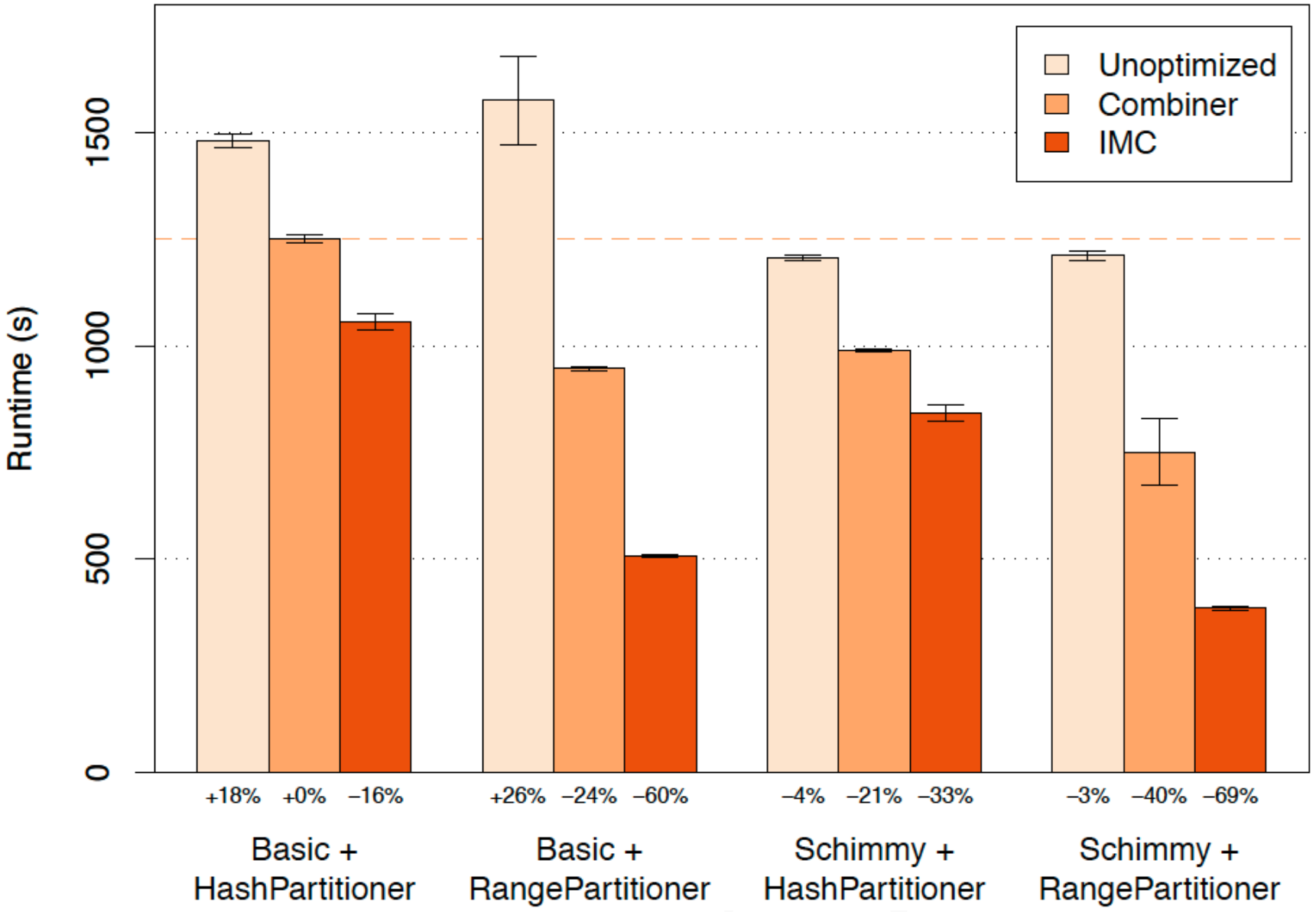
Relies on fact that keys are sorted, and sorts the graph input the same way…..

Schimmy

```
1: class Reducer
2:     method Initialize
3:         P.OpenGraphPartition()
4:     method Reduce(id m, [p1, p2, ...])
5:         repeat
6:             (id n, vertex N) ← P.Read()
7:             if n ≠ m then
8:                 Emit(id n, vertex N)
9:         until n = m
10:        for all p ∈ values [p1, p2, ...] do
11:            s ← s + p
12:        N.PageRank ← s
13:        Emit(id n, vertex N)
```

Figure 5: Reducer pseudo-code for PageRank in MapReduce using the schimmy design pattern to avoid shuffling the graph structure.

Results


Runtime (s)
0
500
1000
1500
Unoptimized
Combiner
IMC
+18%
+0%
−16%
+26%
−24%
−60%
−4%
−21%
−33%
−3%
−40%
−69%
Basic + HashPartitioner
Basic + RangePartitioner
Schimmy + HashPartitioner
Schimmy + RangePartitioner

More details at…

- Overlapping Community Detection at Scale: A Nonnegative Matrix Factorization Approach by J. Yang, J. Leskovec. *ACM International Conference on Web Search and Data Mining (WSDM)*, 2013.

- Detecting Cohesive and 2-mode Communities in Directed and Undirected Networks by J. Yang, J. McAuley, J. Leskovec. *ACM International Conference on Web Search and Data Mining (WSDM)*, 2014.

- Community Detection in Networks with Node Attributes by J. Yang, J. McAuley, J. Leskovec. *IEEE International Conference On Data Mining (ICDM)*, 2013.

Resources

- Stanford SNAP datasets:
  - http://snap.stanford.edu/data/index.html
- ClueWeb (CMU):
  - http://lemurproject.org/clueweb09/
- Univ. of Milan's repository:
  - http://law.di.unimi.it/datasets.php