IEMS5709
Advanced Topics Information Processing: Big Data Systems and Information Processing
Spring 2016

Analyzing Massive Graphs and Graph-based Big Learning Platforms

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - "Data-Intensive Information Processing Applications," by Jimmy Lin, University of Maryland.

  - CS246 Mining Massive Data-sets, by Jure Leskovec, Stanford University.
  - Introduction to Advanced Computing Platform for Data Analysis, by Ruoming Jin, Kent University.
  - G. Malewicz et al, "Pregel: A System for Large-Scale Graph Processing," ACM SIGMOD 2010, http://www.slideshare.net/shatteredNirvana/pregel-a-system-for-largescale-graph-processing
  - Carlos Guestrin et al, "GraphLab 2: Parallel Machine Learning for Large-Scale Natural Graphs," NIPS Big Learning Workshop 2011, http://www.select.cs.cmu.edu/code/graphlab/presentations/nips-biglearn-2011.pptx
  - Yucheng Low, Joseph Gonzalez et al, "GraphLab: A New Framework for Parallel Machine Learning," http://select.cs.cmu.edu/code/graphlab/uai2010_graphlab.pptx
  - Joseph Gonzalez et al, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," talk for OSDI 2012

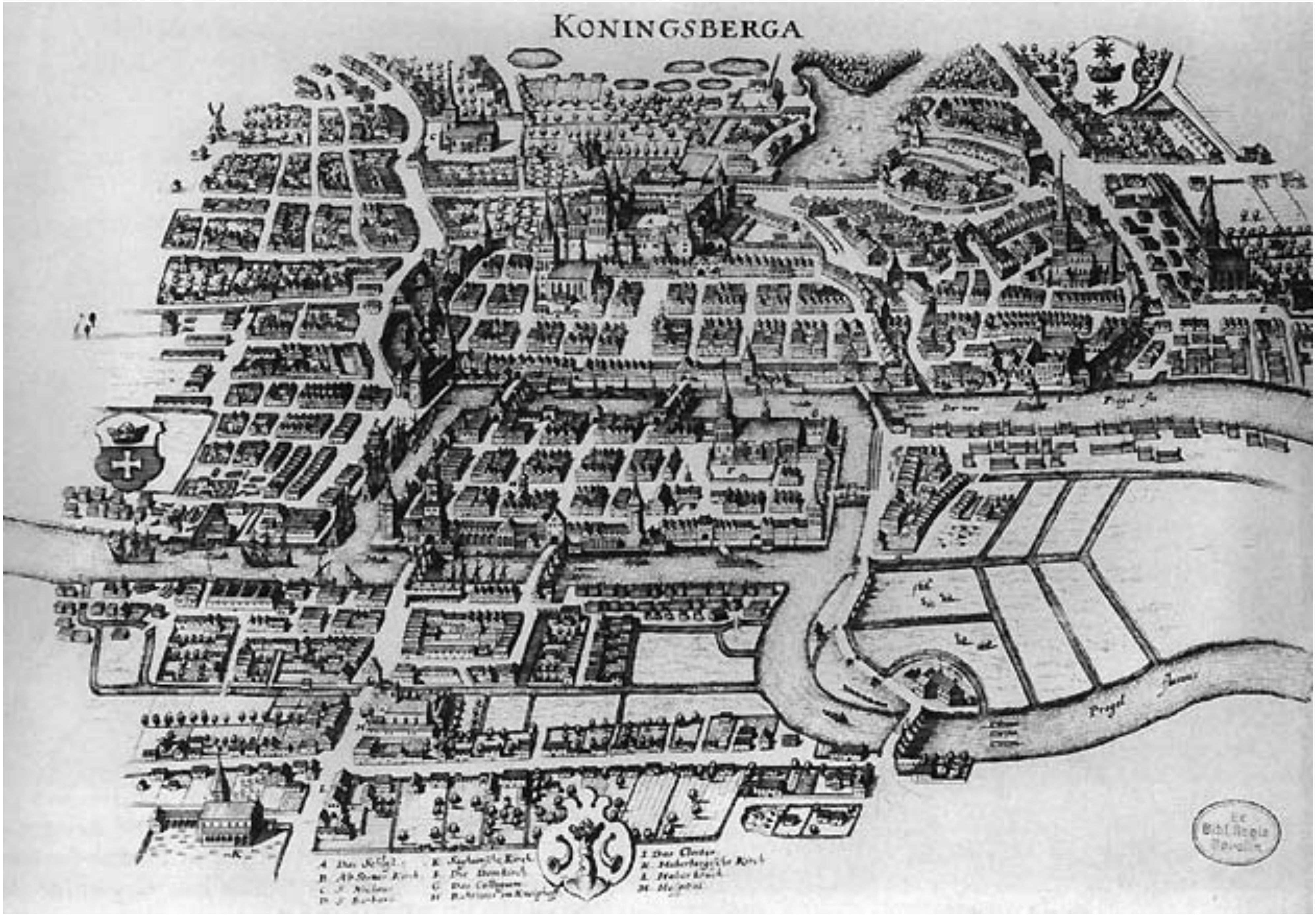
-

Roadmap

- Graph problems and representations
- PageRank
- Emerging Parallel Processing Platforms for Graph-based Big Learning
  - Problems of MapReduce for Graph-based Processing/ MLDM
  - Pregel
  - GraphLab

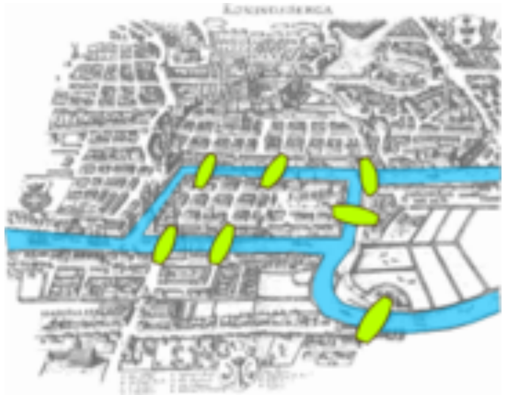
What's a graph?

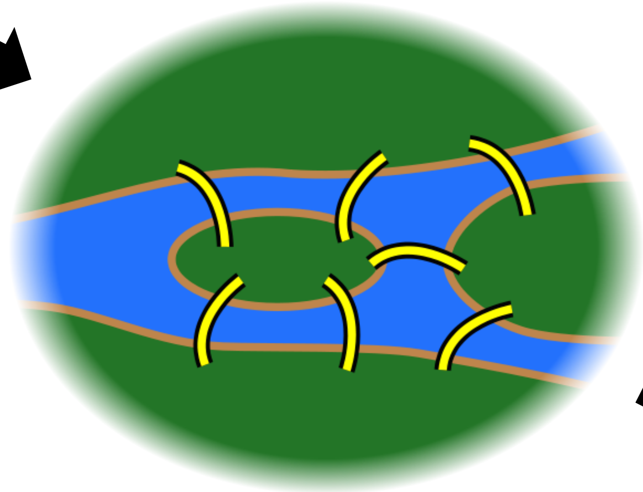
- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink structure of the Web
  - Physical structure of computers on the Internet
  - Interstate highway system
  - Social networks

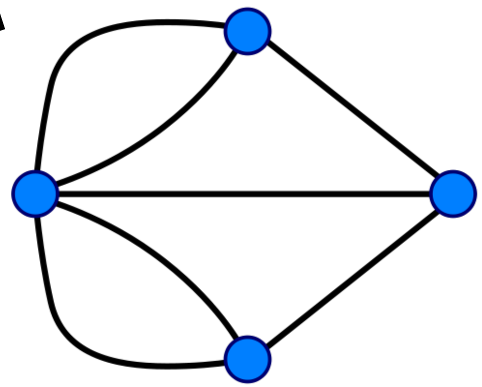


Source: Wikipedia (Königsberg)

The Seven Bridges of Königsberg Problem by Euler





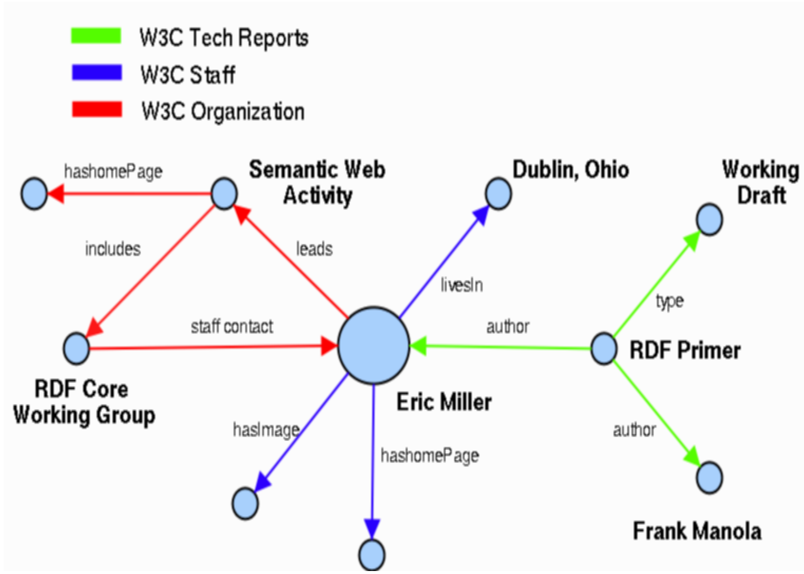


Source: Wikipedia (Königsberg)

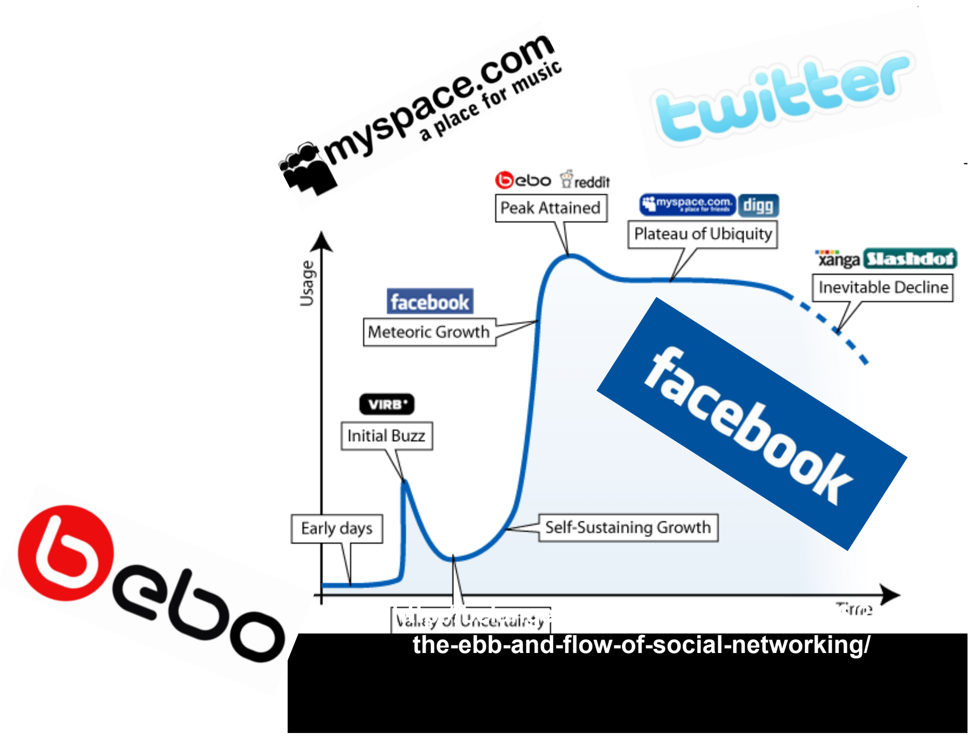
Some Graph Problems

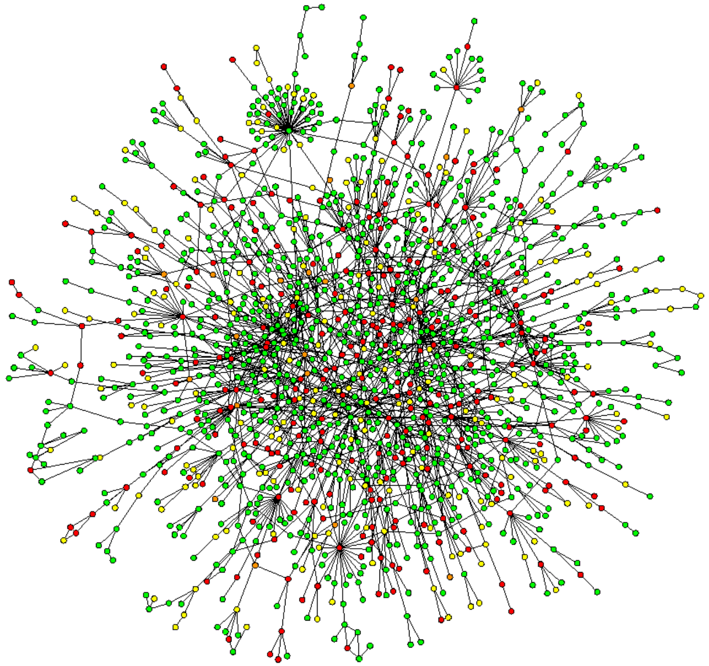
- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify “special” nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank

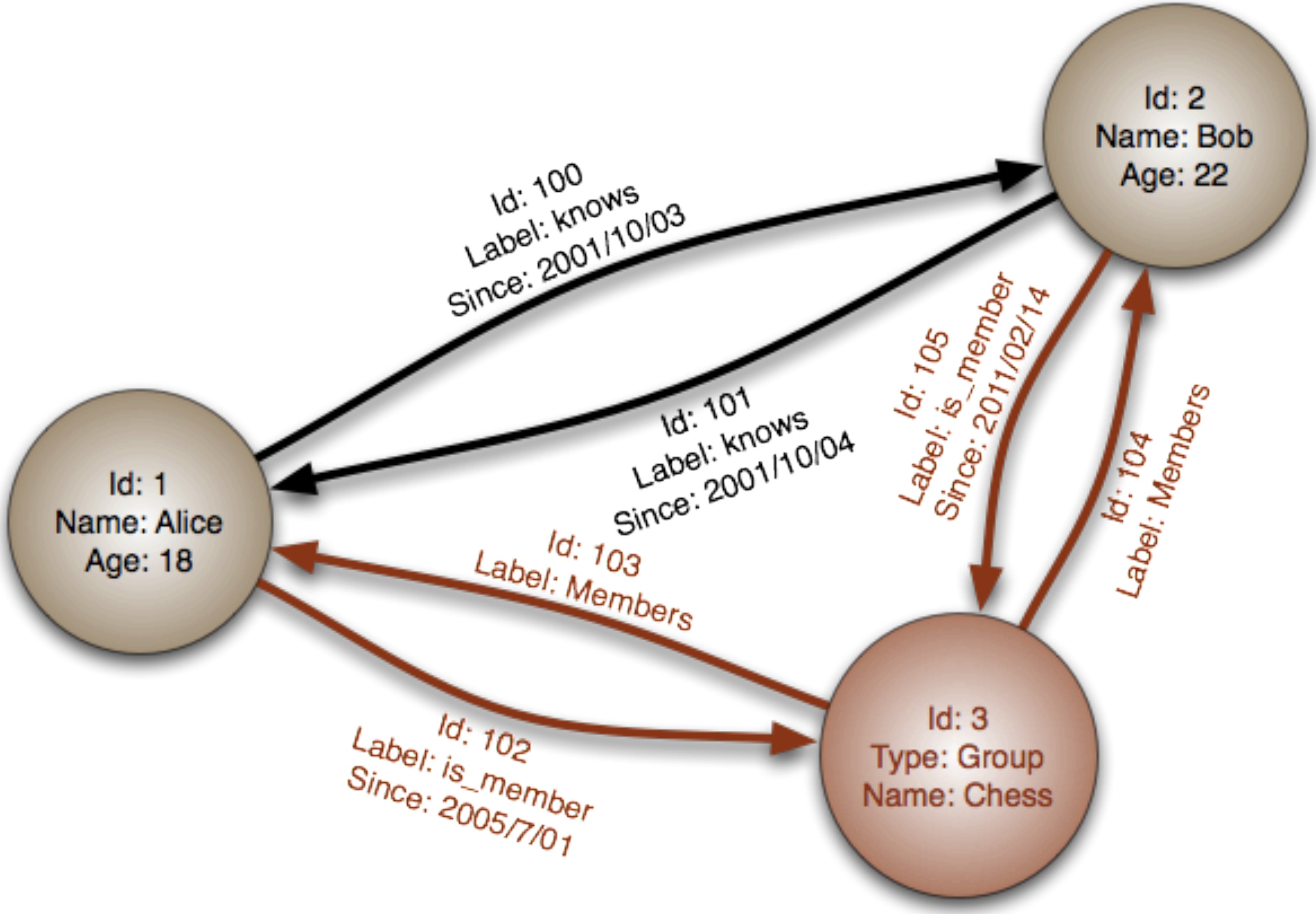
Ubiquitous Network (Graph) Data



Semantic Search, Guha et. al., WWW'03






Id: 1
Name: Alice
Age: 18
Id: 2
Name: Bob
Age: 22
Id: 3
Type: Group
Name: Chess
Id: 100
Label: knows
Since: 2001/10/03
Id: 101
Label: knows
Since: 2001/10/04
Id: 102
Label: is_member
Since: 2005/7/01
Id: 103
Label: Members
Id: 104
Label: Members
Id: 105
Label: is_member
Since: 2011/02/14

Graph (and Relational) Analytics

- General Graph
  - Count the number of nodes whose degree is equal to 5
  - Find the diameter of the graphs
- Web Graph
  - Rank each webpage in the webgraph or each user in the twitter graph using PageRank, or other centrality measure
- Transportation Network
  - Return the shortest or cheapest flight/road from one city to another
- Social Network
  - Determine whether there is a path less than 4 steps which connects two users in a social network
- Financial Network
  - Find the path connecting two suspicious transactions;
- Temporal Network
  - Compute the number of computers who were affected by a particular computer virus in three days, thirty days since its discovery

Challenge in Dealing with Graph Data

- Flat Files
  - No Query Support
- RDBMS
  - Can Store the Graph
  - Limited Support for Graph Query
    - Connect-By (Oracle)
    - Common Table Expressions (CTEs) (Microsoft)
    - Temporal Table

Native Graph Databases

- Emerging Field - http://en.wikipedia.org/wiki/Graph_database
- Storage and Basic Operators
  - Neo4j (an open source graph database)
  - InfiniteGraph
  - VertexDB

Graphs Algorithms and Graph-based Parallel Processing

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Design Challenges
  - Very little computation work required per vertex.
  - Changing degree of parallelism over the course of execution.
- Generic recipe:
  - Represent graphs in some form of data structure, e.g. adjacency lists
  - Perform local computations in each vertex (node)
  - Pass along partial results via outlinks to destination vertices
  - Perform aggregation in each destination vertex (node) after receiving information from inlinks of a node
  - Iterate until convergence

Representing Graphs

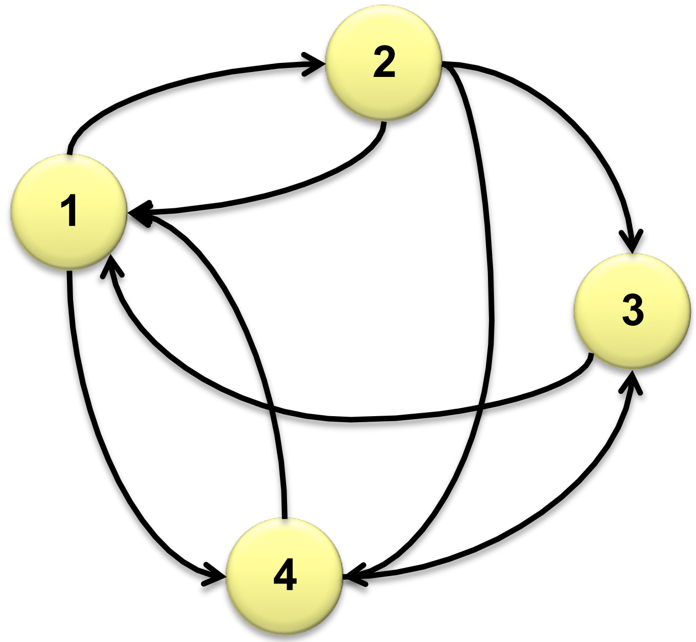
- $G = (V, E)$
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an n x n square matrix M

- $n = |V|$
- $M_{ij} = 1$ means a link from node i to j

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



Adjacency Matrices: Critique

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

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

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

The PageRank Algorithm

Random Walks Over the Web

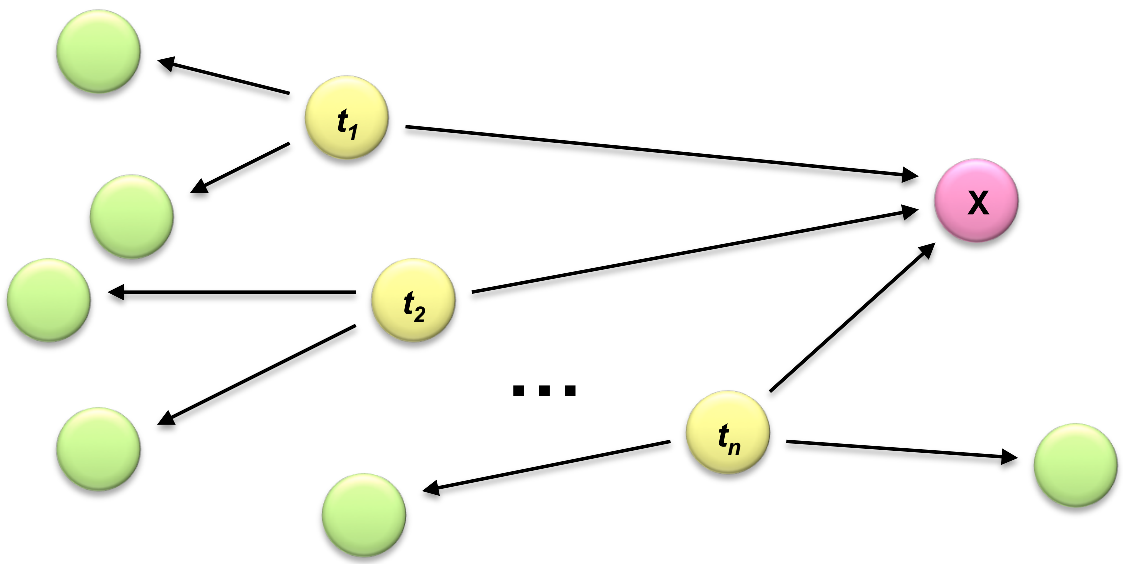
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  - Correspondence to human intuition?
  - One of thousands of features used in web search
  - Note: query-independent

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

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

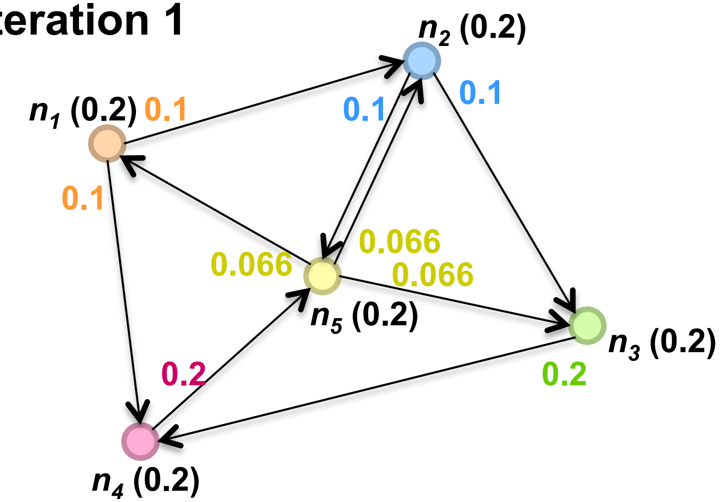


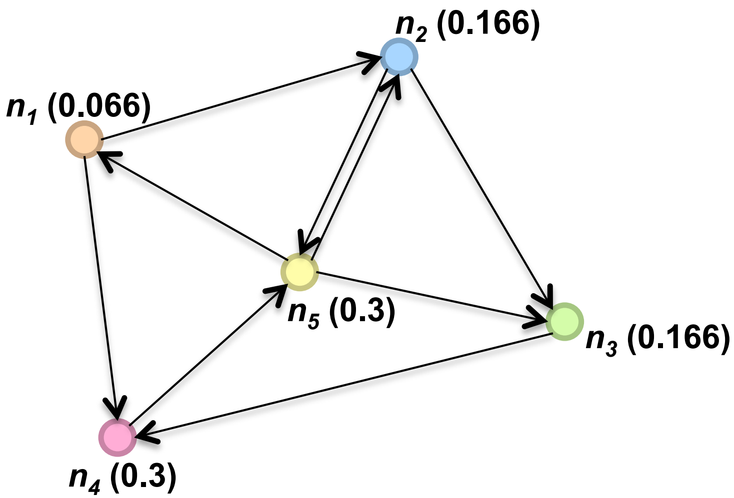
Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values
  - Each page distributes PR_i "credit" to all pages it links to
  - Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

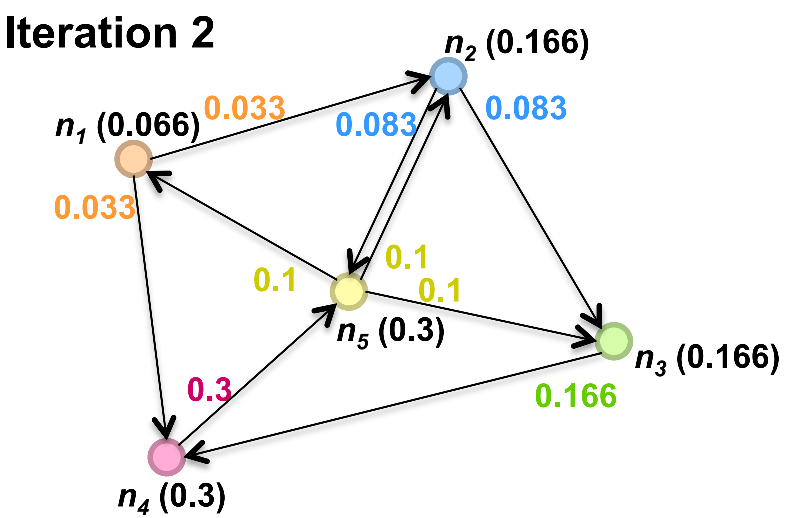
Sample PageRank Iteration (1)

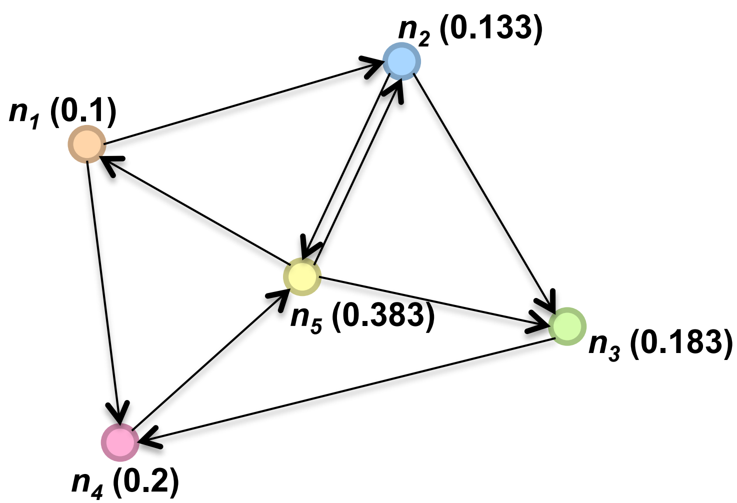
Iteration 1





Sample PageRank Iteration (2)

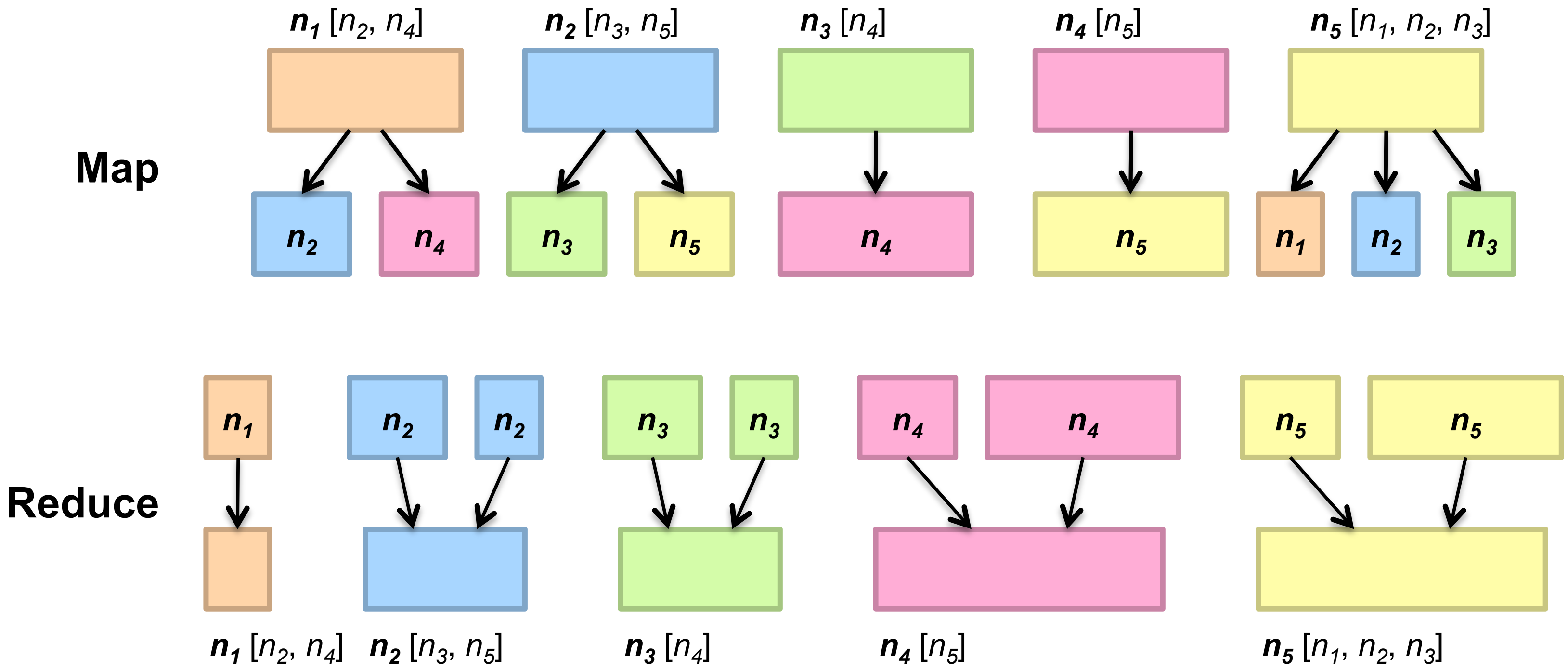




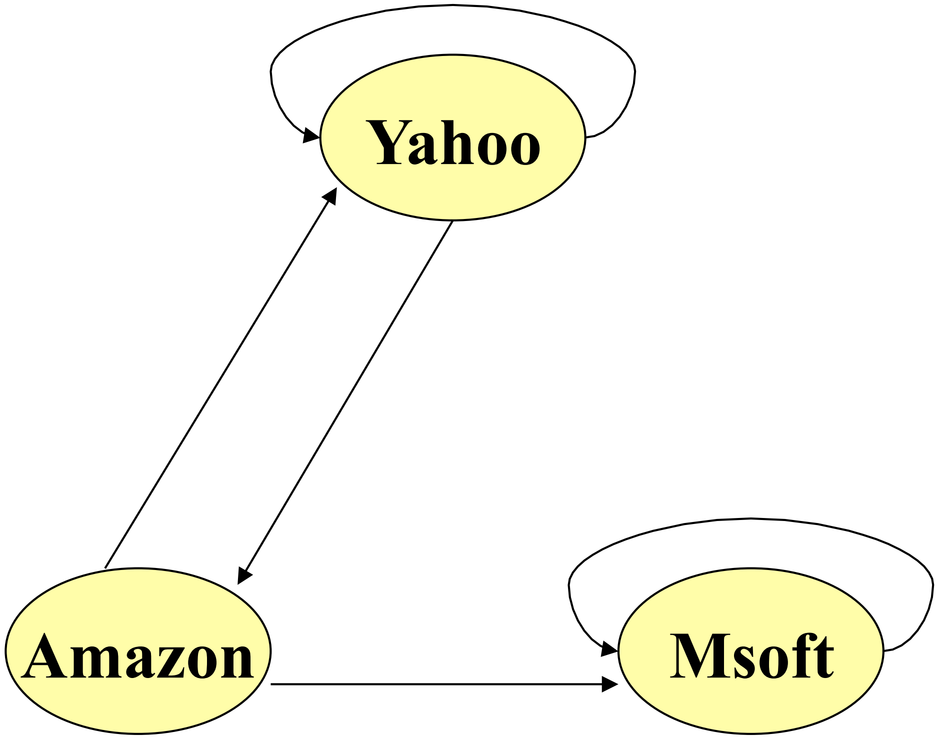
PageRank Pseudo-Code

```
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)
```

PageRank in MapReduce



Problems of Dead-ends (Traps) in a Graph



In the above example, Msoft is a Dead-end (Trap) for the Random Walker

Google’s Solution to Dead-ends (Traps) in a Graph when comparing PageRank

Problem:

PageRank “Credits” received by Dead-end nodes cannot be distributed to further to other Nodes

- The sum of the PageRank Credits over the entire graph will eventually be absorbed by all those few Dead-end Nodes

Solution:

- “Tax” each page a fixed percentage at each iteration.
- Add the same constant to all pages.
- Models a random walk with a fixed probability of going to a random place next.

Complete PageRank

- Two additional complexities
  - What is the proper treatment of dangling nodes (dead-ends) ?
  - How do we factor in the random jump factor?
- Solution:
  - Second pass to redistribute "missing PageRank mass" and account for random jumps

$$p' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{m}{|G|} + p\right)$$

  - *p* is PageRank value from before, *p'* is updated PageRank value
  - |G| is the number of nodes in the graph
  - *m* is the missing PageRank mass

PageRank Convergence

- Alternative convergence criteria
  - Iterate until PageRank values don't change
  - Iterate until PageRank rankings don't change
  - Fixed number of iterations
- Convergence for web graphs?

Some Problems with PageRank

- **Measures generic popularity of a page**
  - Biased against topic-specific authorities
  - **Solution:** Topic-Specific PageRank – biased towards specific restarting sites/points
- **Uses a single measure of importance**
  - Other models e.g., hubs-and-authorities
  - **Solution:** Hubs-and-Authorities: HITS from Cornell,
    - Each webpage has 2 scores:

      1. An Expert Score measuring the quality of the content of pages it points to/recommend ;

      2. An Authority Score measuring the quality of its content ;
- **Susceptible to Link spam**
  - Artificial link topographies created in order to boost page rank
  - **Solution:** TrustRank - biased to use trustworthy sites, e.g. .edu, .mil, .gov sites as restart points

Beyond PageRank

- Link structure is important for web search
  - PageRank is one of many link-based features: HITS, SALSA, etc.
  - One of many thousands of features used in ranking…
- Adversarial nature of web search
  - Link spamming
  - Spider traps
  - Keyword stuffing
  - …

Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies

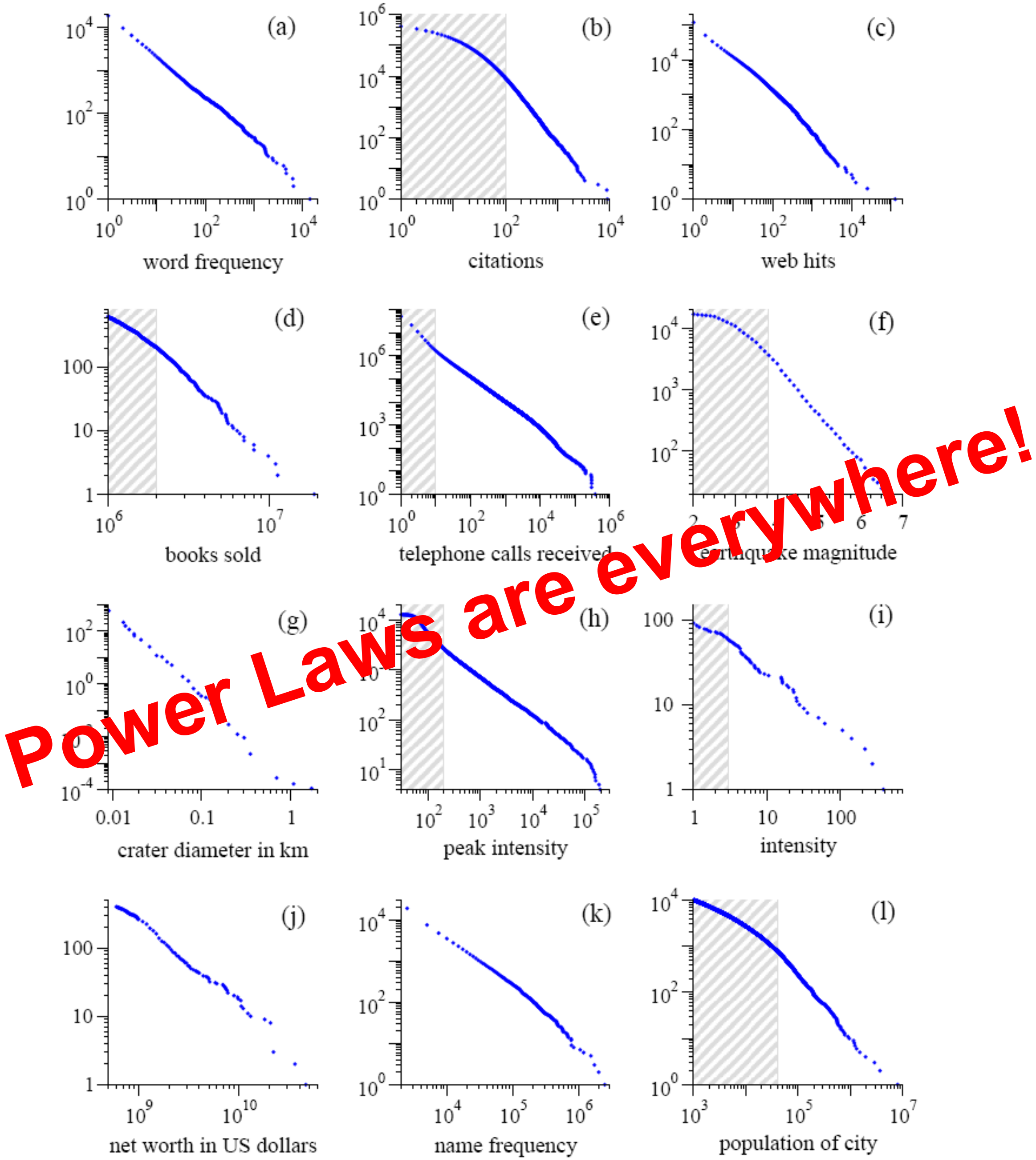


Figure from: Newman, M. E. J. (2005) "Power laws, Pareto distributions and Zipf's law." Contemporary Physics 46:323–351.

Graph analytics industry in practice

- Graph data in many industries
- Graph analytics are powerful and can bring great business values/insights
- Graph analytics not utilized enough in small and medium sized enterprises due to lack of available platforms/tools (except leading tech companies which have high caliber in house engineering teams and resources)
- Luckily, this is changing fast with the new Graph-based Parallel/Big Learning Platforms like GraphLab

Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!

Data-Parall

Is there more to Machine Learning

?

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Embarrassingly Parallel Tasks

Another Concrete Example

Label Propagation in

Online Social Networks (Graphs)

Label Propagation Algorithm

- Social Arithmetic:

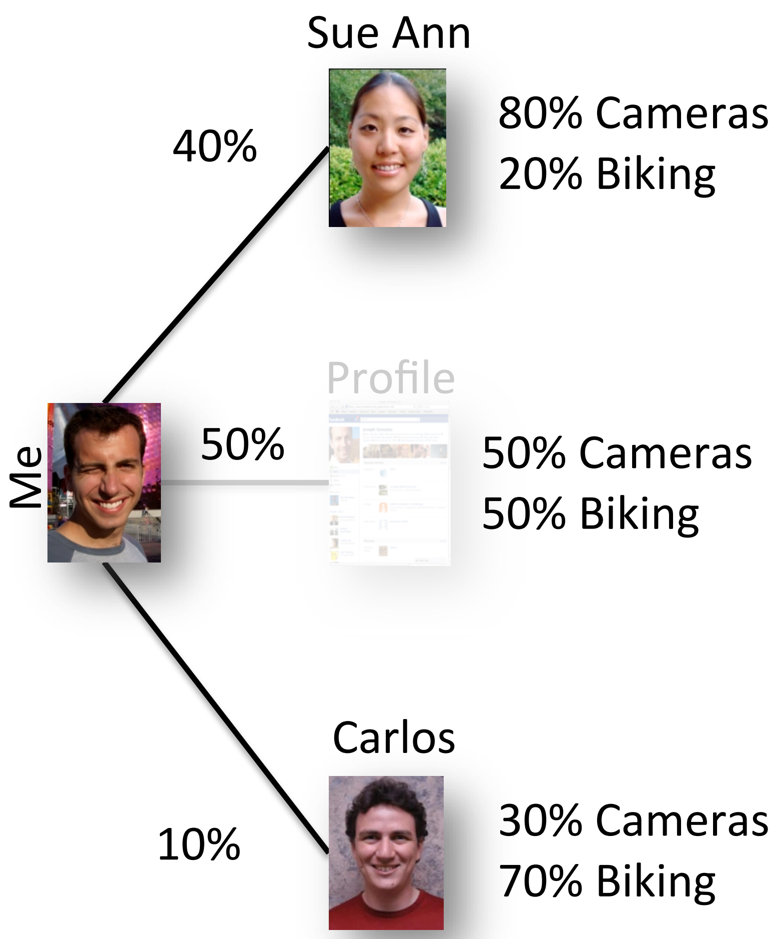
  50% What I list on my profile
  40% Sue Ann Likes
  \+ 10% Carlos Like

  I Like: 60% Cameras, 40% Biking

- Recurrence Algorithm:

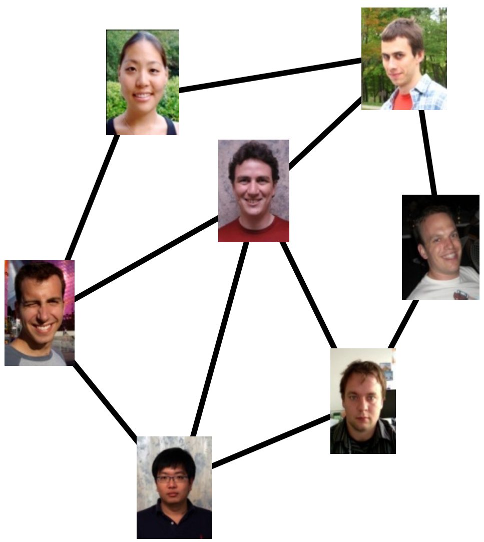
$$Likes[i] = \sum_{j \in Friends[i]} W_{ij} \times Likes[j]$$

  - iterate until convergence
- Parallelism:
  - Compute all *Likes[i]* in parallel

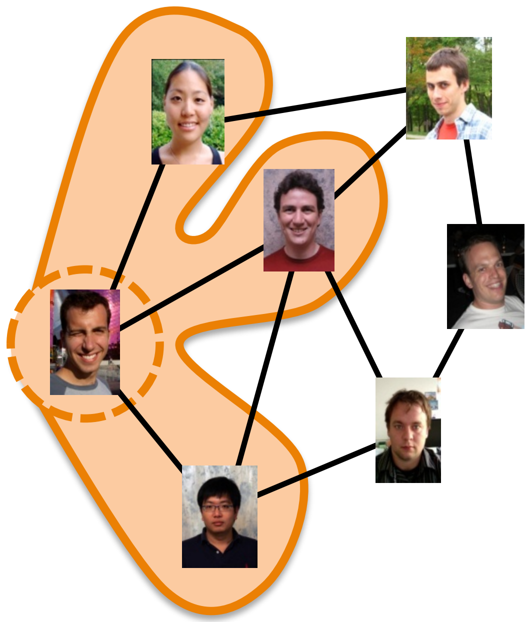


Properties of Graph Parallel Algorithms

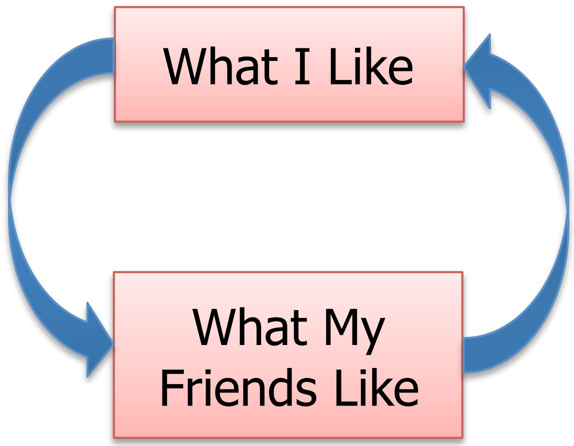
Dependency Graph



Factored Computation



Iterative Computation



Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!

Data-Parallel Graph-Parallel

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Embarrassingly Parallel Tasks

Map Reduce?

Lasso

Label Propagation

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

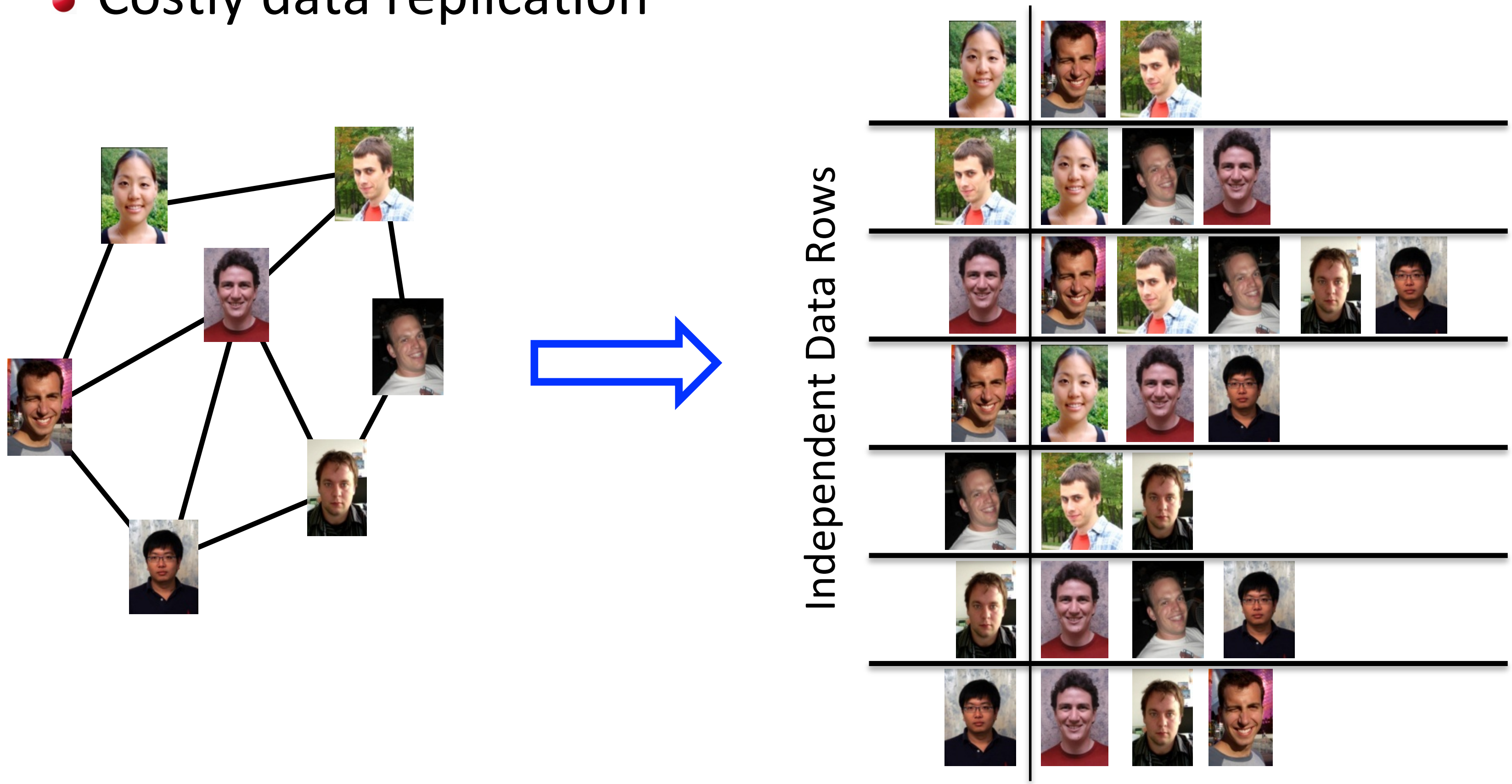
Deep Belief Networks

Neural Networks

Why not use Map-Reduce for Graph Parallel Algorithms?

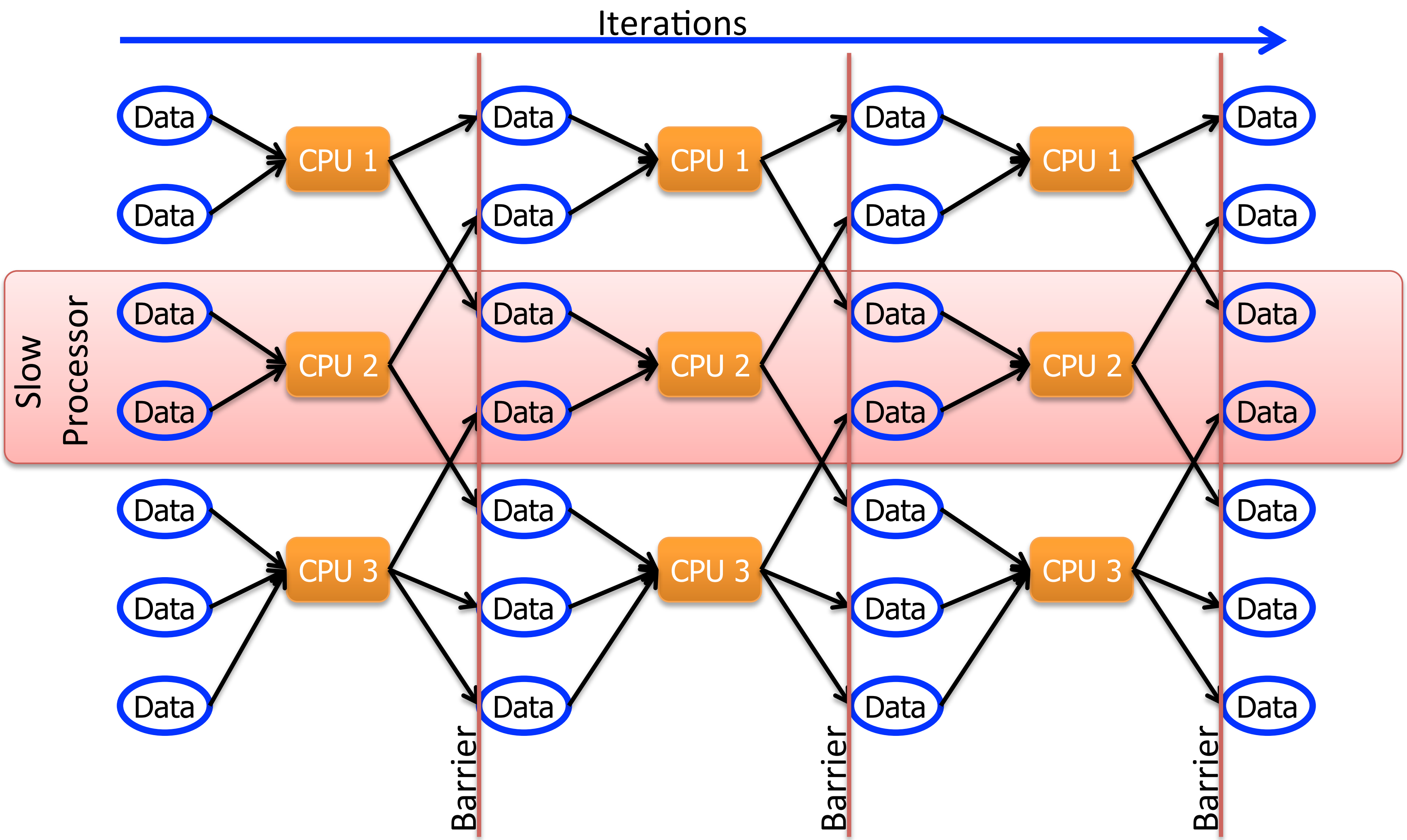
Data Dependencies

- Map-Reduce does not efficiently express dependent data
  - User must code substantial data transformations
  - Costly data replication



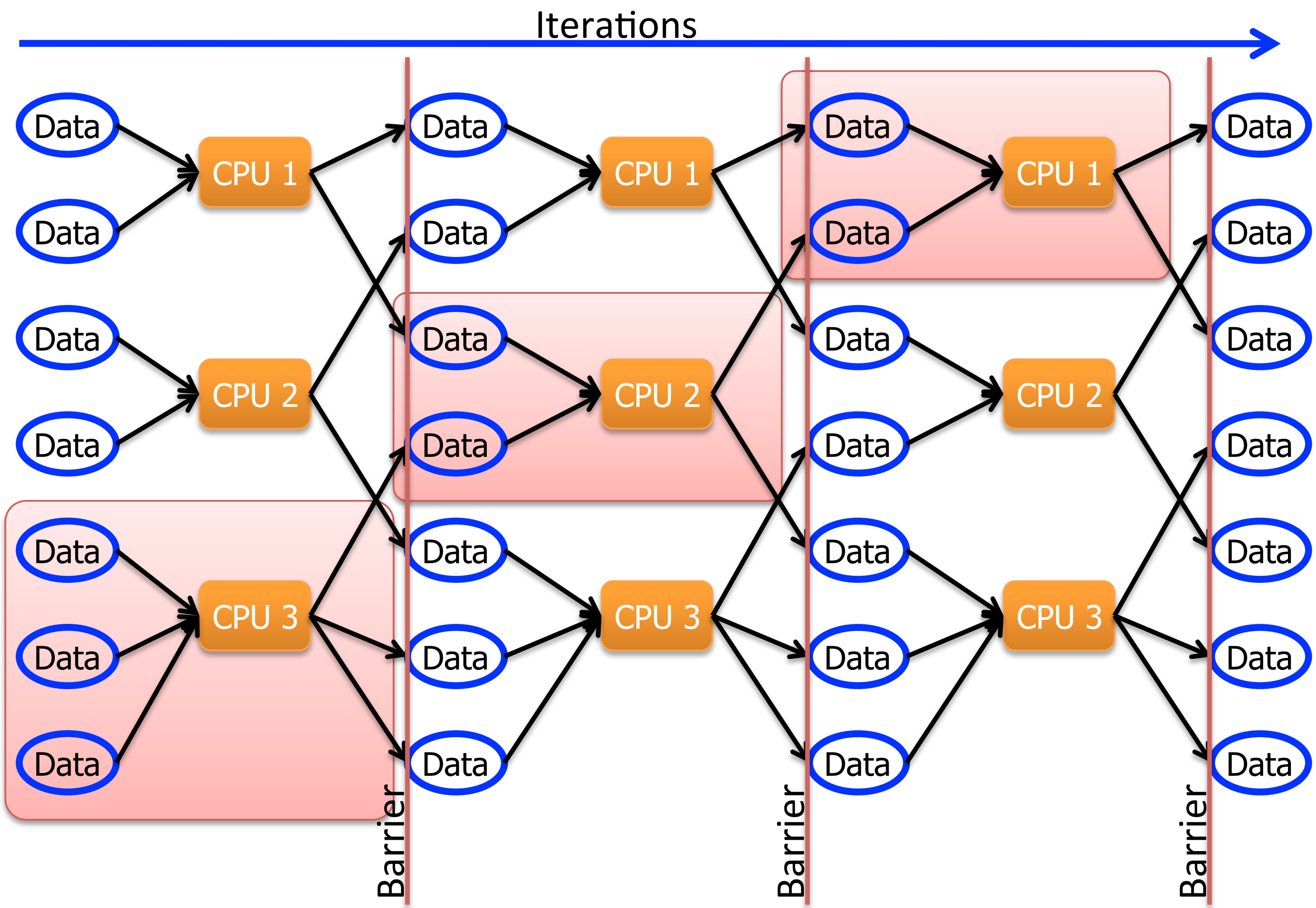
Iterative Algorithms

- Map-Reduce not efficiently express iterative algorithms:



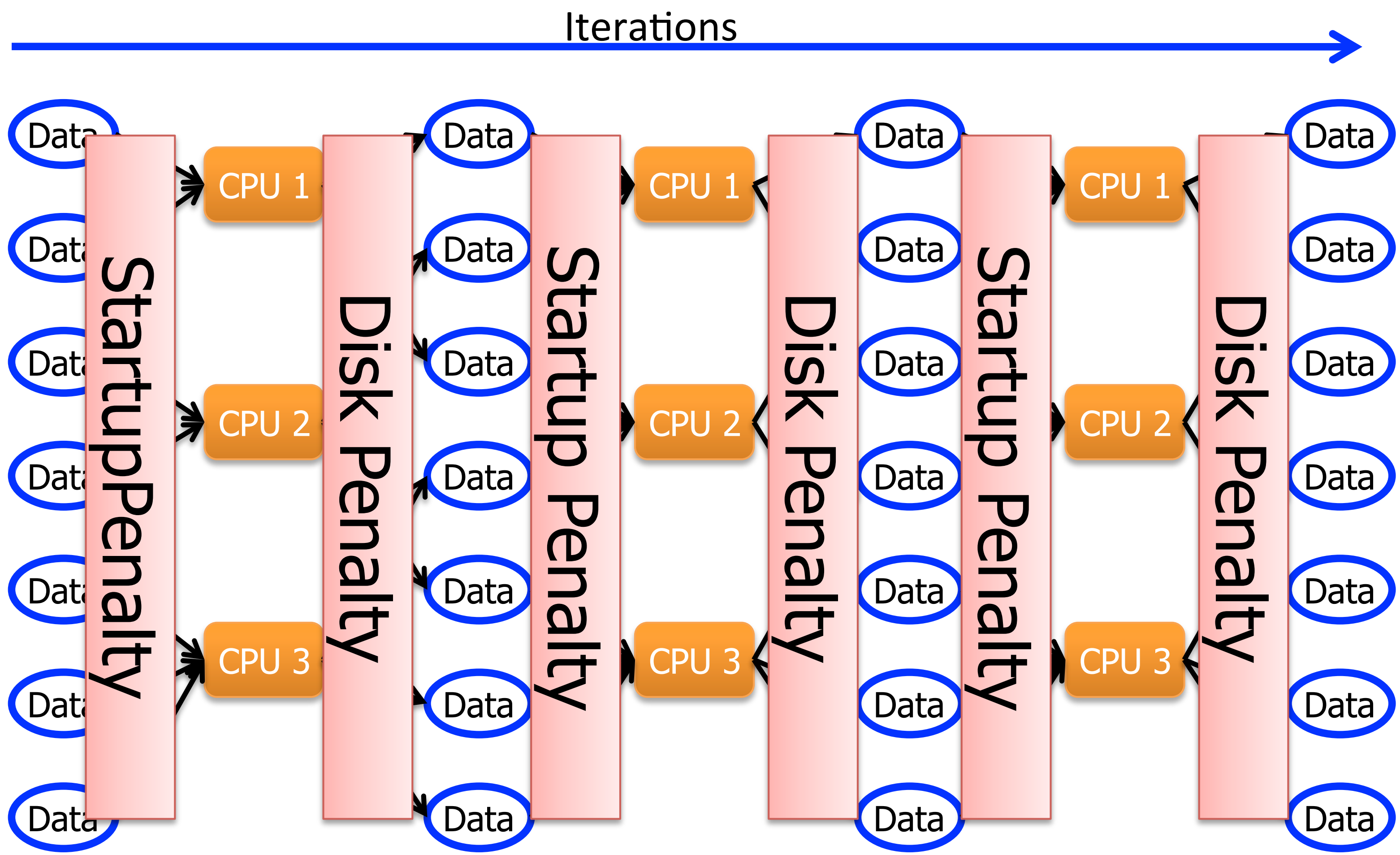
MapAbuse: Iterative MapReduce

- Only a subset of data needs computation:



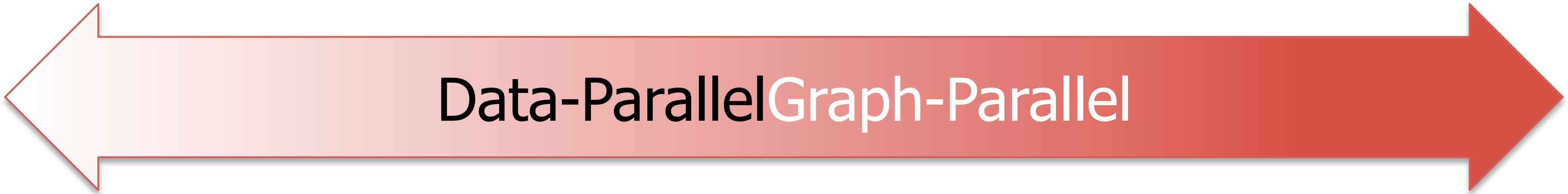
MapAbuse: Iterative MapReduce

- System is not optimized for iteration:



Map-Reduce for Data-Parallel ML

- Excellent for large data-parallel tasks!



Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Pregel (Giraph)?

SVM

Lasso

Kernel Methods

Belief Propagation

Tensor Factorization

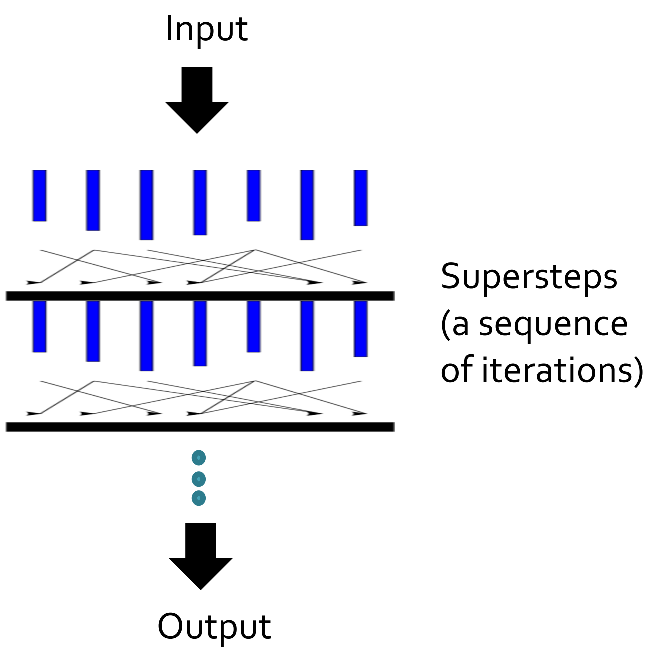
PageRank

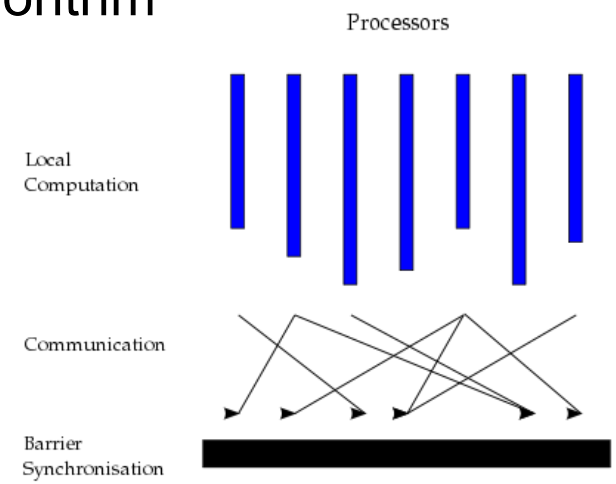
Deep Belief Networks

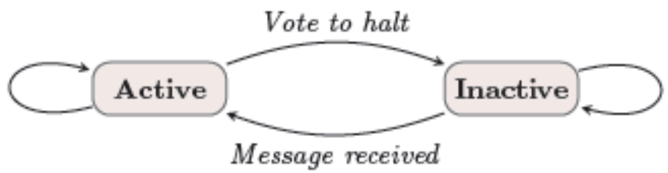
Neural Networks

Pregel (Giraph)

- Google's Pregel for Distributed Graph Processing (mostly in-memory-only)
  - Vertex-centric computation with barrier between successive iterations (aka Super-steps)
  - Inspired by Valiant's Bulk Synchronous Parallel model[4]
  - Open-source version under the Apache Giraph project
  - API with flexibility to express arbitrary algorithm
  - Scalable and Fault-tolerant platform

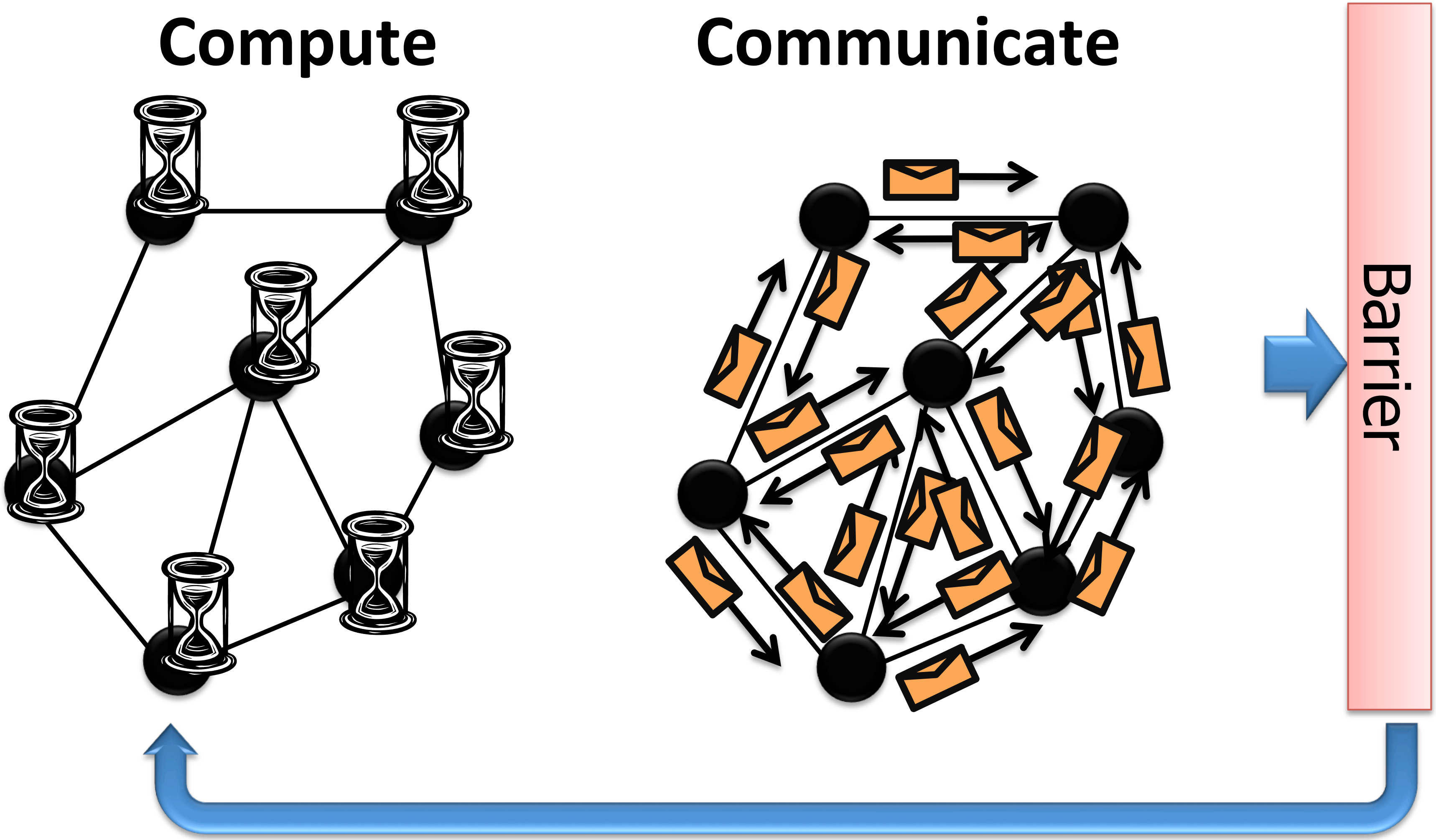






Pregel (Giraph)

- Bulk Synchronous Parallel Model:



PageRank in Giraph (Pregel)

$$R[i] = \alpha + (1 - \alpha) \sum_{(j,i) \in E} \frac{1}{L[j]} R[j]$$

```
bsp_page_rank() {
  sum = 0
  forall (message in in_messages())
     sum = sum + message
  rank = ALPHA + (1-ALPHA) * sum;
  set_vertex_value(rank);
```

Sum PageRank over incoming messages

```
  if (current_super_step() < MAX_STEPS) {
     nedges = num_out_edges()
     forall (neighbors in out_neighbors())
        send_message(rank / nedges);
  } else vote_to_halt();
}
```

Send new messages to neighbors or terminate

Computation Model for Pregel

- Within each Super-Step, concurrent computation and communication need not be ordered in time
- Communication through message passing
- Each vertex
  - Receives messages sent in the previous Super-step
  - Executes the same user-defined function
  - Modifies its value or that of its outgoing edges
  - Sends messages to other vertices (to be received in the next superstep)
  - Mutates the topology of the graph
  - Votes to halt if it has no further work to do

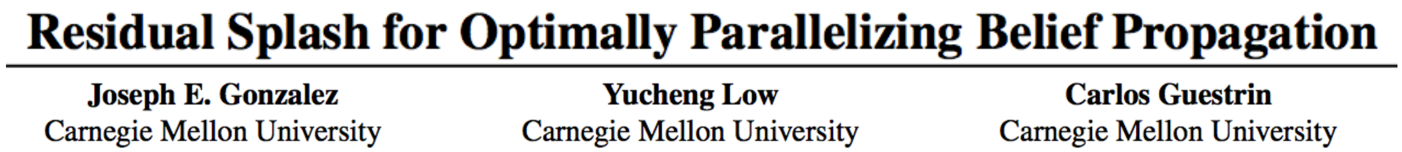
Problem

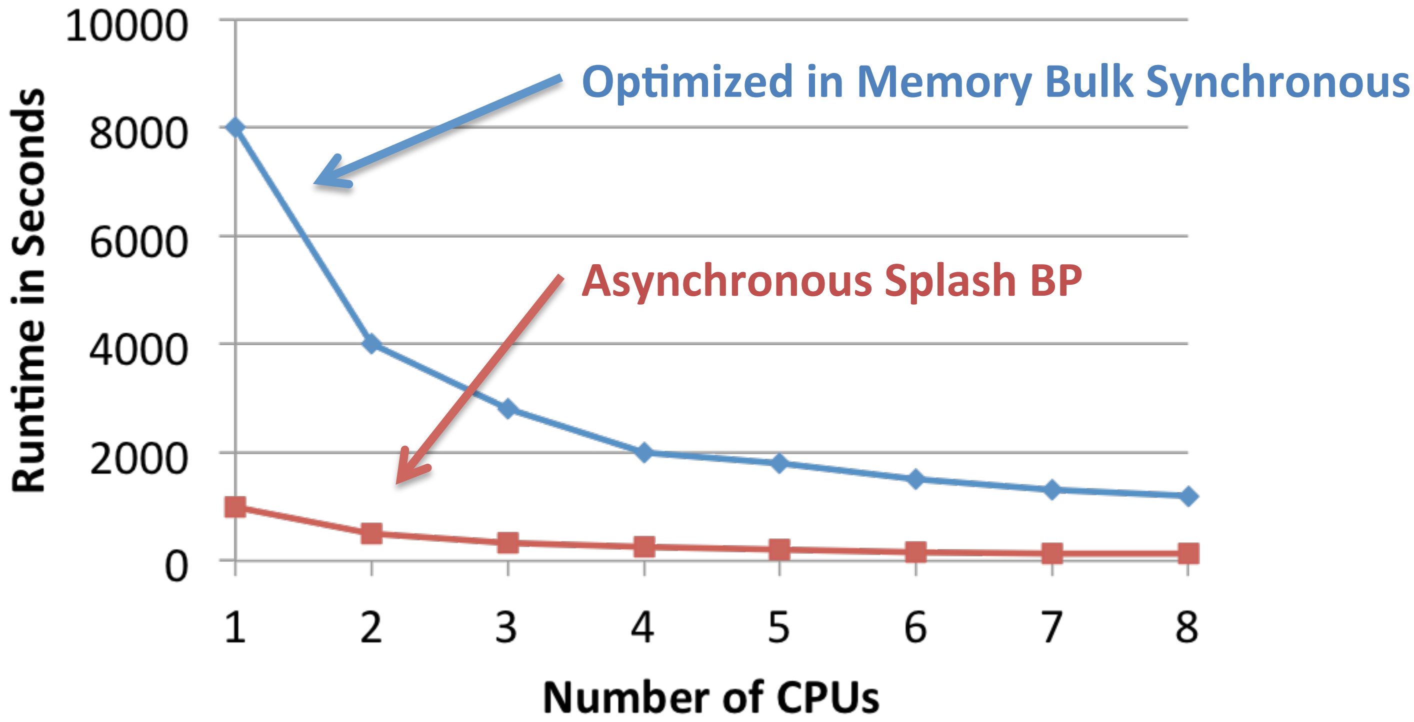
Bulk synchronous computation can be highly inefficient.

Example:

Loopy Belief Propagation

Data-Parallel Algorithms can be **Inefficient**





The limitations of the Map-Reduce abstraction can lead to inefficient parallel algorithms.

The Need for a New Abstraction

- Map-Reduce is not well suited for Graph-Parallelism

Data-Parallel Graph-Parallel

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Pregel (Giraph)

Carnegie Mellon

SVM

Kernel Methods

Belief Propagation

Tensor Factorization

PageRank

Lasso

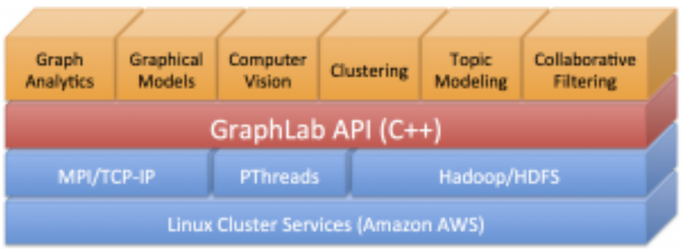
Deep Belief Networks

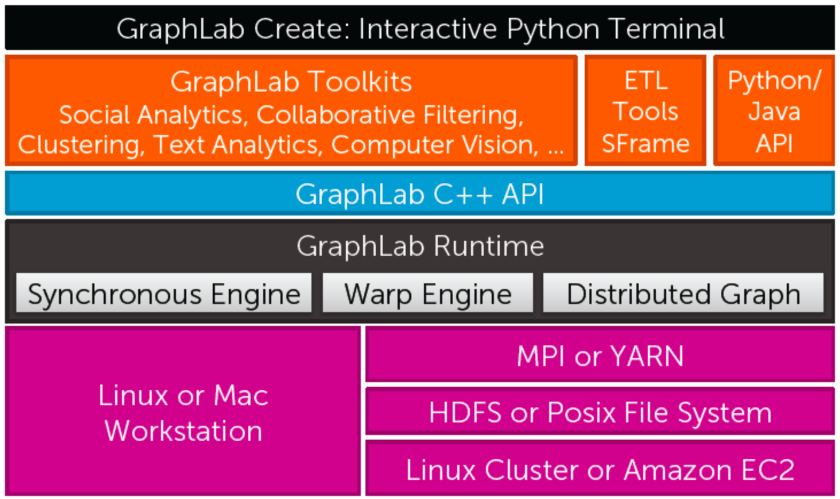
Neural Networks

What is GraphLab?

Graph-based Big Learning/ Parallel Processing Platforms (cont'd)

- GraphLab – another vertex-centric model (http://GraphLab.org/projects, http://GraphLab.com)
  - Originated from CMU and now by UWashington@Seattle ;
  - Different versions supporting wide-range of platforms:
    - GraphLab 1.0 was designed to run on closely-coupled, shared-memory multicore machine.
    - GraphChi enables a Single PC to process graphs with billions of edges
    - GraphLab (Ver2.x) or so-called the PowerGraph model targets for seriously-imbalanced node degrees found in practical (Natural) graphs and support parallel processing on Share-Nothing Cluster architecture
      - Taking the split-vertex instead split-edge approach
    - GraphCreate (Beta) allows you to code in your PC using Python but deploy to run over Cloud-based shared-nothing clusters.





The GraphLab Framework

Graph Based
Data Representation

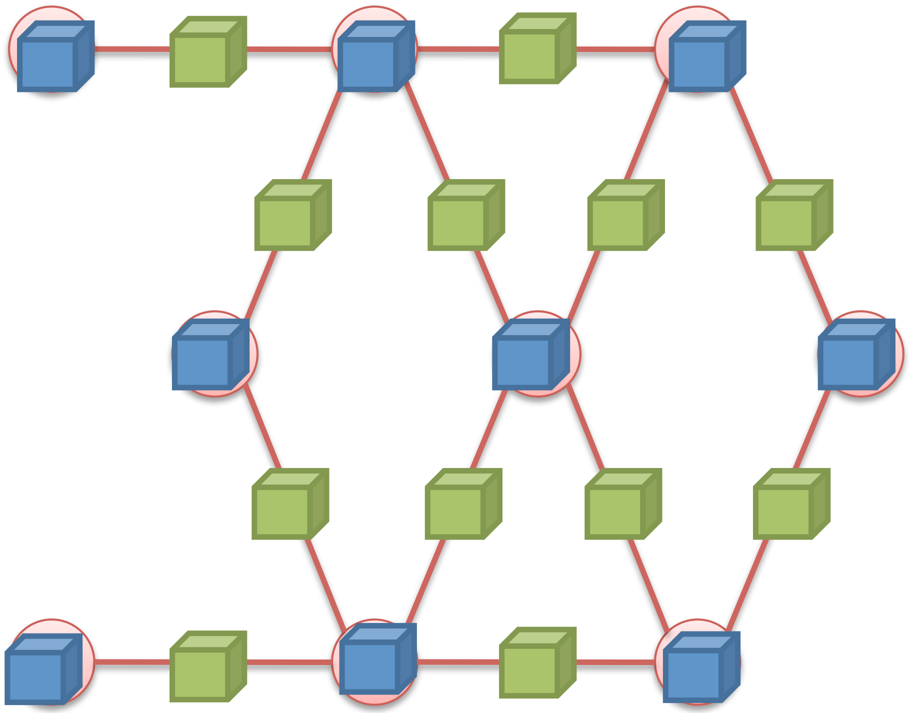
Update Functions
User Computation

Scheduler

Consistency Model

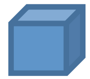
Data Graph

A **graph** with arbitrary data (C++ Objects) associated with each vertex and edge.

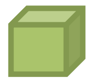


Graph: 

- Social Network

Vertex Data: 

•User profile text
- Current interests estimates

Edge Data: 

- Similarity weights

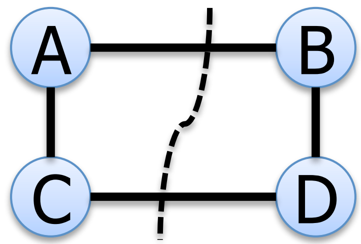
Implementing the Data Graph

Multicore Setting

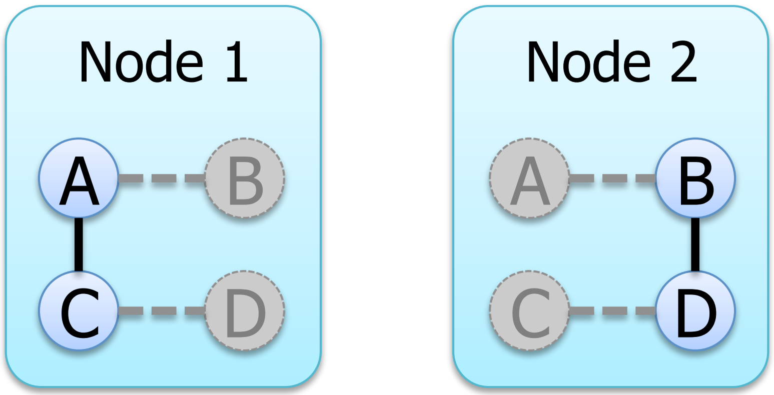
- **In Memory**
- Relatively Straight Forward
  - vertex_data(vid) → data
  - edge_data(vid,vid) → data
  - neighbors(vid) → vid_list
- Challenge:
  - Fast lookup, low overhead
- Solution:
  - Dense data-structures
  - Fixed Vdata&Edata types
  - Immutable graph structure

Cluster Setting

- **In Memory**
- Partition Graph:
  - ParMETIS or Random Cuts



- Cached Ghosting



The GraphLab Framework

Graph Based
Data Representation

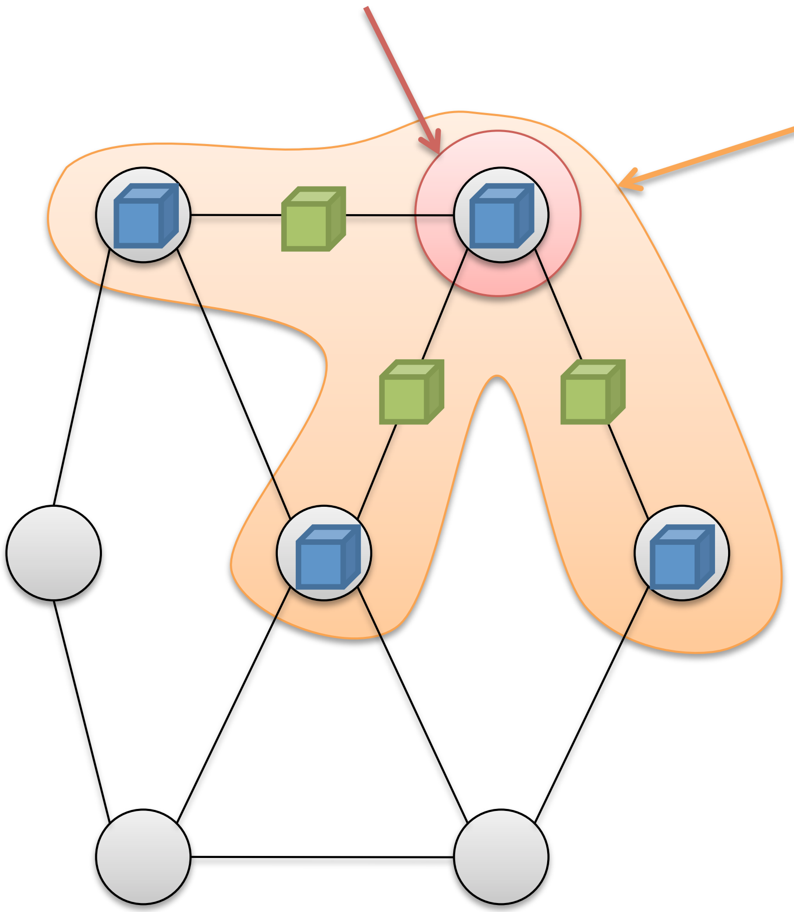
Update Functions
User Computation

Scheduler

Consistency Model

Update Functions

An **update function** is a user defined program which when applied to a **vertex** transforms the data in the **scope**of the vertex



```
label_prop(i, scope){
  // Get Neighborhood data
  (Likes[i], Wij, Likes[j]) ←scope;

  // Update the vertex data
```

$$Likes[i] \leftarrow \sum_{j \in Friends[i]} W_{ij} \times Likes[j];$$

```
  // Reschedule Neighbors if needed
  if Likes[i] changes then
    reschedule_neighbors_of(i);
}
```

The GraphLab Framework

Graph Based
Data Representation

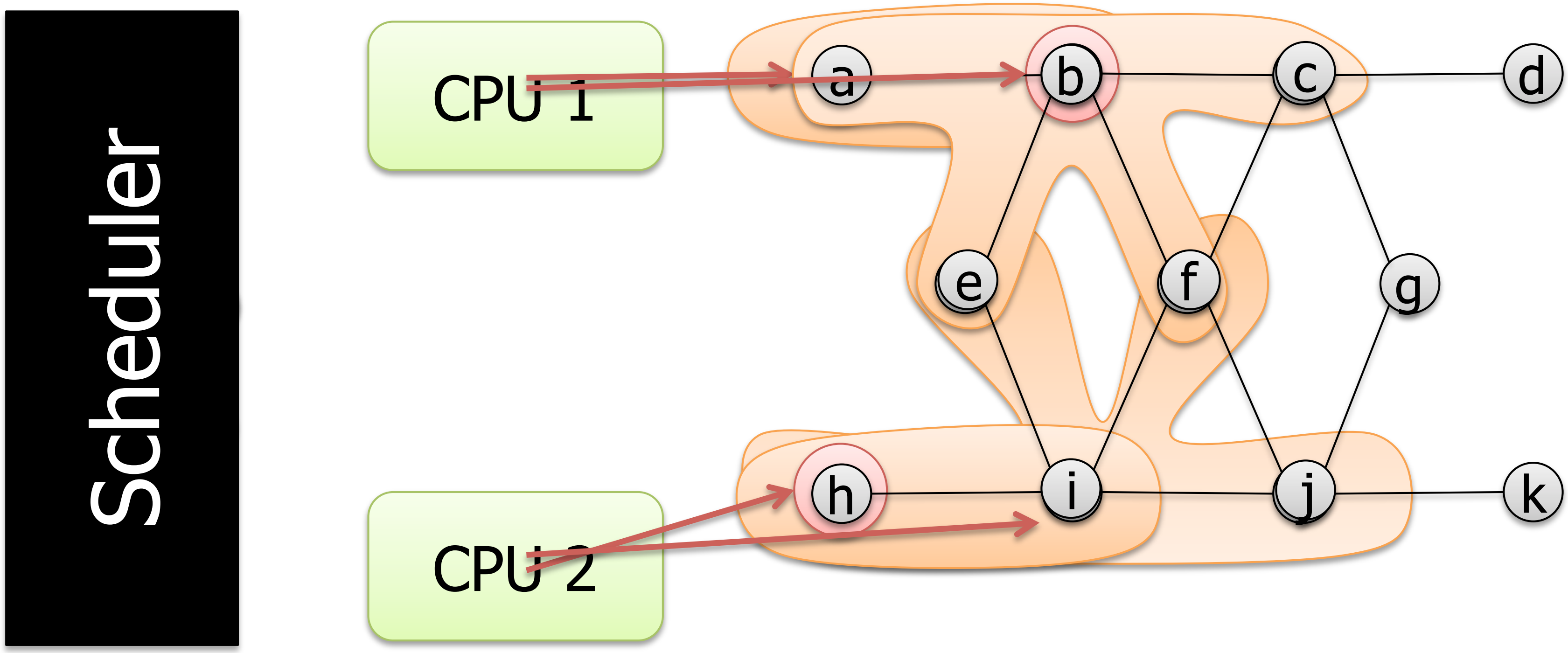
Update Functions
User Computation

Scheduler

Consistency Model

The Scheduler

The **scheduler** determines the order that vertices are updated.



The process repeats until the scheduler is empty.

Choosing a Schedule

The choice of schedule affects the correctness and parallel performance of the algorithm

- GraphLab provides several different schedulers
  - Round Robin: vertices are updated in a fixed order
  - FIFO: Vertices are updated in the order they are added
  - Priority: Vertices are updated in priority order

Obtain different algorithms by simply changing a flag!

```
--scheduler=roundrobin
          --scheduler=fifo
                 --scheduler=priority
```

The GraphLab Framework

Graph Based
Data Representation

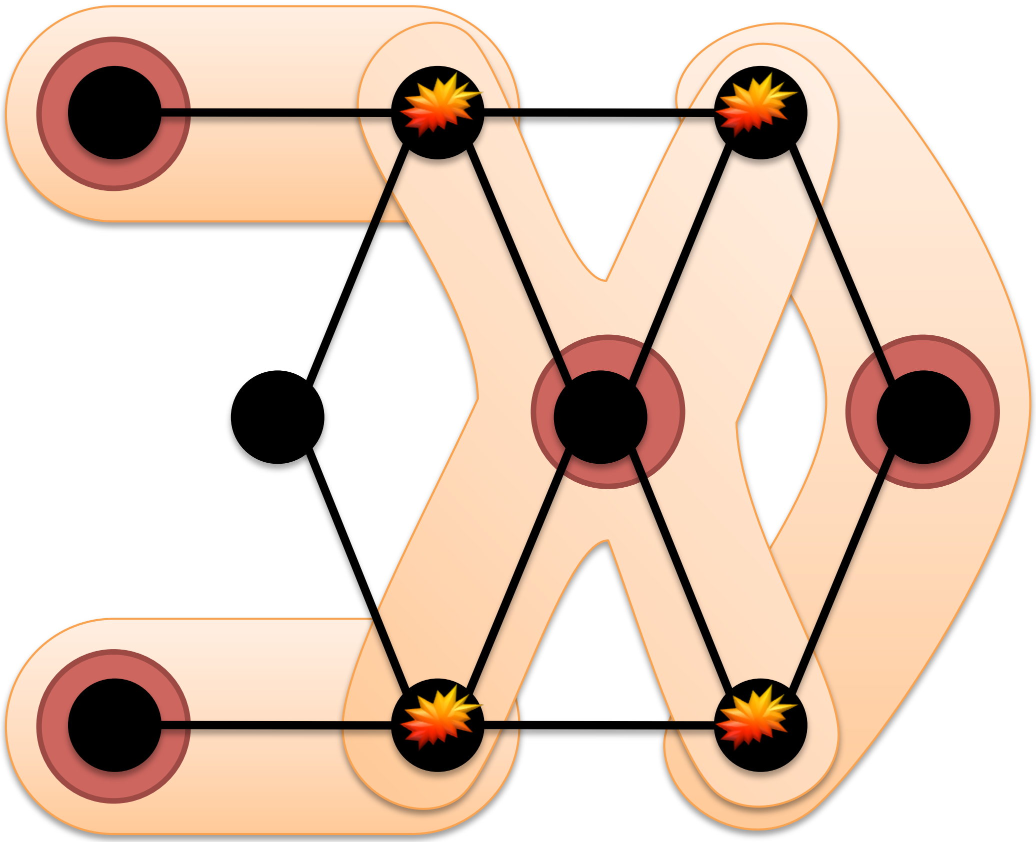
Update Functions
User Computation

Scheduler

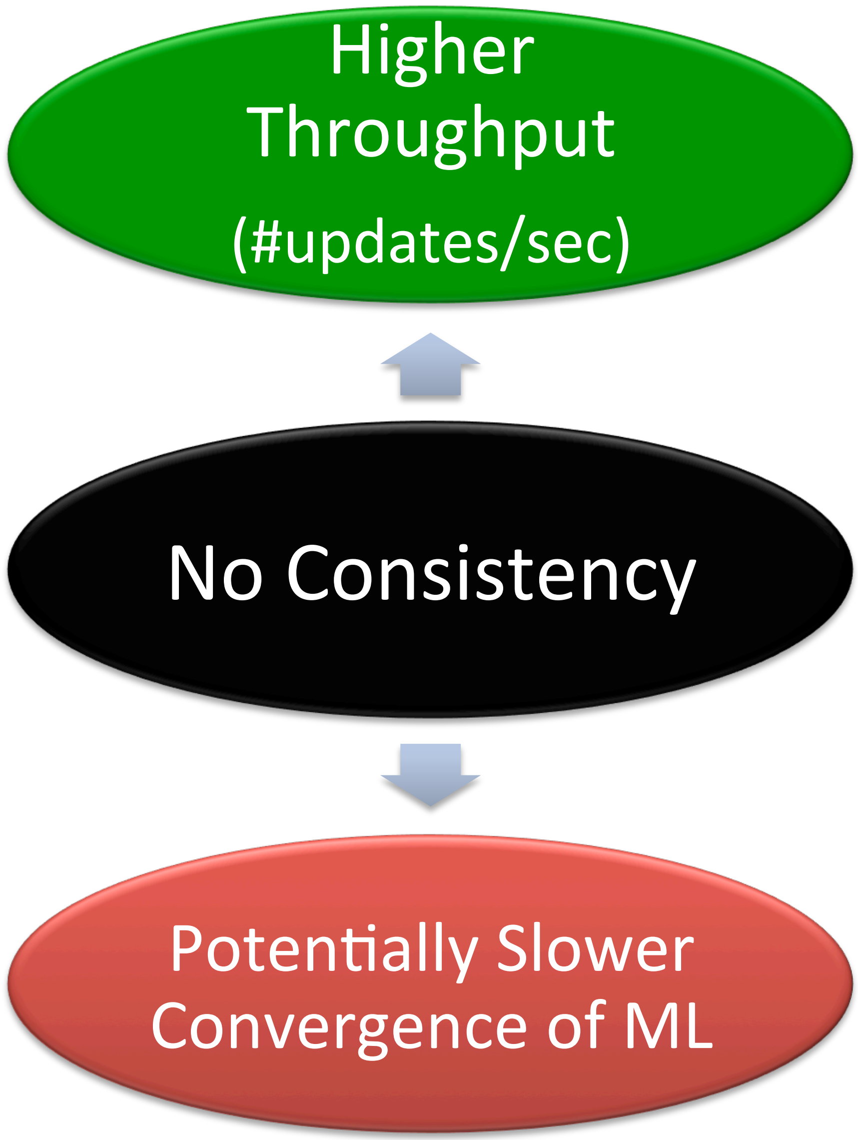
Consistency Model

Ensuring Race-Free Code

- How much can computation **overlap**?



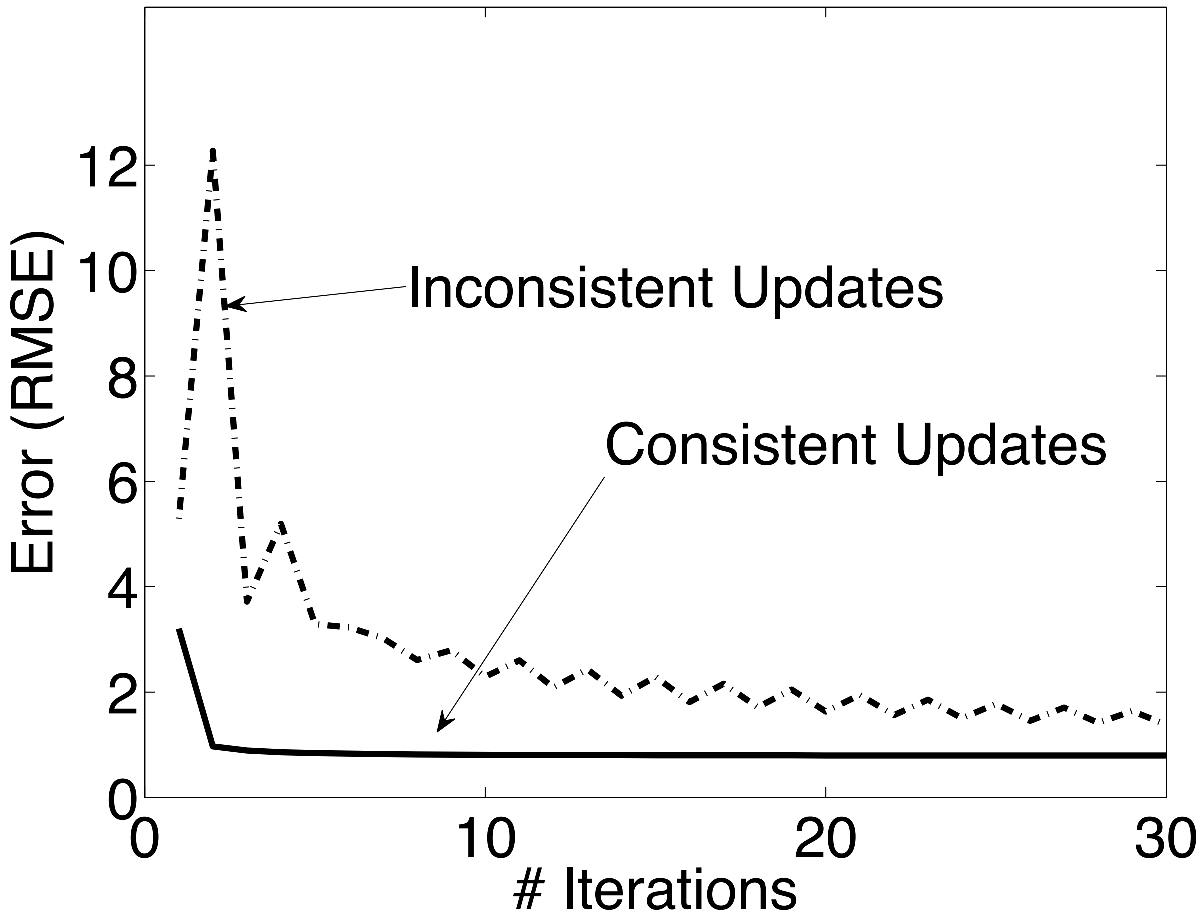
Need for Consistency?



Importance of Consistency

Many algorithms require strict consistency, or performs significantly better under strict consistency.

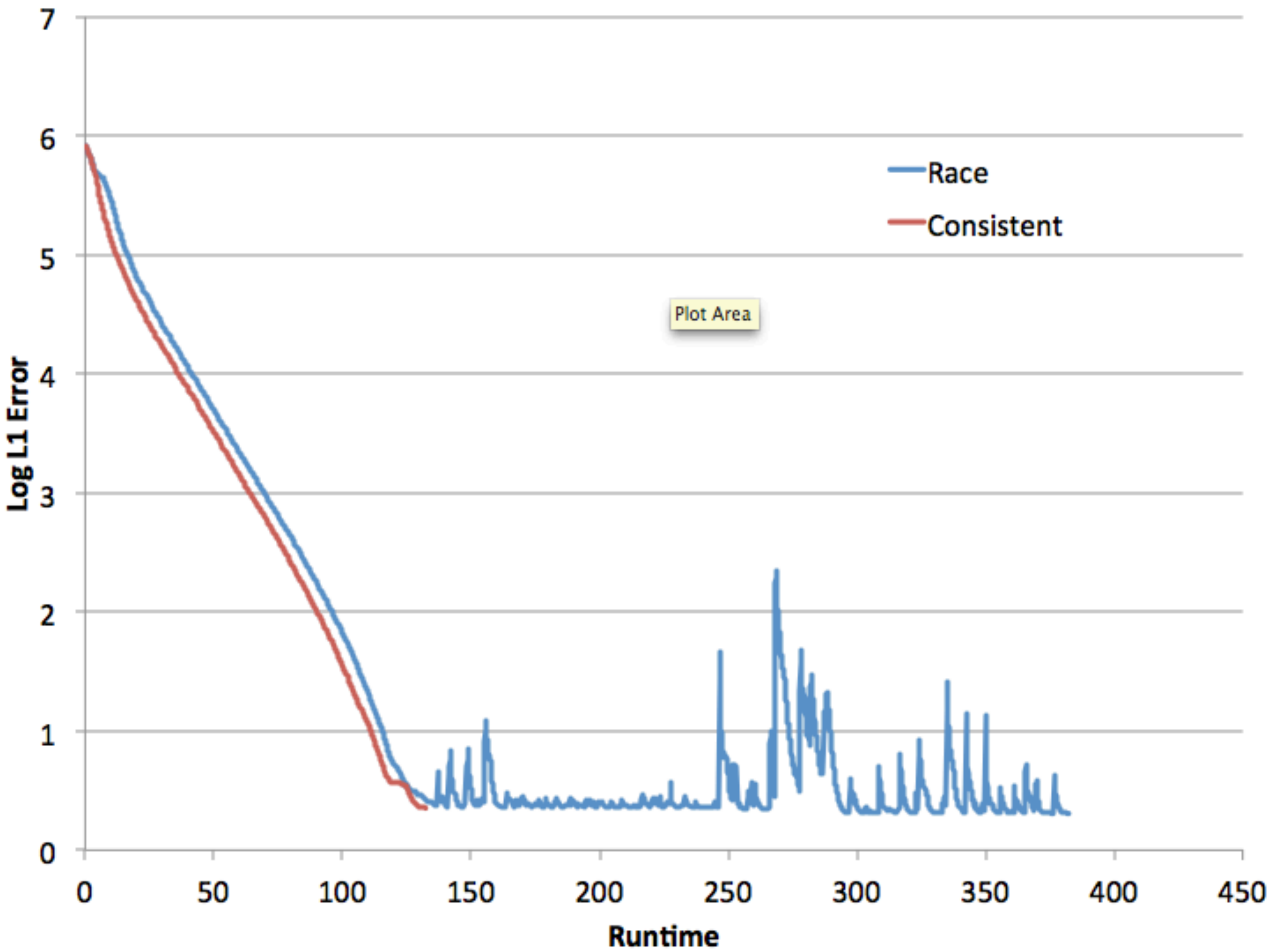
Alternating Least Squares



Even Simple PageRank can be Dangerous

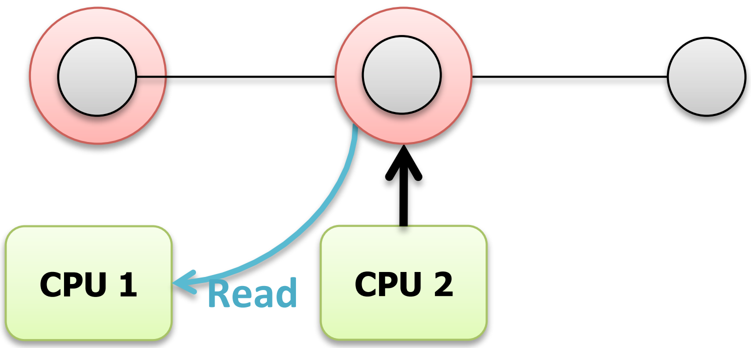
```
GraphLab_pagerank(scope) {
      ref sum = scope.center_value
      sum = 0
      forall (neighbor in scope.in_neighbors )
            sum = sum + neighbor.value / nbr.num_out_edges
      sum = ALPHA + (1-ALPHA) * sum
      …
```

Inconsistent PageRank



Even Simple PageRank can be Dangerous

```
GraphLab_pagerank(scope) {
    ref sum = scope.center_value
    sum = 0
    forall (neighbor in scope.in_neighbors)
        sum = sum + neighbor.value / nbr.num_out_edges
    sum = ALPHA + (1-ALPHA) * sum
    …
```



Read-write race ➔
CPU 1 reads bad PageRank estimate,
as CPU 2 computes value

Race Condition Can Be Very Subtle

Unstable

```
GraphLab_pagerank(scope) {
      ref sum = scope.center_value
      sum = 0
      forall (neighbor in scope.in_neighbors)
            sum = sum + neighbor.value /
neighbor.num_out_edges
      sum = ALPHA + (1-ALPHA) * sum
      …
```

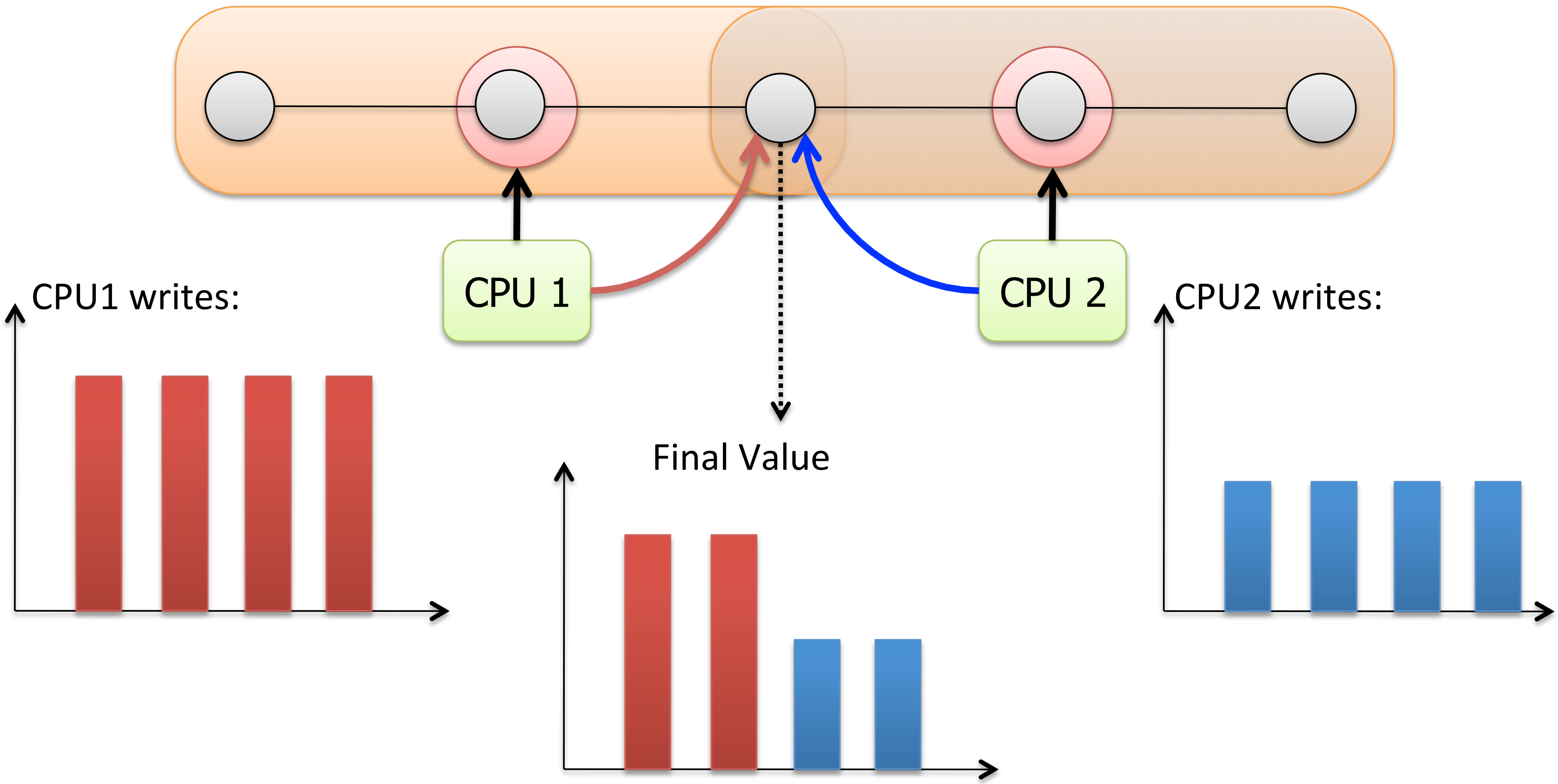
Stable

```
GraphLab_pagerank(scope) {
      sum = 0
      forall (neighbor in scope.in_neighbors)
            sum = sum + neighbor.value / nbr.num_out_edges
      sum = ALPHA + (1-ALPHA) * sum
      scope.center_value = sum
      …
```

This was actually encountered in user code.

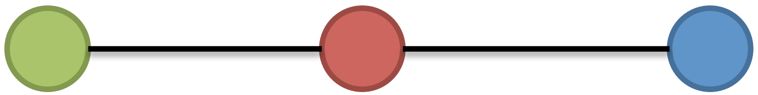
Common Problem: Write-Write Race

Processors running **adjacent update functions** simultaneously modify shared data:



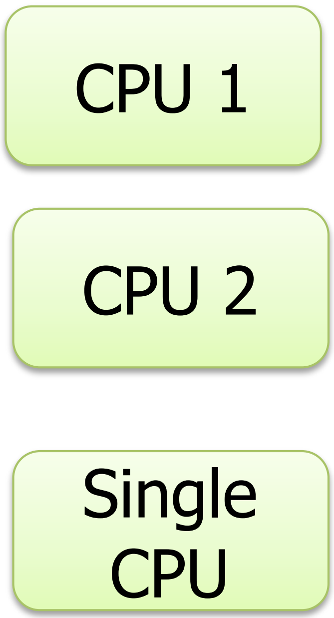
GraphLab Ensures **Sequential Consistency**

For **each parallel execution**, there exists a **sequential execution** of update functions which produces the same result.



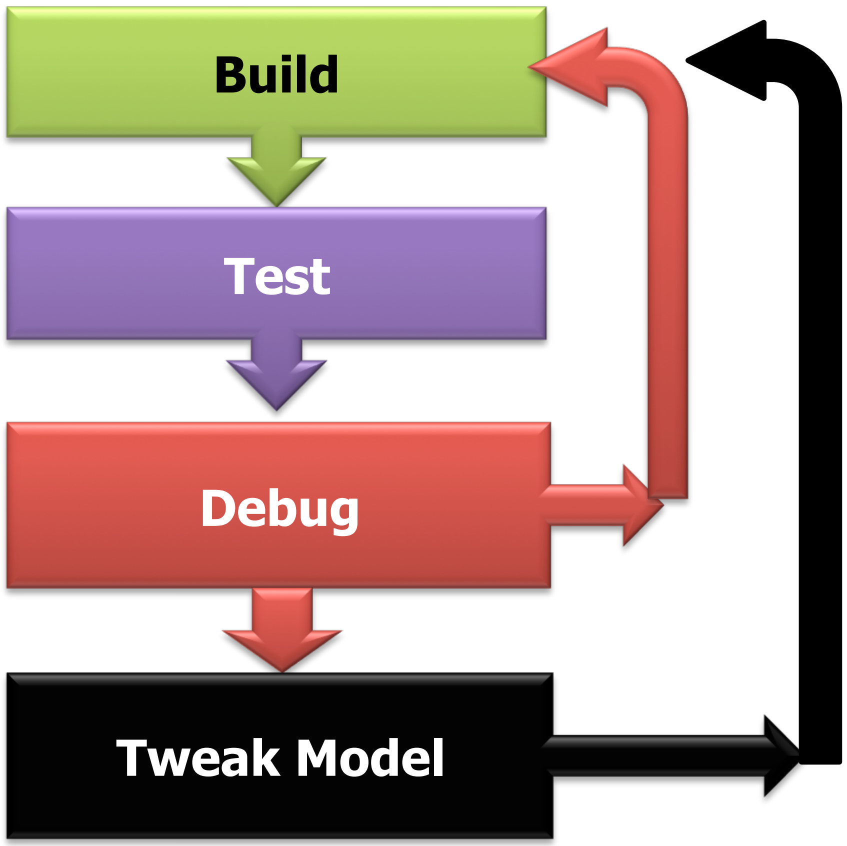
Parallel

Sequential

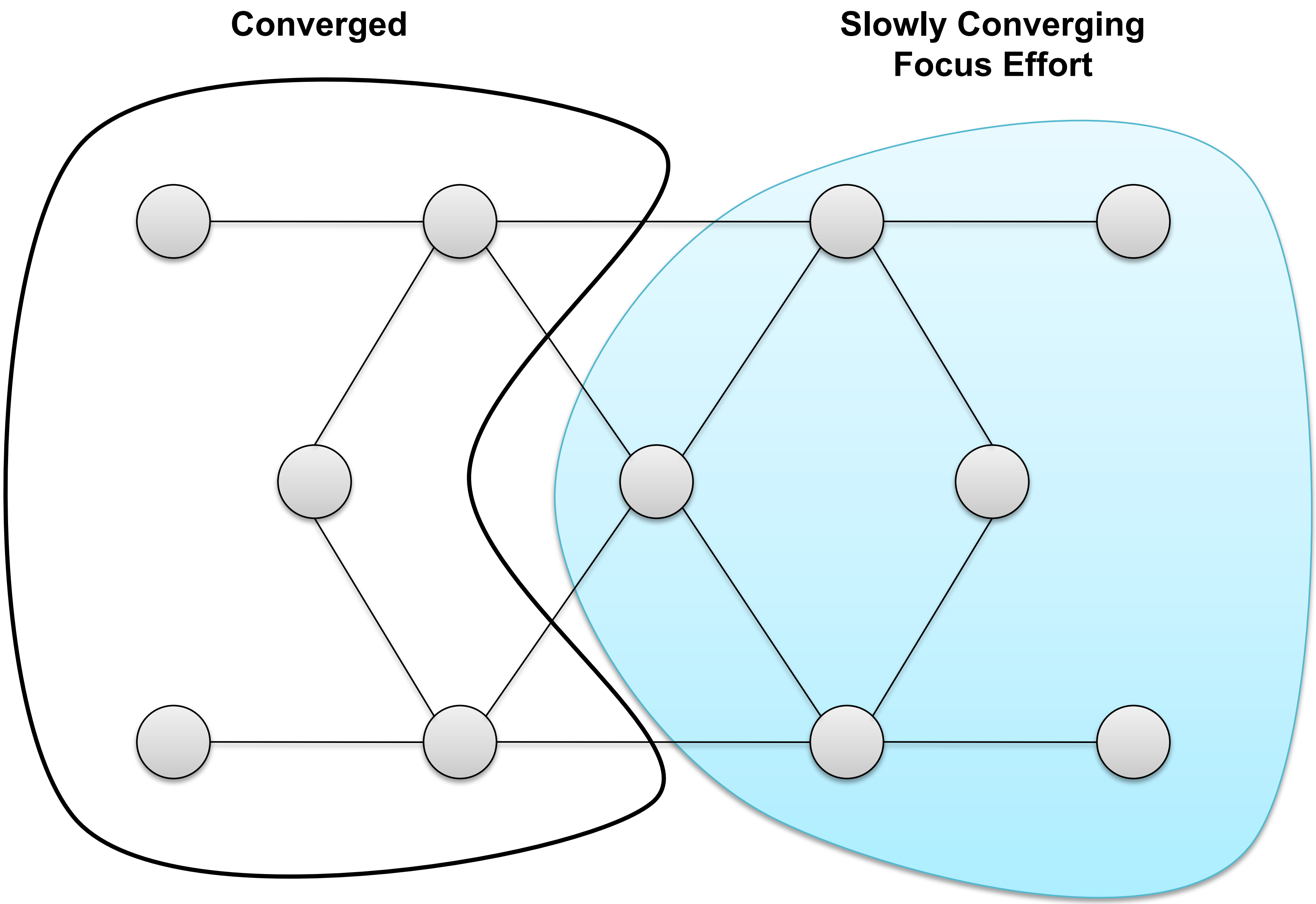


Importance of Consistency

Machine learning algorithms require "model debugging"



Dynamic Computation



PageRank Update Function

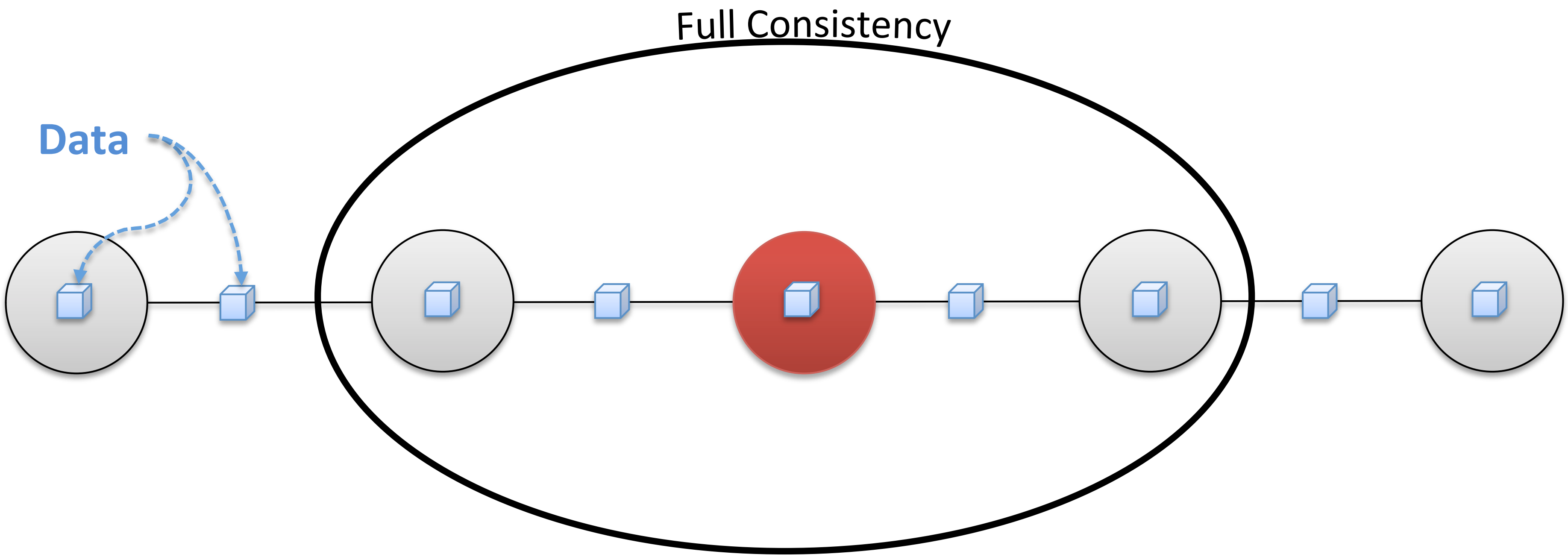
$$R[i] = \alpha + (1-\alpha) \sum_{(j,i)\in E} \frac{1}{L[j]} R[j]$$

Directly Read Neighbor Values

Dynamically Schedule Computation

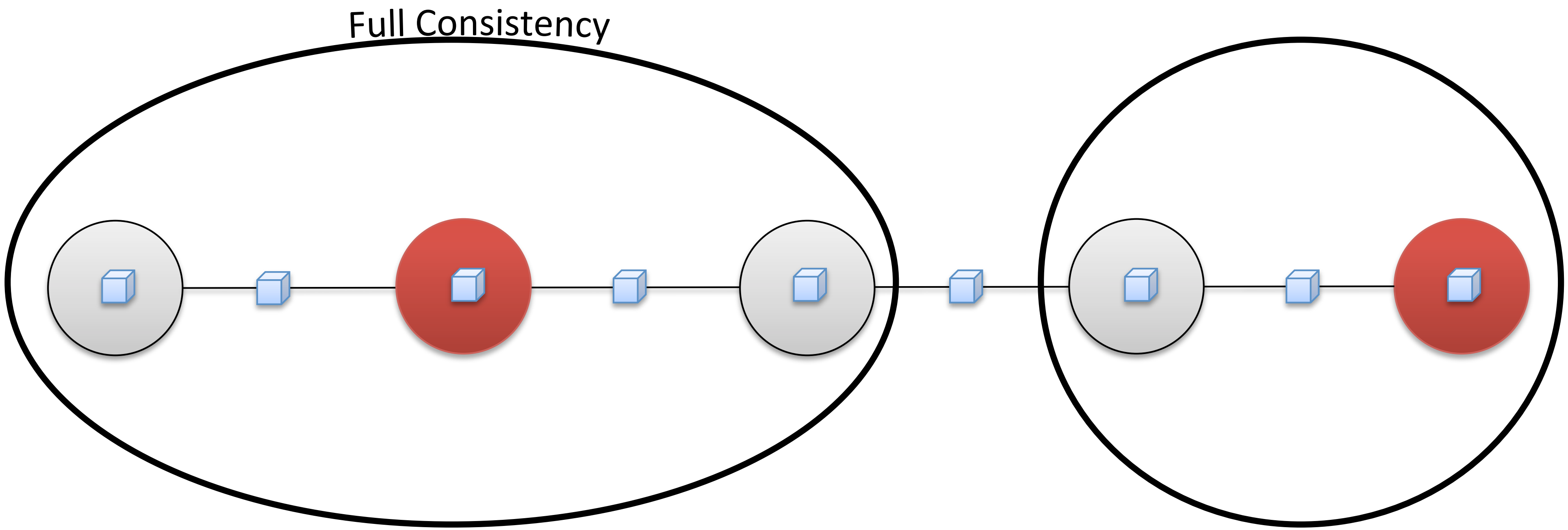
```
GraphLab_pagerank(scope) {
      double sum = 0;
      forall ( nbr in scope.in_neighbors() )
            sum = sum + neighbor.value() /
nbr.num_out_edges();

      double old_rank = scope.ve
      scope.center_value() = ALP

      double residual = abs(scope.center_value() – old_rank);
      if (residual > EPSILON)
            reschedule_out_neighbors();
}
```

Consistency Rules

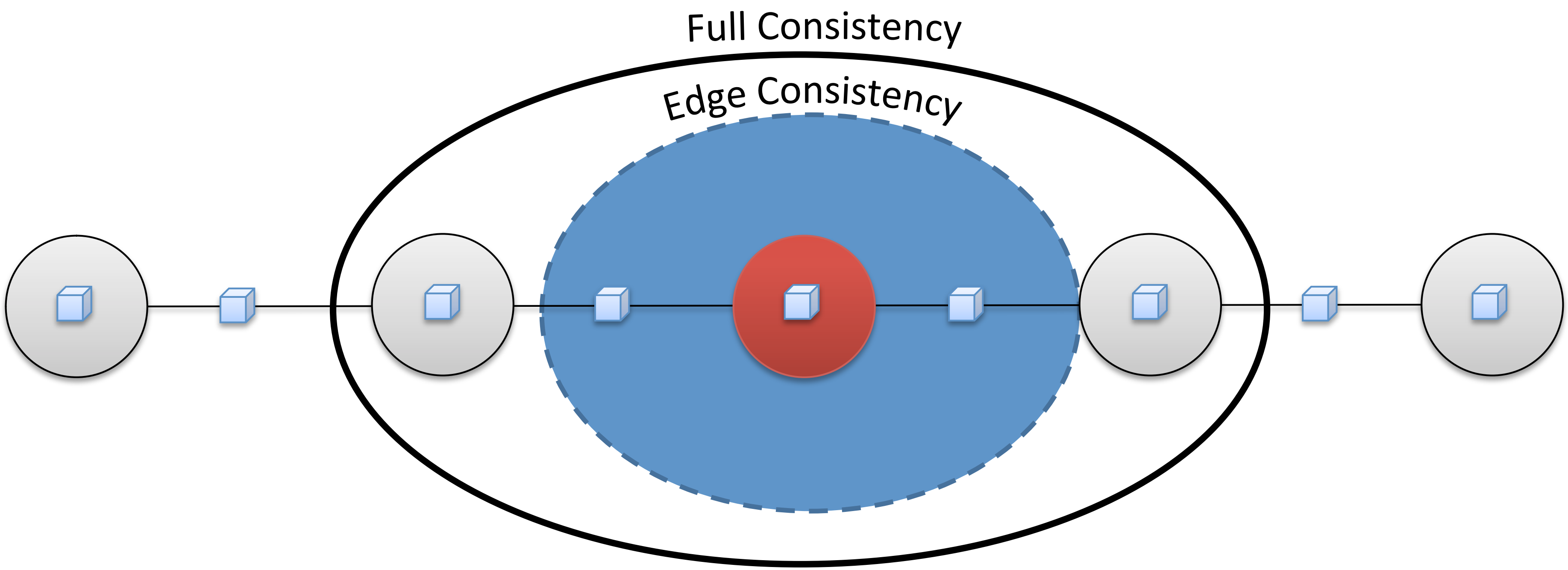


Guaranteed sequential consistency for all update functions

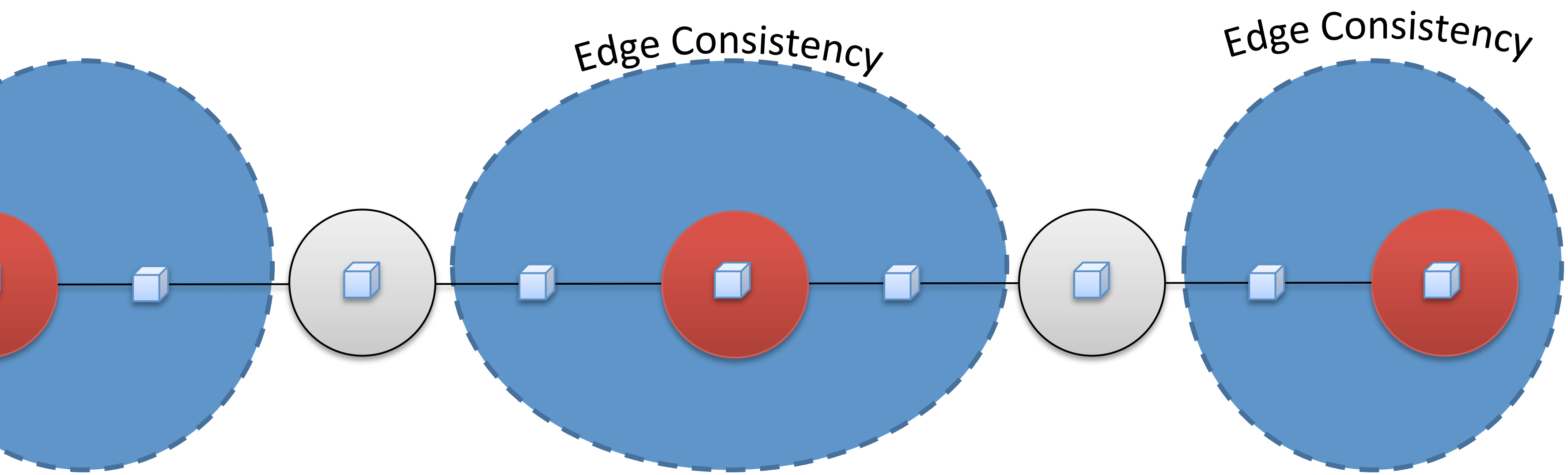
Full Consistency



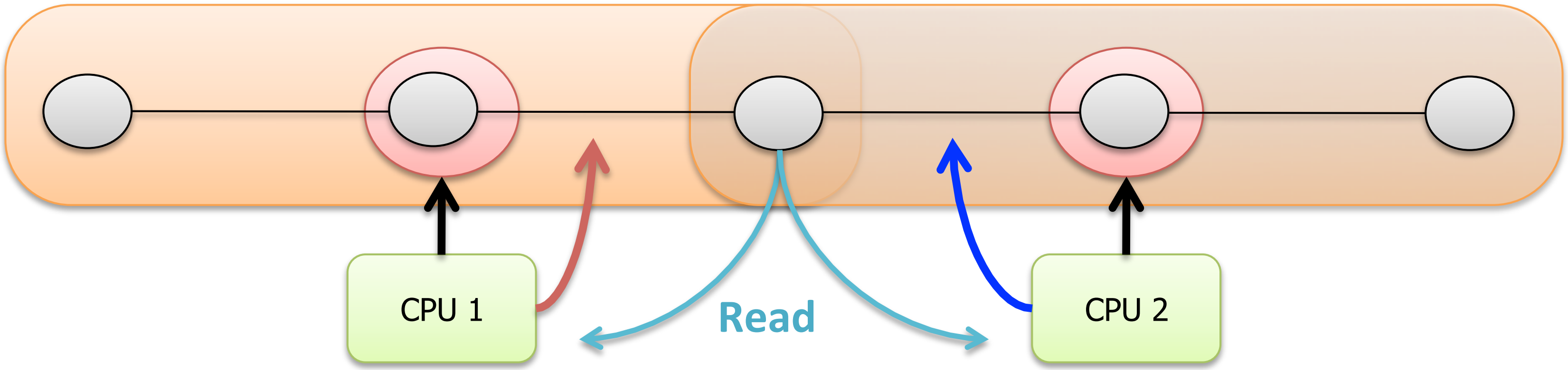
Obtaining More Parallelism



Edge Consistency

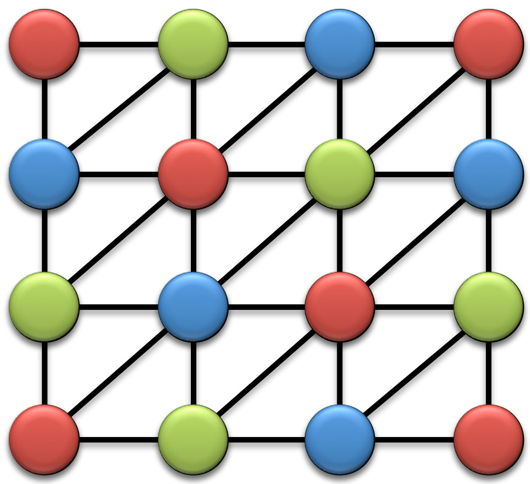


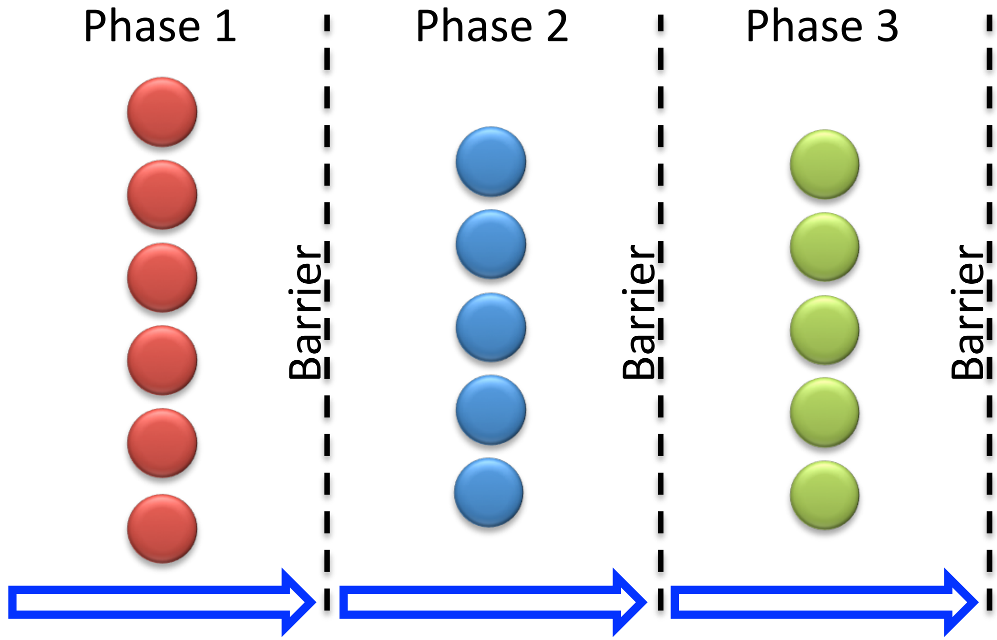
Safe



Consistency Through Scheduling

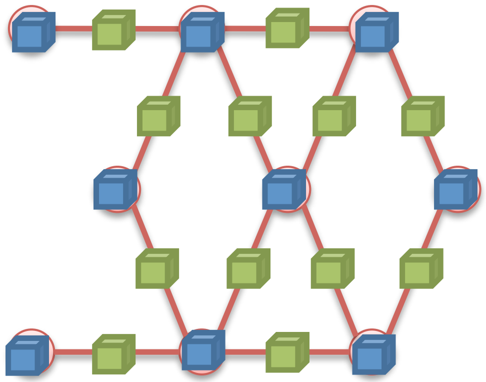
- Edge Consistency Model:
  - Two vertices can be **Updated** *simultaneously* if they do not share an edge.
- Graph Coloring:
  - Two vertices can be assigned the same color if they do not share an edge.



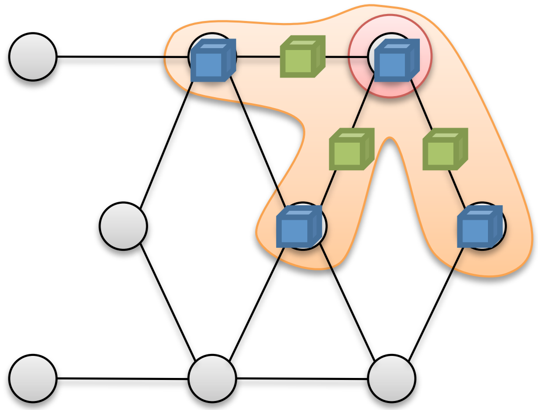


The GraphLab Framework

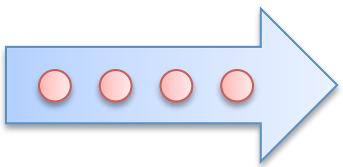
Graph Based
Data Representation



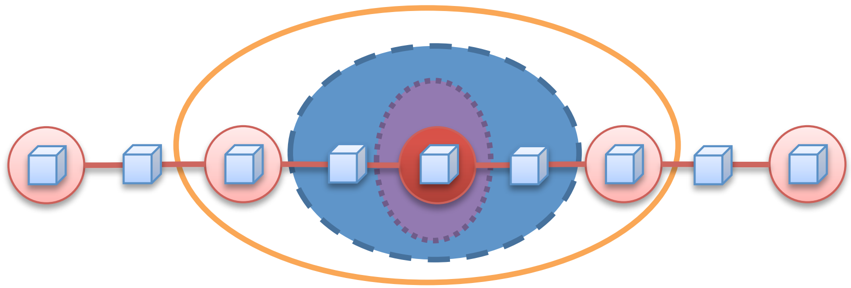
Update Functions
User Computation



Scheduler



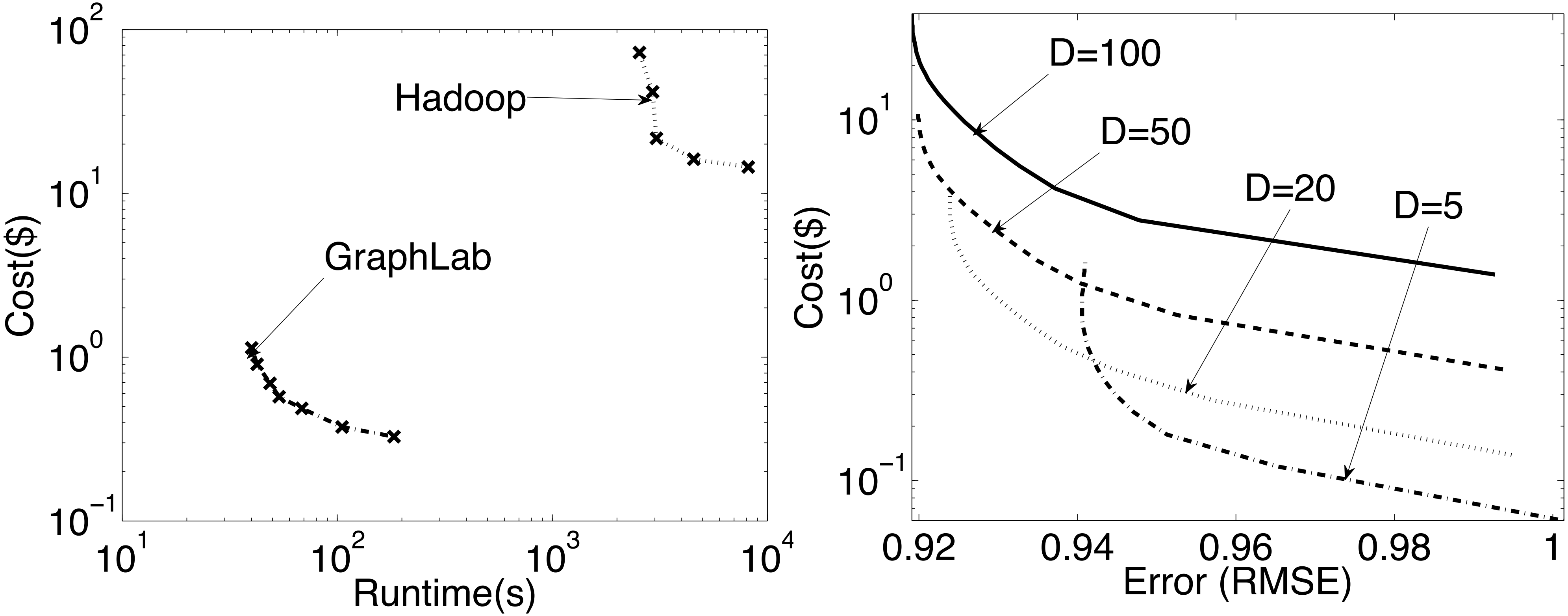
Consistency Model



Algorithms Implemented in GraphLab (1.x)

- PageRank
- K-Means++
- Matrix Factorization
- 5-line codes for a real Recommendation Systems
- Label-Propagation
- Loopy Belief Propagation
- Gibbs Sampling
- CoEM
- Graphical Model Parameter Learning
- Probabilistic Matrix/Tensor Factorization
- Alternating Least Squares
- Lasso with Sparse Features
- Support Vector Machines with Sparse Features
- ...

The Cost of Hadoop

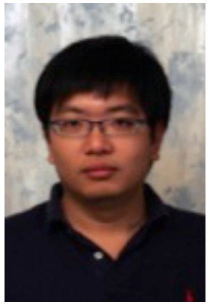


PowerGraph (GraphLab Ver.2)

Distributed Graph-Parallel Computation on Natural Graphs

Joseph Gonzalez

Joint work with:



Yucheng Low

Haijie Gu

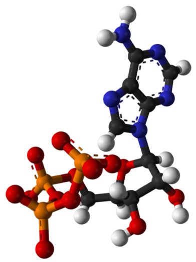
Danny Bickson

Carlos Guestrin

Carnegie Mellon University

Problem:

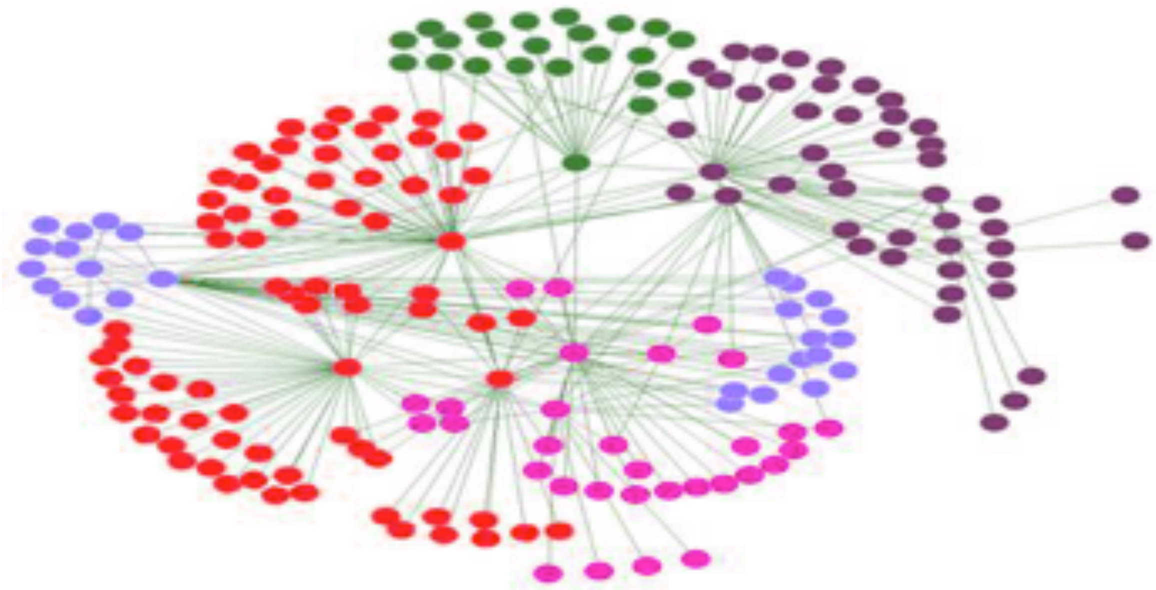
Existing ***distributed*** graph computation systems, including GraphLab v1.x, perform poorly on **Natural Graphs**.



Natural Graphs

Graphs derived from natural phenomena

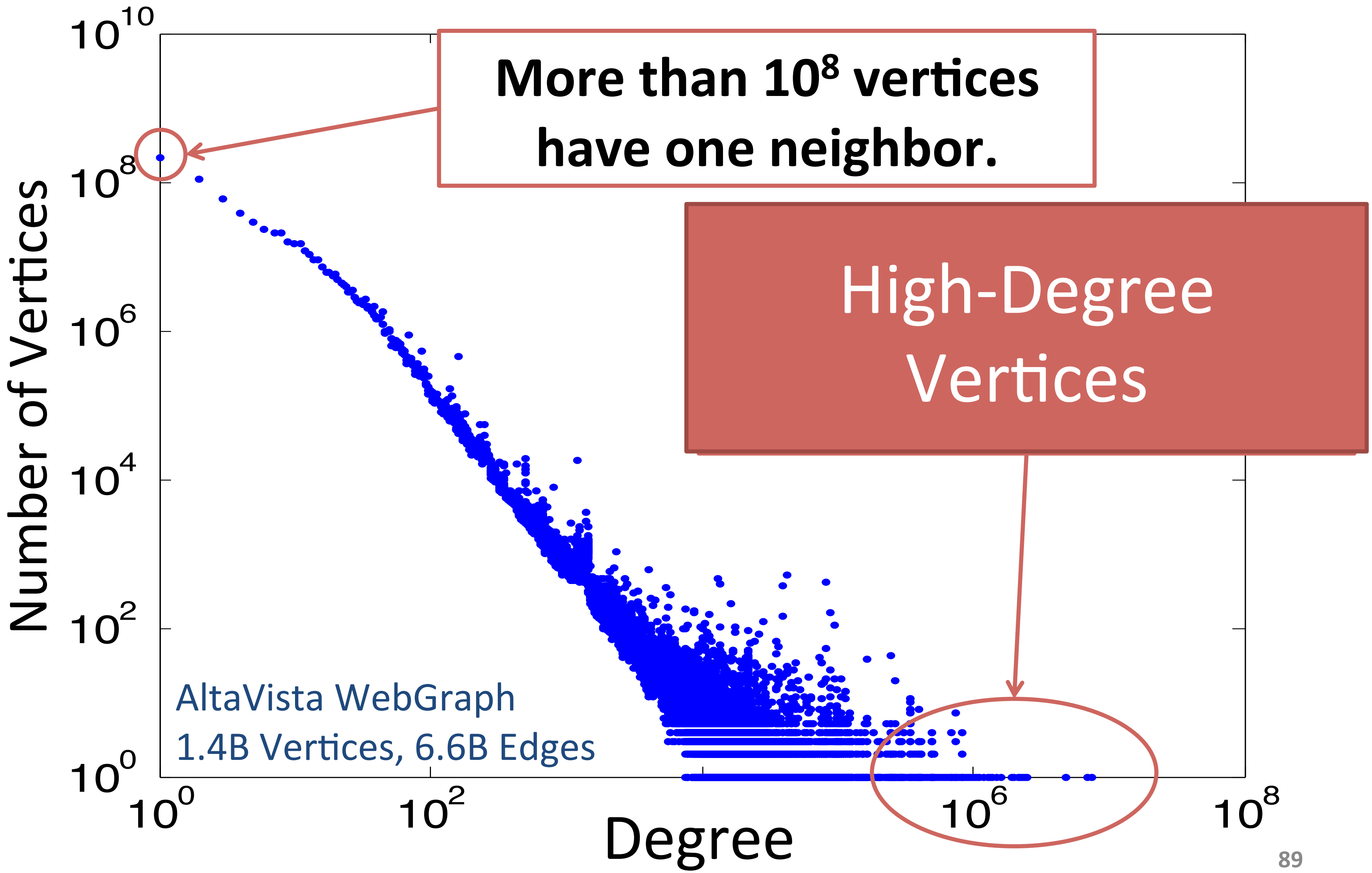
Properties of Natural Graphs



Power-Law Degree Distribution

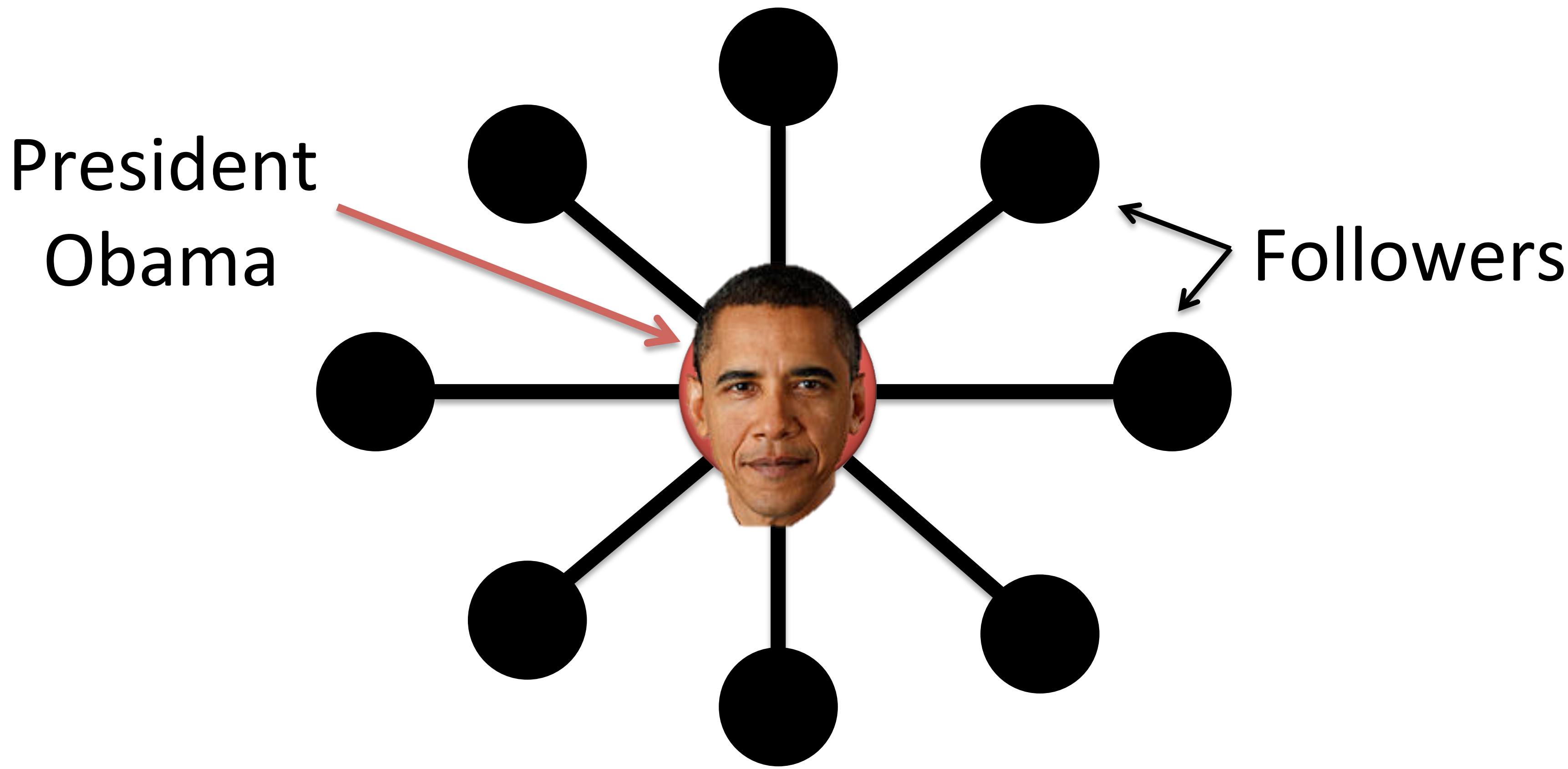
Reference: Zipf, Power-Laws and Pareto: A Ranking Tutorial, by L. Adamic, http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html

Power-Law Degree Distribution

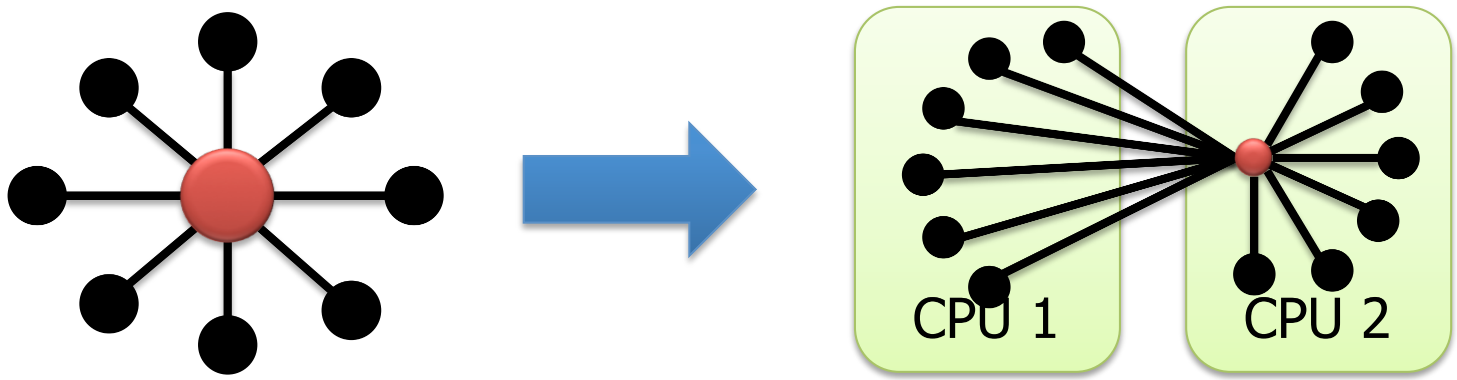


Power-Law Degree Distribution

"Star Like" Motif



Power-Law Graphs are **Difficult to Partition**



- Power-Law graphs do not have **low-cost** balanced cuts *[Leskovec et al. 08, Lang 04]*
- Traditional graph-partitioning algorithms perform poorly on Power-Law Graphs.
  [Abou-Rjeili et al. 06]

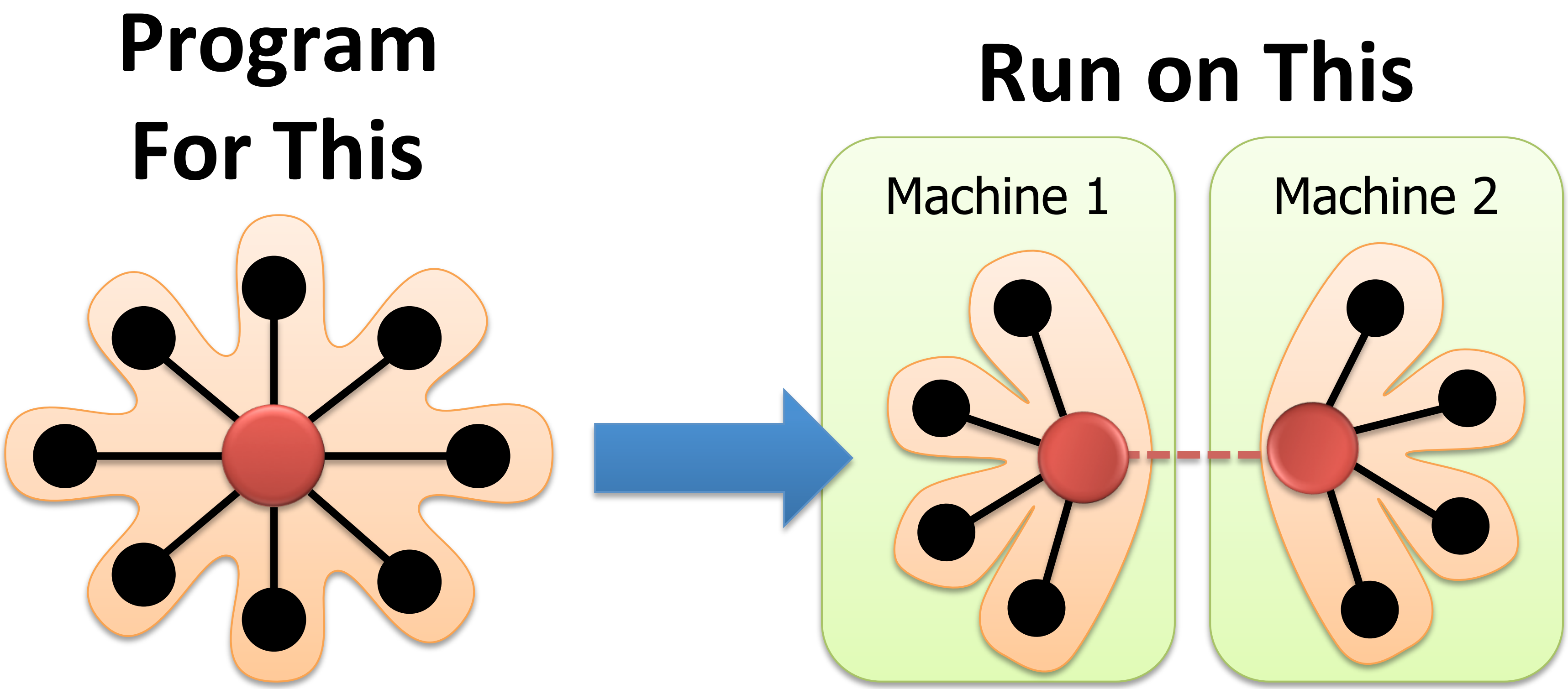
Properties of Natural Graphs

High-degree Vertices

Power-Law Degree Distribution

Low Quality Partition

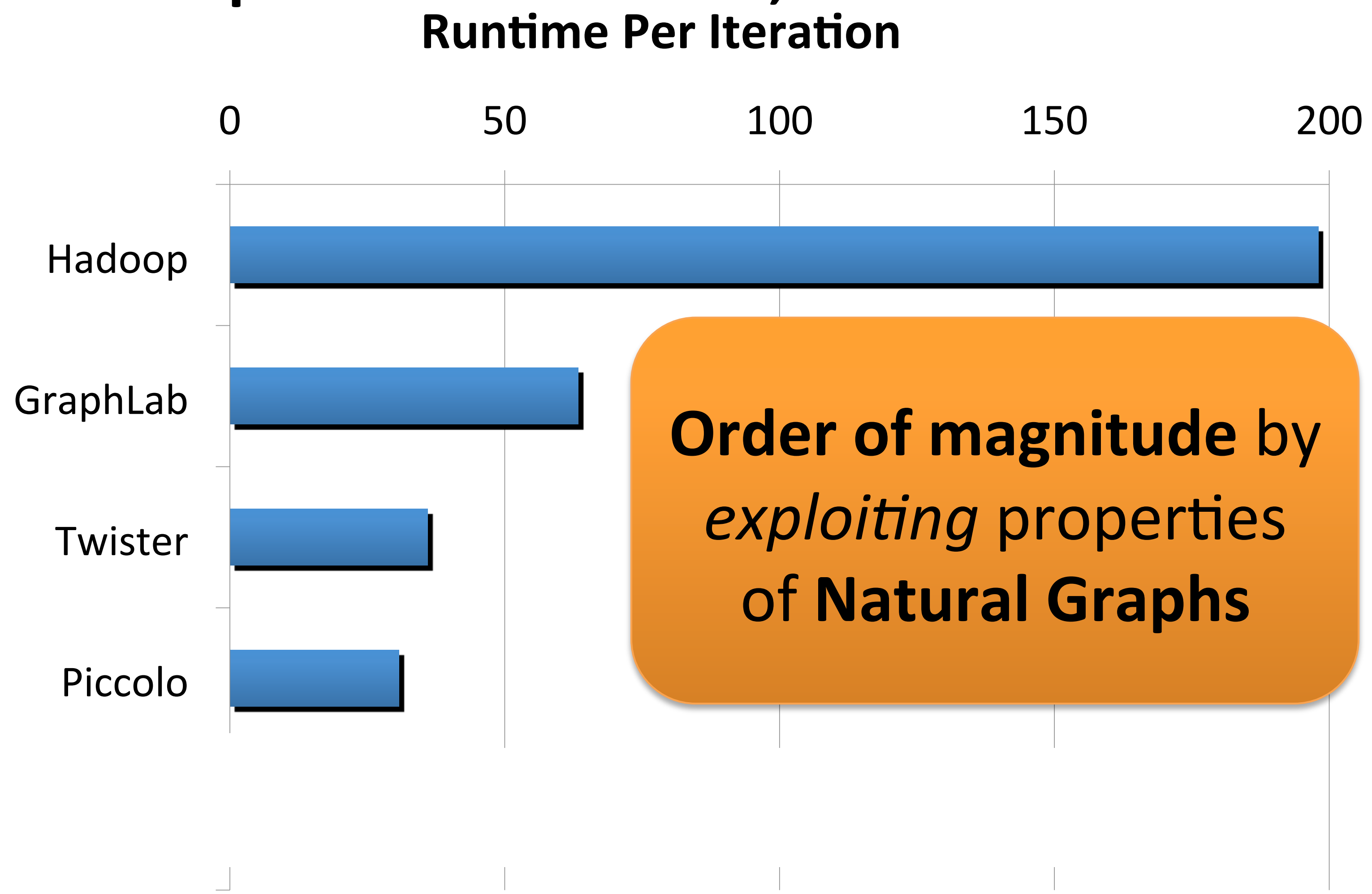
PowerGraph



- Split **High-Degree** vertices
- **New Abstraction → *Equivalence* on *Split Vertices***

PageRank on Twitter Follower Graph

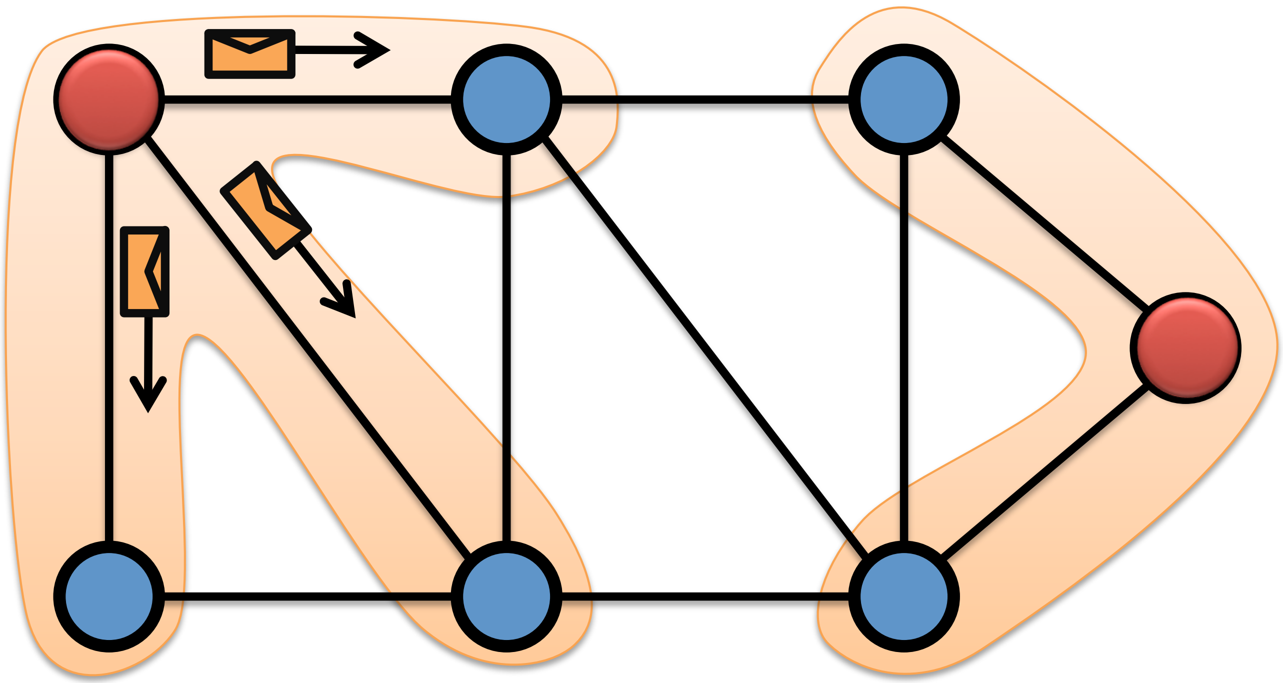
Natural Graph with 40M Users, 1.4 Billion Links



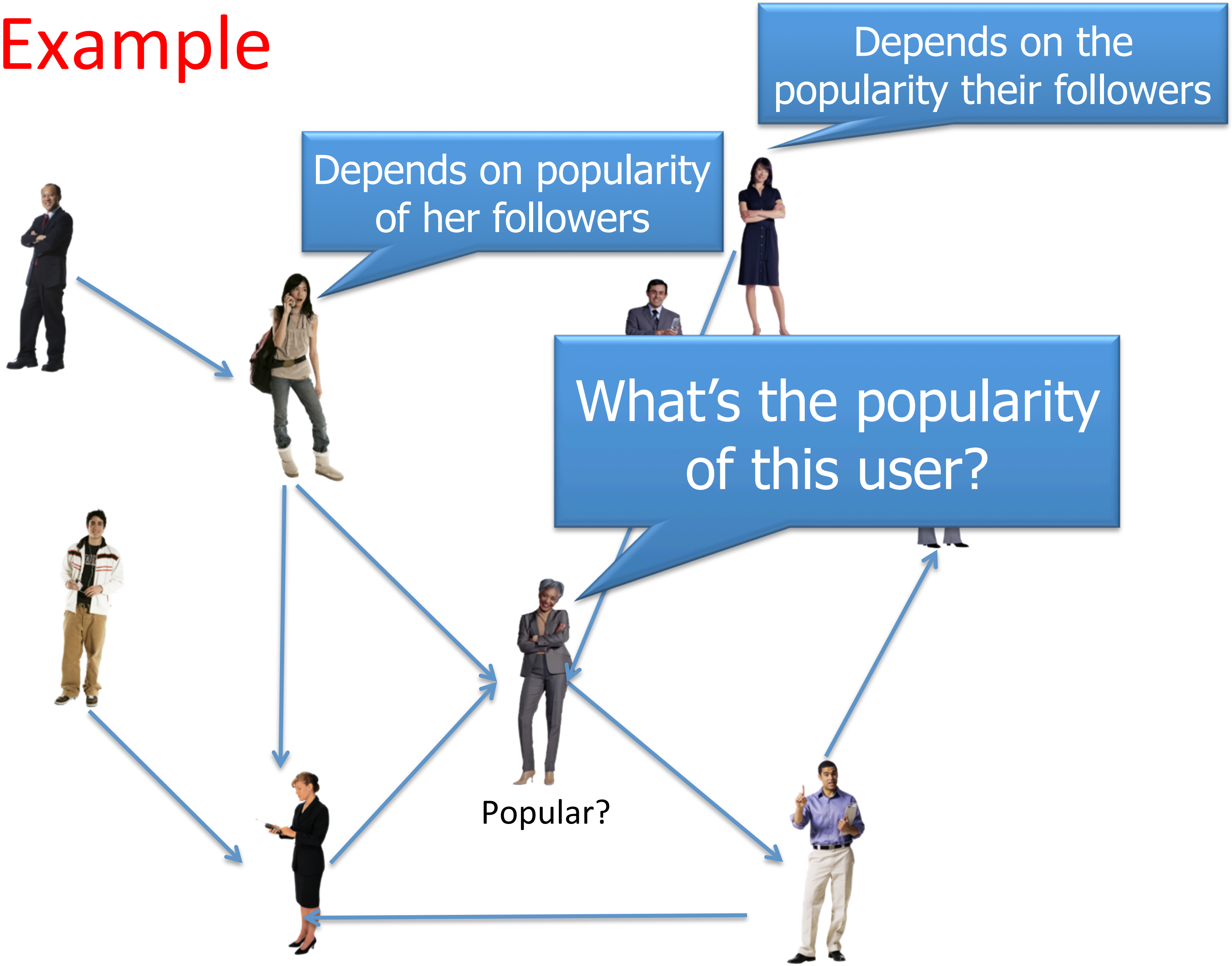
Hadoop results from [Kang et al. '11]
Twister (in-memory MapReduce) [Ekanayake et al. '10]

The **Graph-Parallel** Abstraction

- A user-defined **Vertex-Program** runs on each vertex
- **Graph** constrains **interaction** along edges
  - Using **messages** (e.g. **Pregel** [PODC'09, SIGMOD'10])
  - Through **shared state** (e.g., **GraphLab** [UAI'10, VLDB'12])
- **Parallelism**: run multiple vertex programs simultaneously



Example


Depends on the popularity their followers
Depends on popularity of her followers
What's the popularity of this user?
Popular?

PageRank Algorithm

$$R[i] = 0.15 + \sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]$$

Rank of user *i*

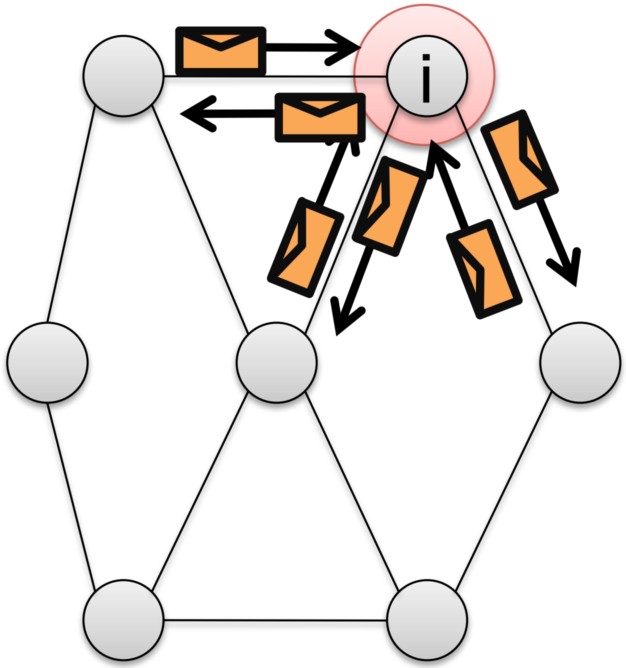
Weighted sum of neighbors' ranks

- Update ranks in parallel
- Iterate until convergence

The Pregel Abstraction

Vertex-Programs interact by sending **messages**.

```
Pregel_PageRank(i, messages) :
  // Receive all the messages
  total = 0
  foreach( msg in messages) :
    total = total + msg

  // Update the rank of this vertex
  R[i] = 0.15 + total

  // Send new messages to neighbors
  foreach(j in out_neighbors[i]) :
    Send  msg(R[i] * w_ij) to vertex j
```

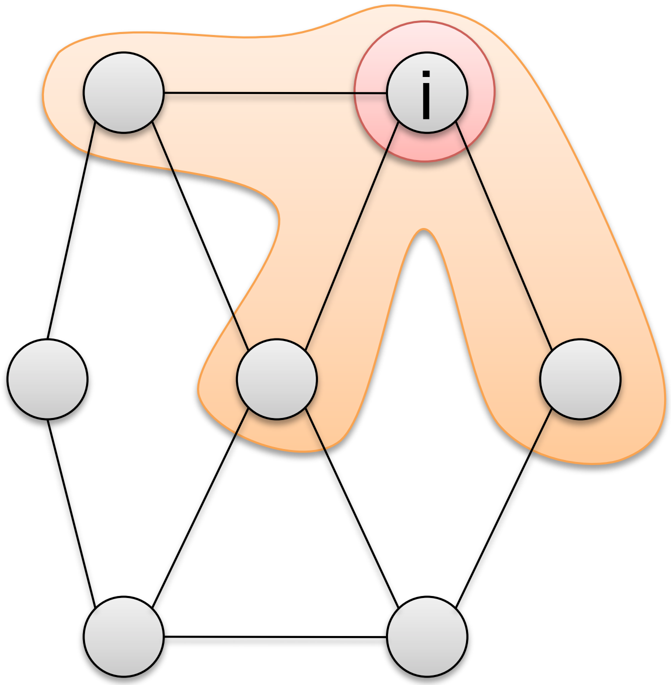


Malewicz et al. [PODC'09, SIGMOD'10]

The GraphLab Abstraction

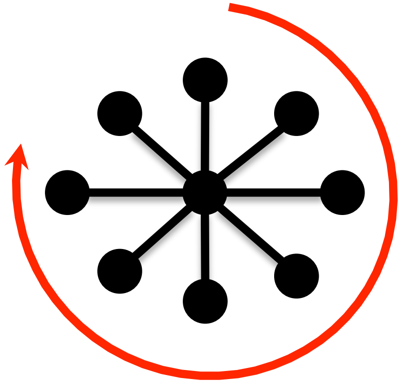
Vertex-Programs directly **read** the neighbors state

```
GraphLab_PageRank(i)
  // Compute sum over neighbors
  total = 0
  foreach( j in in_neighbors(i)):
    total = total + R[j] * w_ji

  // Update the PageRank
  R[i] = 0.15 + total

  // Trigger neighbors to run again
  if R[i] not converged then
    foreach( j in out_neighbors(i)):
      signal vertex-program on j
```

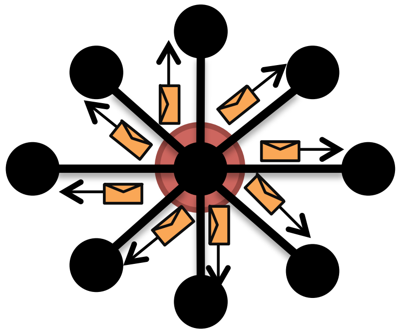


Low et al. [UAI'10, VLDB'12]

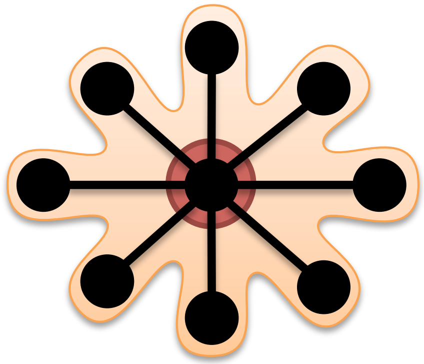
Challenges of High-Degree Vertices



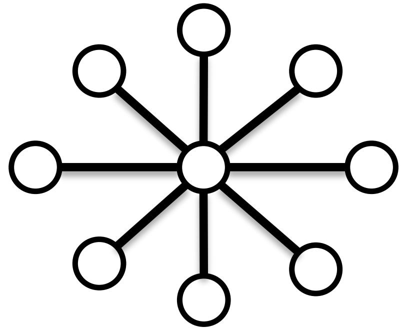
Sequentially process edges



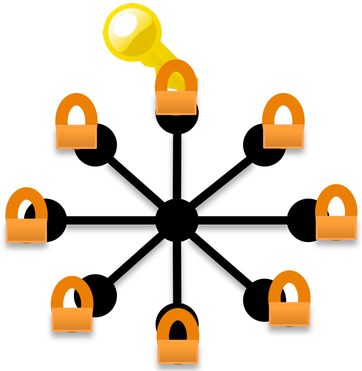
Sends many messages (Pregel)



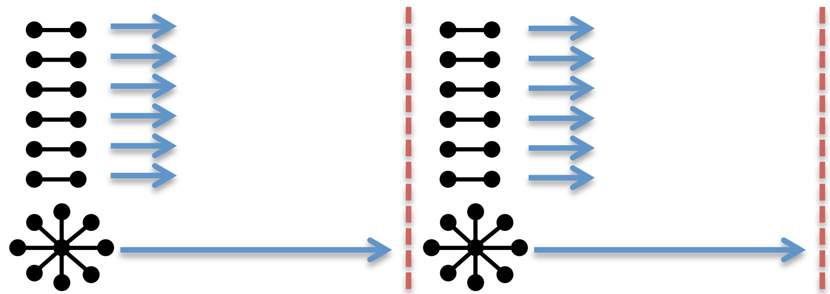
Touches a large fraction of graph (GraphLab)



Edge meta-data too large for single machine



Asynchronous Execution requires heavy locking (GraphLab)

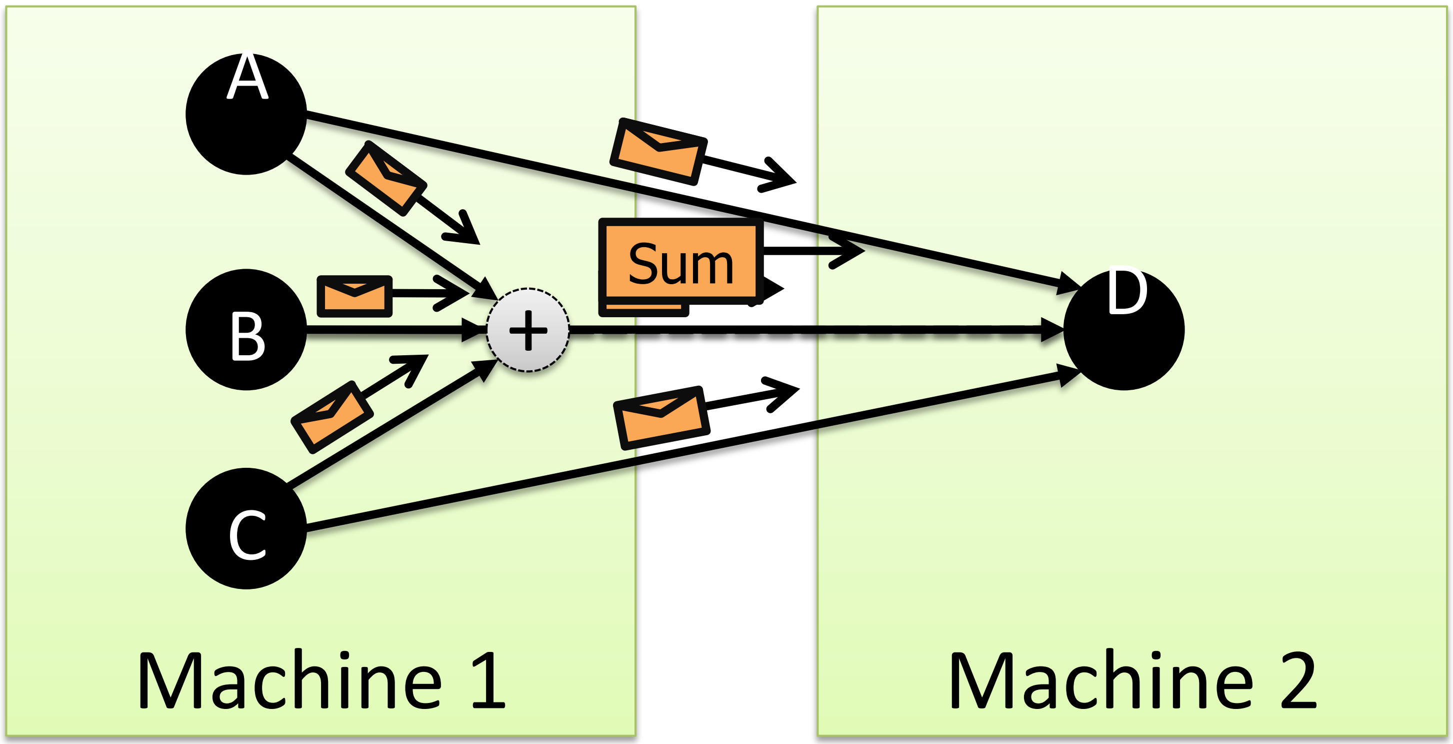


Synchronous Execution prone to stragglers (Pregel)

Communication Overhead for High-Degree Vertices

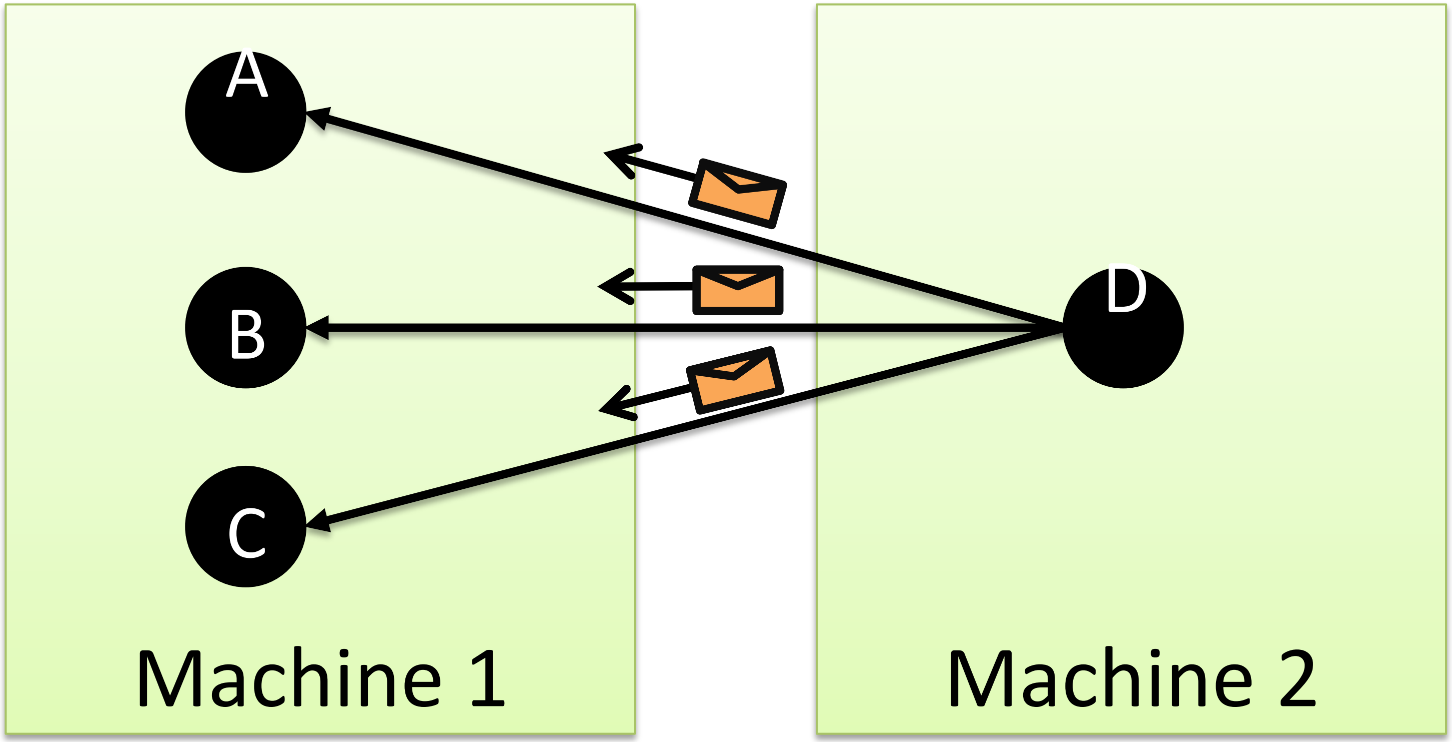
Fan-In vs. Fan-Out

Pregel **Message Combiners** on **Fan-In**



- User defined **commutative associative** (+) message operation:

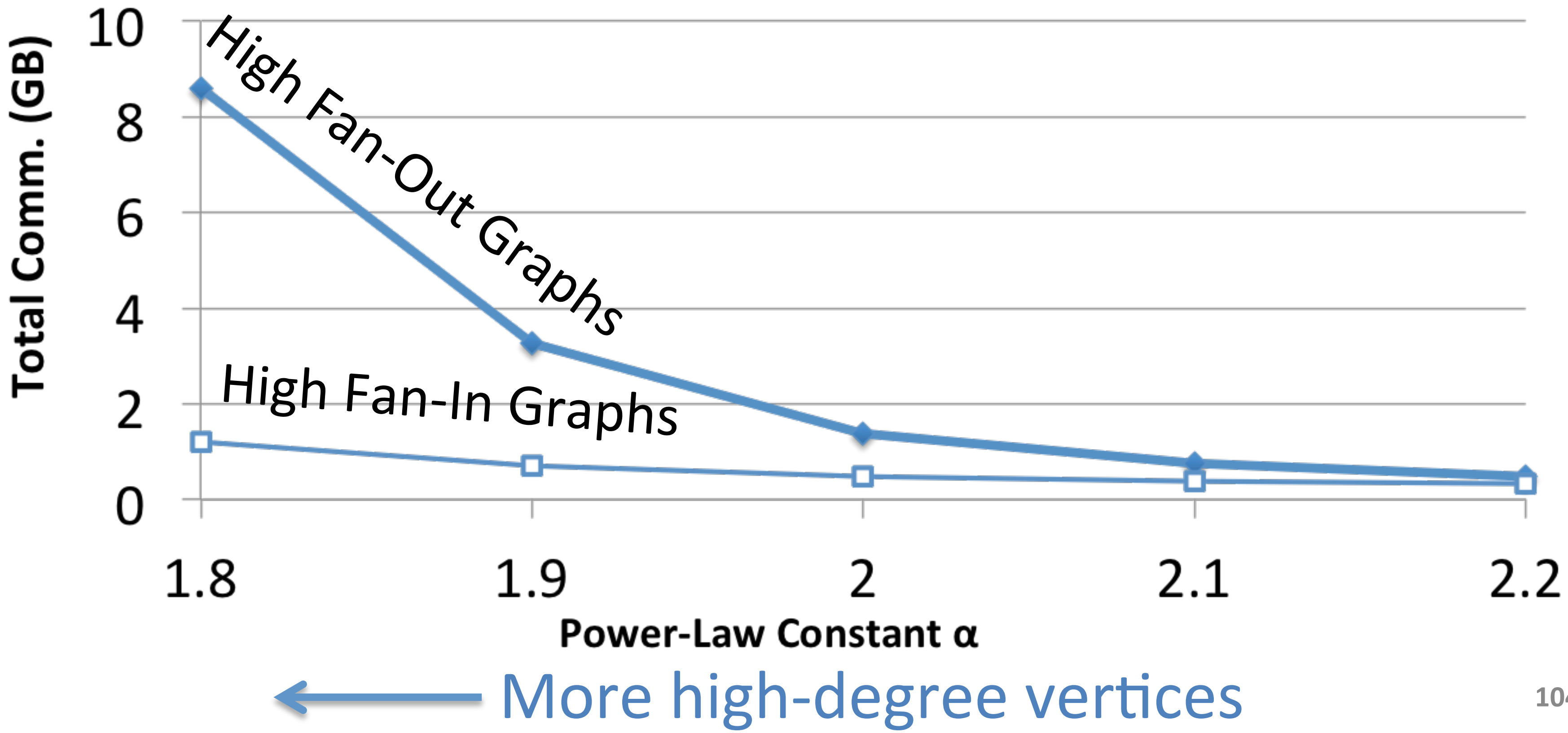
Pregel Struggles with **Fan-Out**



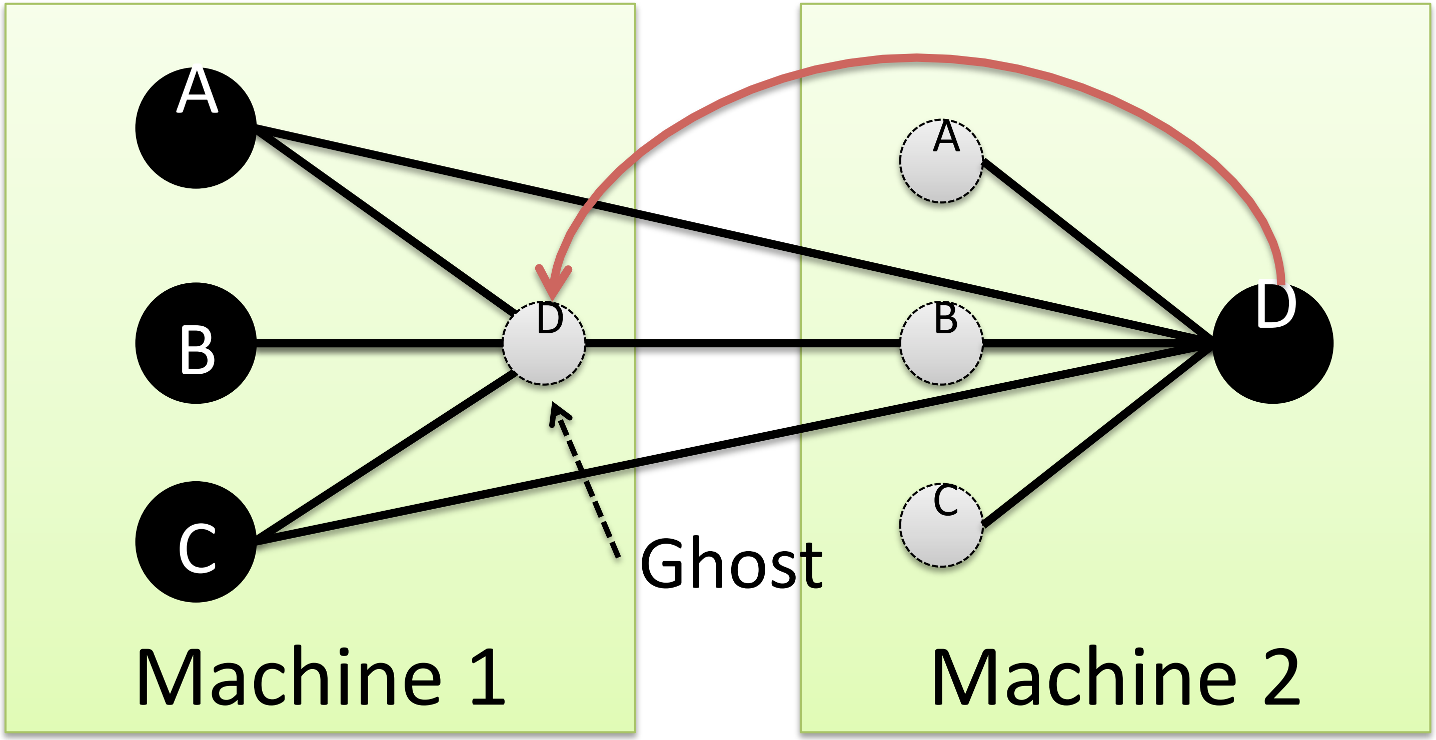
- **Broadcast** sends many copies of the same message to the same machine!

Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
  - Piccolo was used to simulate Pregel with combiners

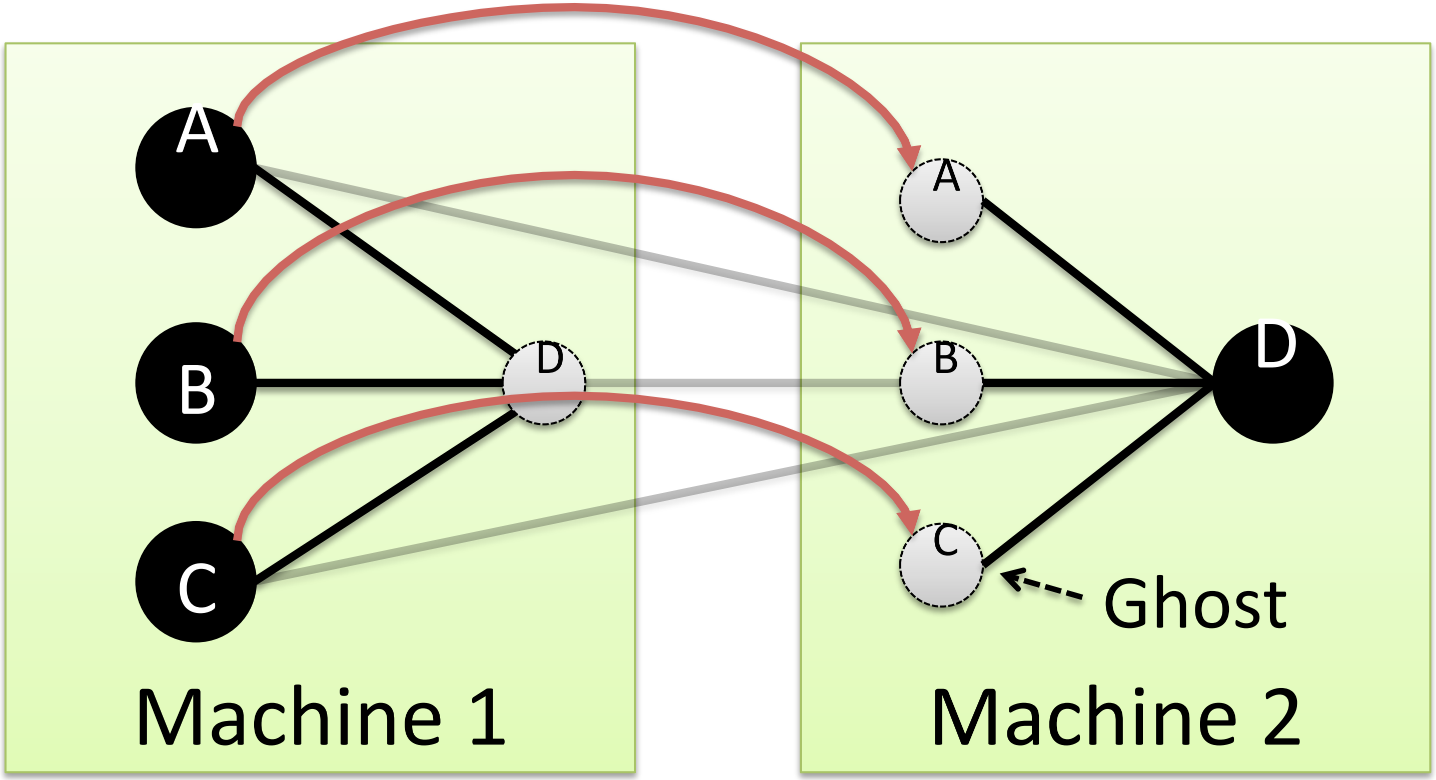


GraphLab Ghosting



- Changes to master are synced to ghosts

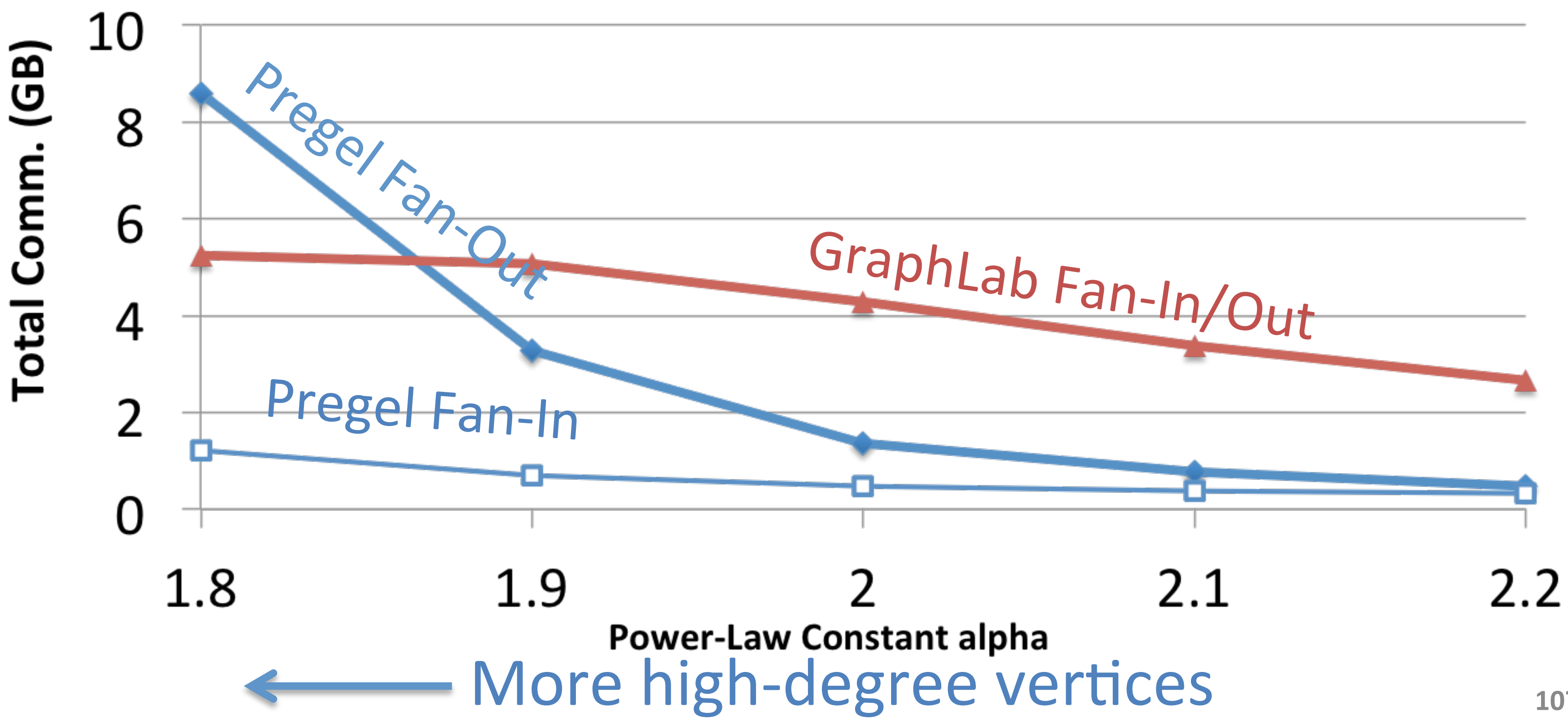
GraphLab Ghosting



- Changes to **neighbors** of **high degree vertices** creates substantial network traffic

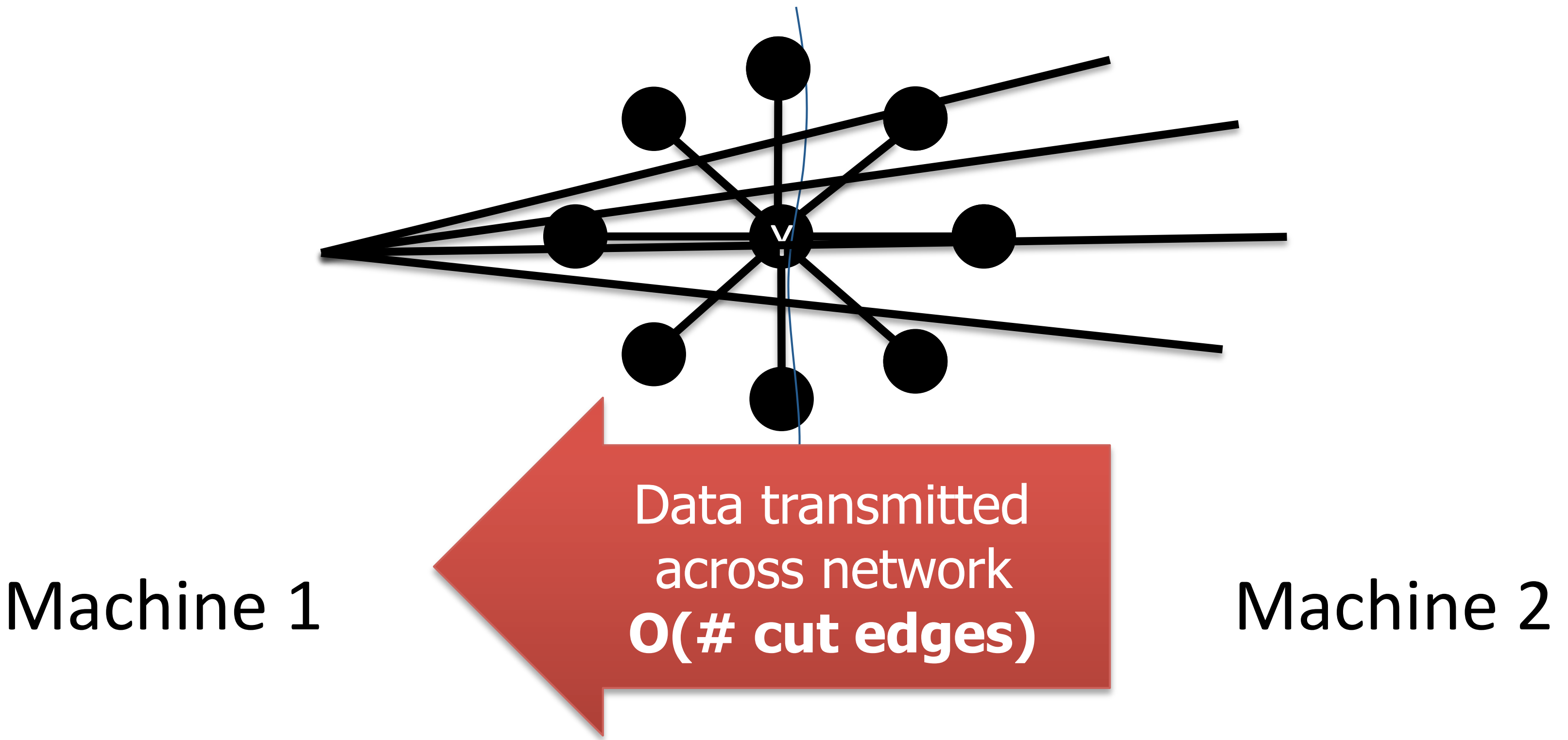
Fan-In and Fan-Out Performance

- PageRank on synthetic Power-Law Graphs
- GraphLab is **undirected**



Graph Partitioning

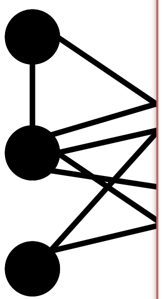
- Graph parallel abstractions rely on partitioning:
  - Minimize communication
  - Balance computation and storage



Random Partitioning

- Both GraphLab and Pregel resort to **random** (hashed) partitioning on **natural graphs**

$$\mathbb{E}\left[\frac{|Edges\ Cut|}{|E|}\right] = 1 - \frac{1}{p}$$



10 Machines → 90% of edges cut
100 Machines → 99% of edges cut!

In Summary

GraphLab and **Pregel** are not well suited for **natural graphs**

- Challenges of **high-degree vertices**
- Low quality **partitioning**

PowerGraph

- **GAS Decomposition**: distribute vertex-programs
  - Move computation to data
  - Parallelize **high-degree** vertices

- **Vertex Partitioning:**
  - Effectively distribute large power-law graphs

A **Common Pattern** for Vertex-Programs

```
GraphLab_PageRank(i)
```

```
// Compute sum over neighbors
total = 0
foreach( j in in_neighbors(i)):
  total = total + R[j] * w_ji
```

Gather Information About Neighborhood

```
// Update the PageRank
R[i] = 0.1 + total
```

Update Vertex

```
// Trigger neighbors to run again
if R[i] not converged then
  foreach( j in out_neighbors(i))
    signal vertex-program on j
```

Signal Neighbors & Modify Edge Data

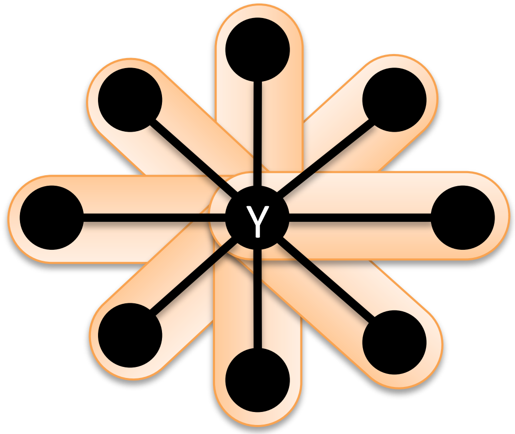
GAS Decomposition

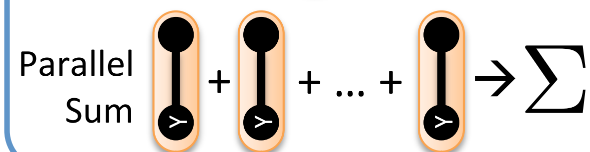
Gather (Reduce)

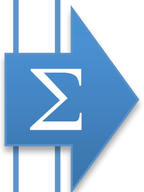
Accumulate information about neighborhood

User Defined:

- **Gather**( ) → Σ
- $\Sigma_1 \oplus \Sigma_2 \rightarrow \Sigma_3$





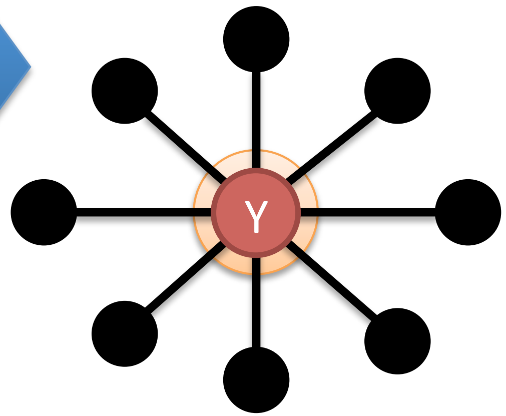


Apply

Apply the accumulated value to center vertex

User Defined:

- **Apply**(Y , Σ) → Y’

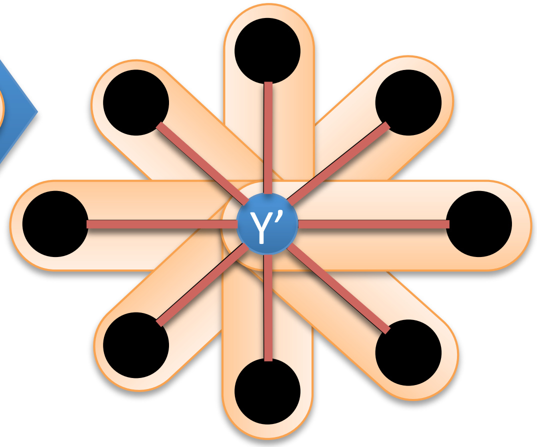


Scatter

Update adjacent edges and vertices.

User Defined:

- **Scatter**( ) →



Update Edge Data & Activate Neighbors

PageRank in PowerGraph

$$R[i] = 0.15 + \boxed{\sum_{j \in \mathrm{Nbrs}(i)} w_{ji} R[j]}$$

PowerGraph_PageRank(i)

Gather(j → i) : return w_{ji} * R[j]

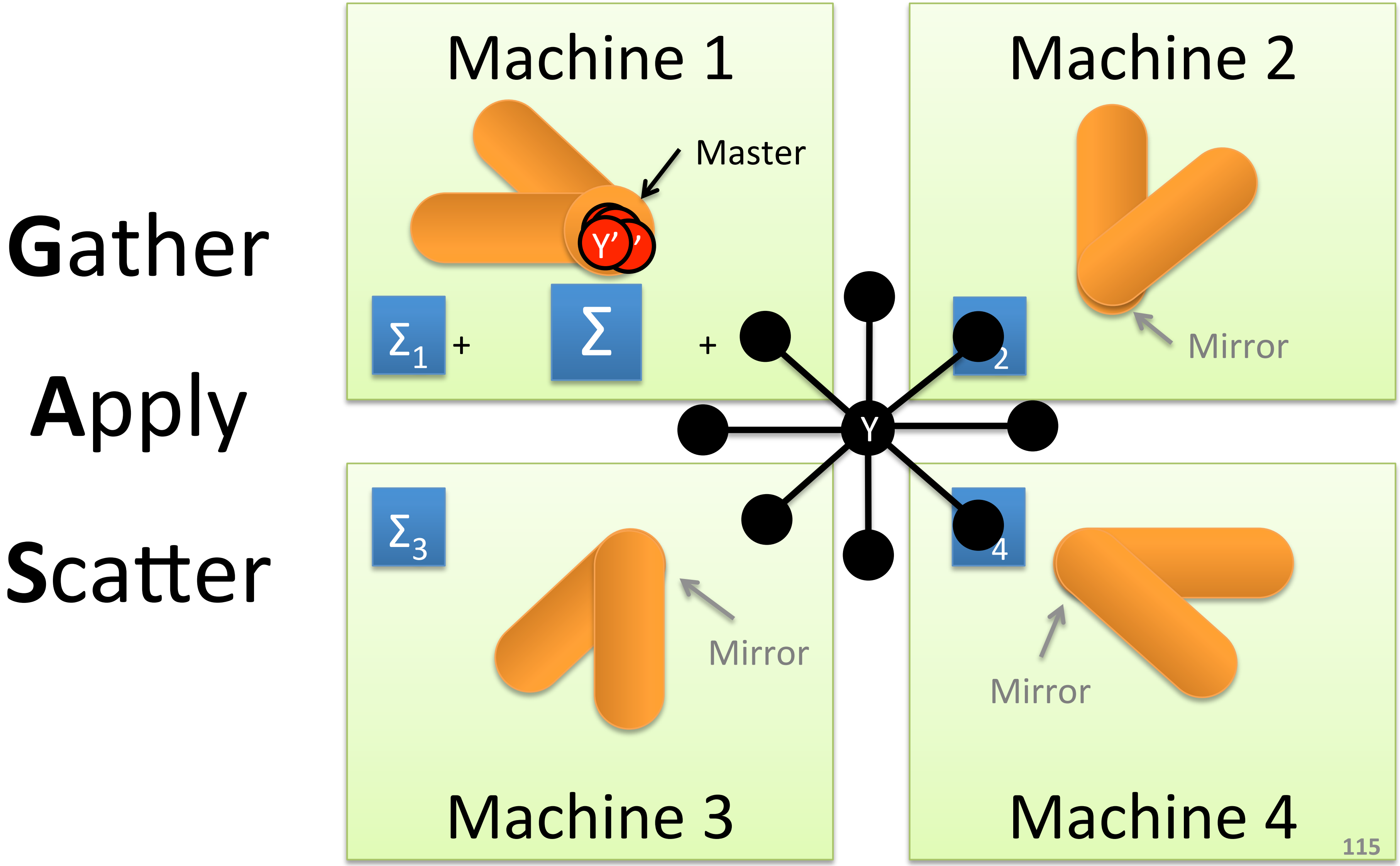
sum(a, b) : return a + b;

Apply(i, Σ) : R[i] = 0.15 + Σ

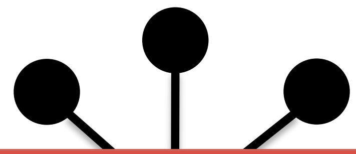
Scatter(i → j) :

if *R[i]* changed then trigger *j* to be **recomputed**

Distributed Execution of a PowerGraph Vertex-Program



Minimizing Communication in PowerGraph



Communication is linear in the number of machines each vertex spans

A **vertex-cut** minimizes machines each vertex spans

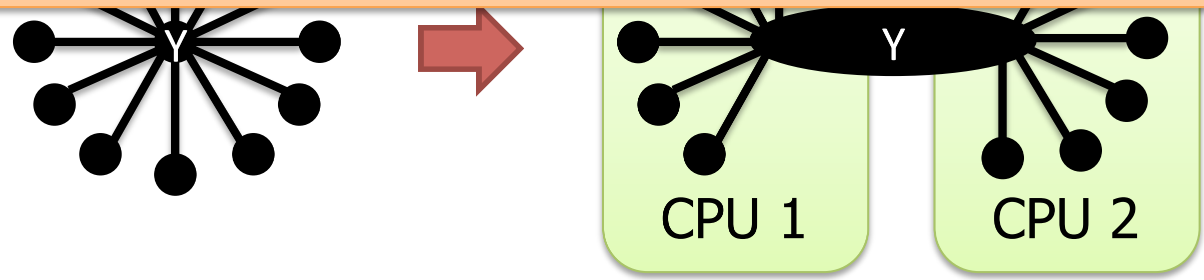
*Percolation theory suggests that power law graphs have **good vertex cuts**. [Albert et al. 2000]*

New Approach to Partitioning

- Rather than cut edges:

New Theorem:

*For **any edge-cut** we can directly construct a vertex-cut which requires **strictly less** communication and storage.*



Must synchronize a **single** vertex

Constructing Vertex-Cuts

- **Evenly** assign **edges** to machines
  - Minimize machines spanned by each vertex

- Assign each edge **as it is loaded**
  - Touch each edge only once

- Propose three **distributed** approaches:
  - ***Random*** *Edge Placement*
  - ***Coordinated Greedy*** *Edge Placement*
  - ***Oblivious Greedy*** *Edge Placement*

Random Edge-Placement

- Randomly assign edges to machines

Machine 1

Machine 2

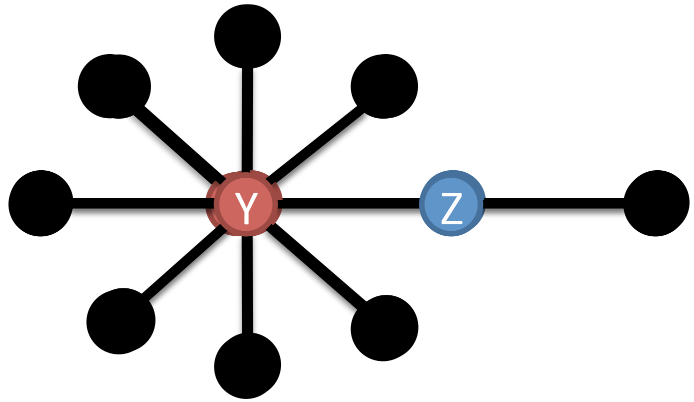
Machine 3

Balanced Vertex-Cut

Y Spans 3 Machines

Z Spans 2 Machines

Not cut!

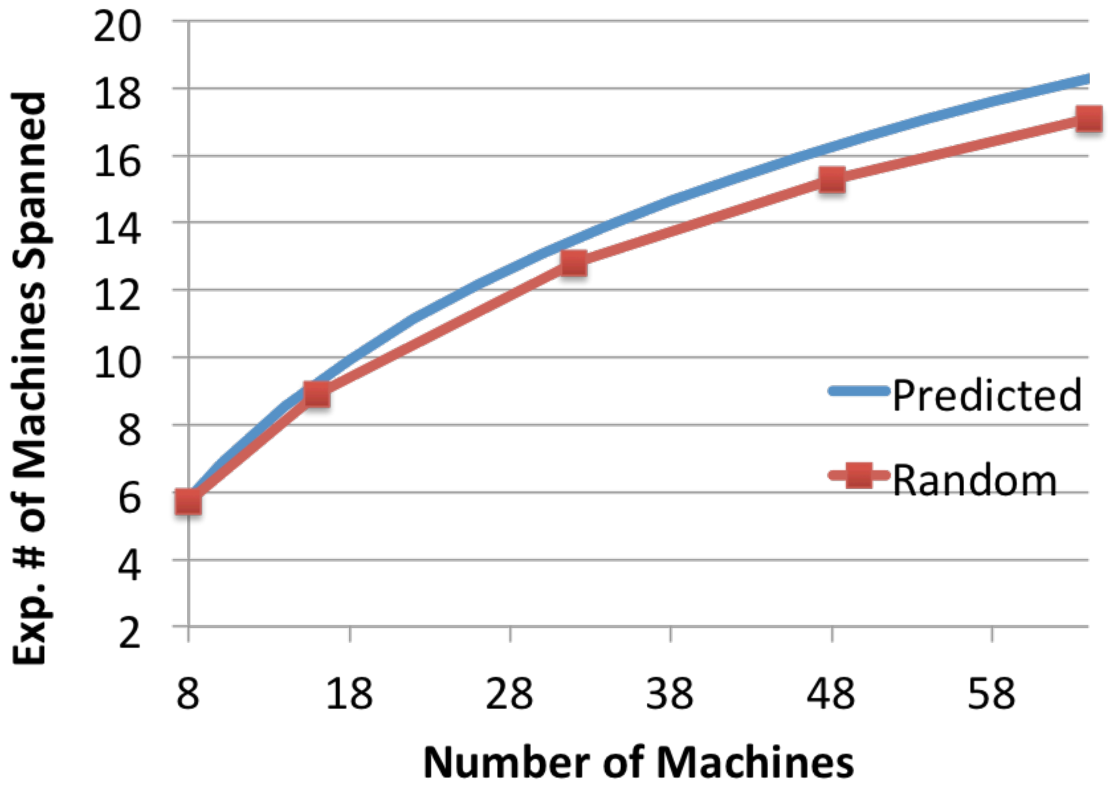


Analysis Random Edge-Placement

- Expected number of machines spanned by a vertex:

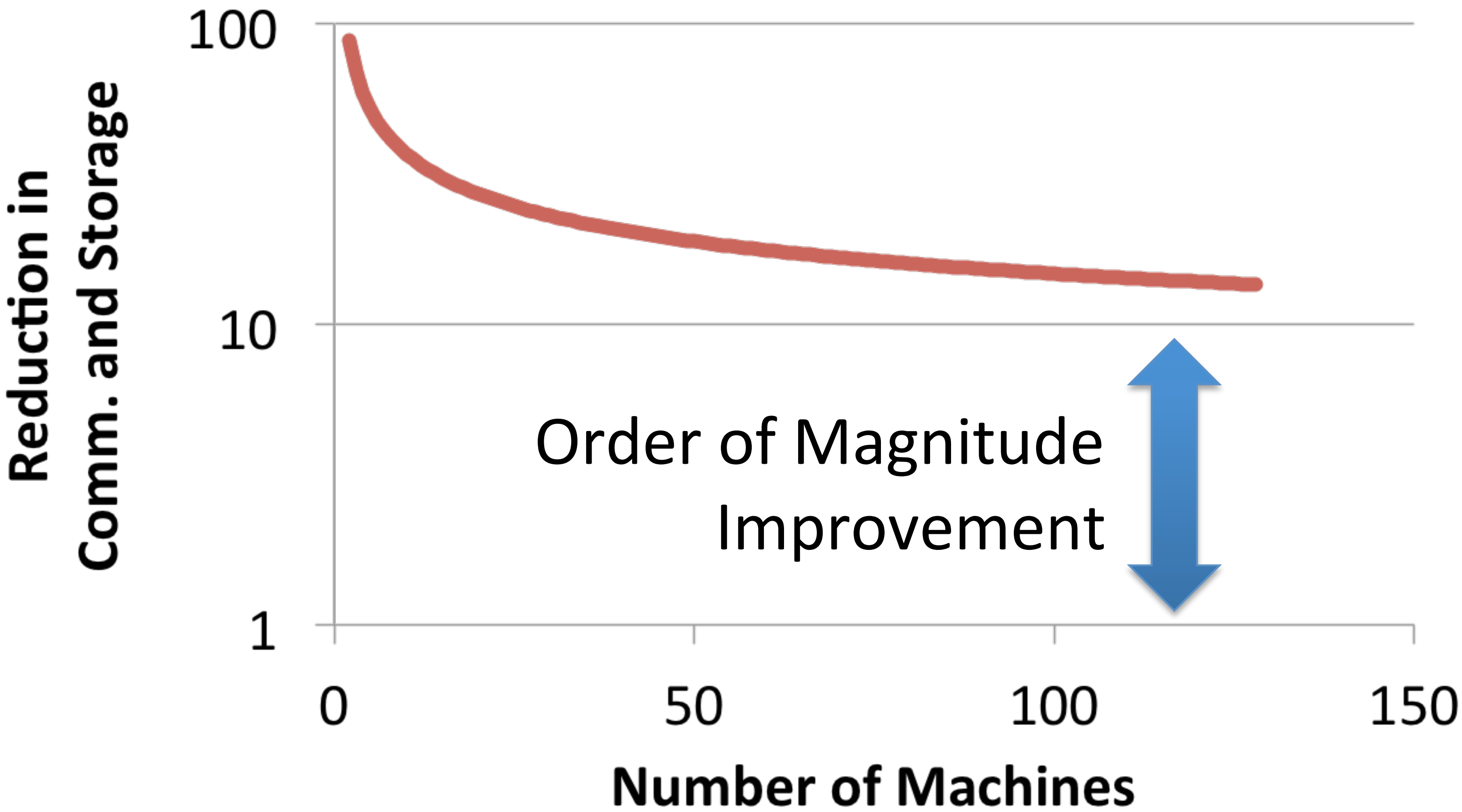
Twitter Follower Graph
41 Million Vertices
1.4 Billion Edges

Accurately Estimate Memory and Comm. Overhead



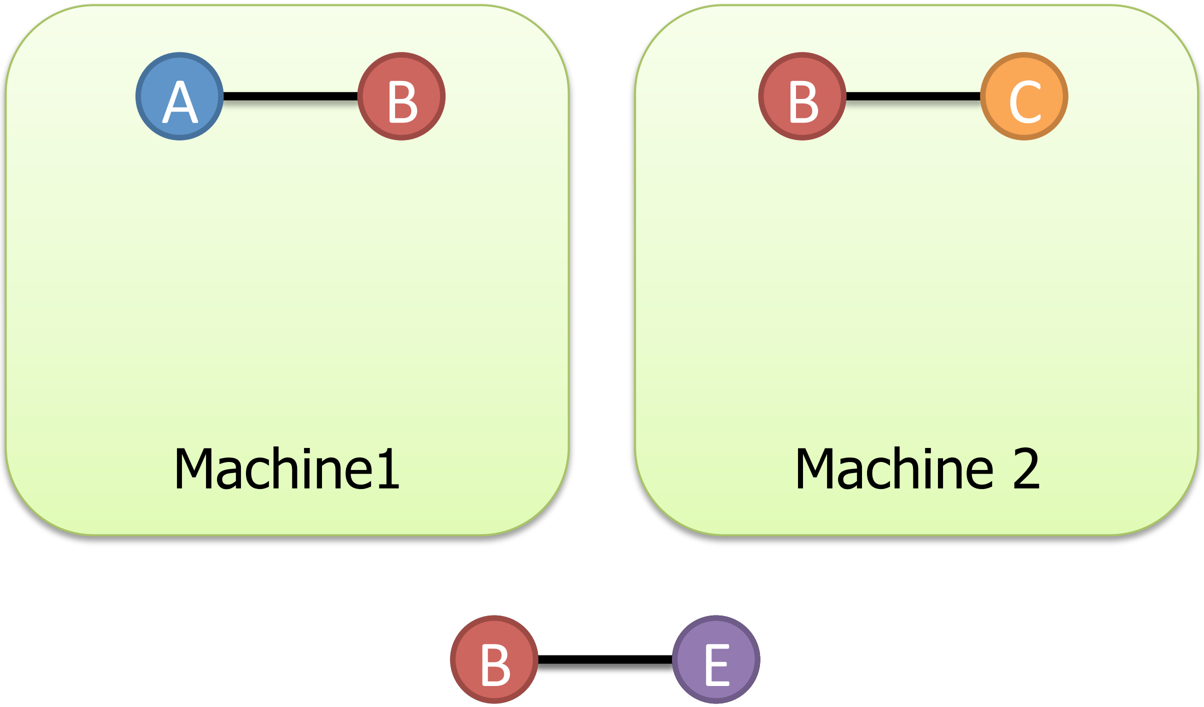
Random Vertex-Cuts vs. Edge-Cuts

- Expected improvement from vertex-cuts:



Greedy Vertex-Cuts

- Place edges on machines which already have the vertices in that edge.

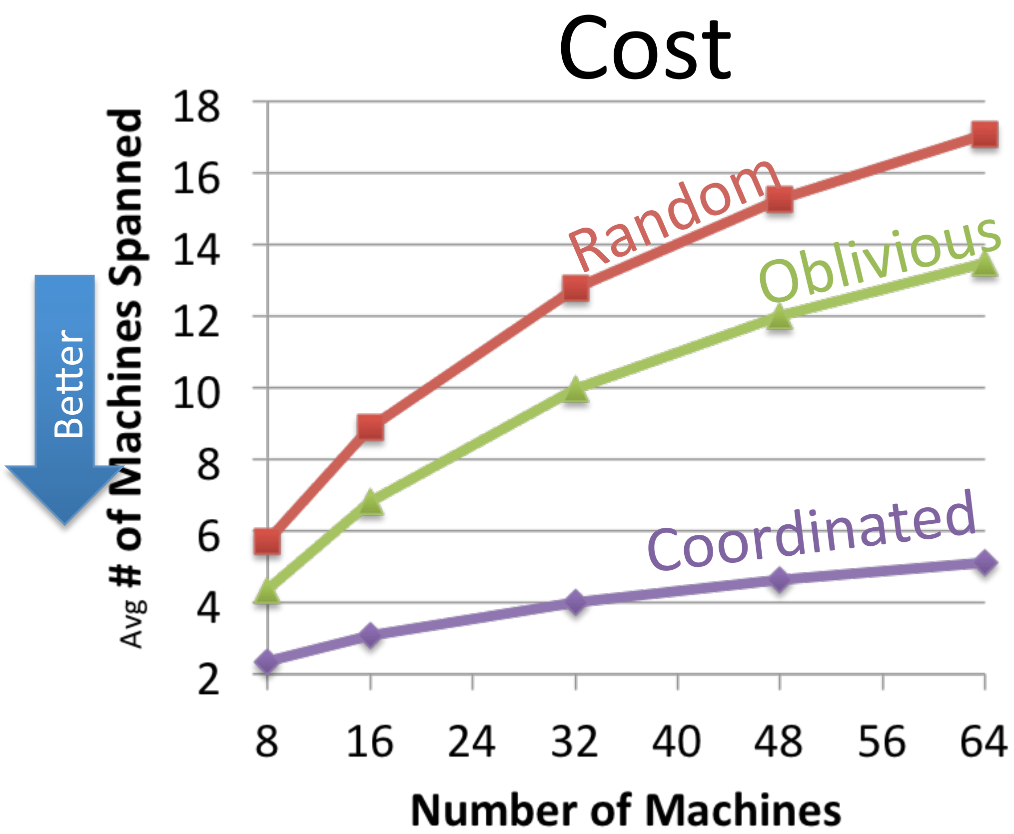


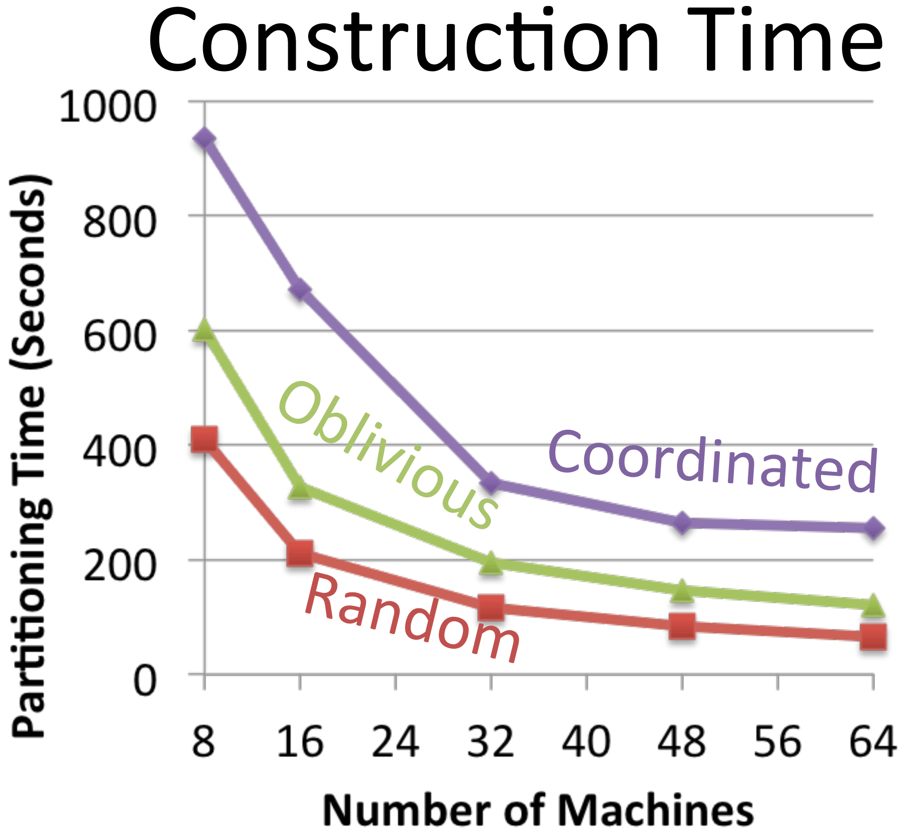
Greedy Vertex-Cuts

- **De-randomization** → greedily minimizes the expected number of machines spanned
- **Coordinated** Edge Placement
  - Requires coordination to place each edge
  - Slower: higher quality cuts
- **Oblivious** Edge Placement
  - Approx. greedy objective without coordination
  - Faster: lower quality cuts

Partitioning Performance

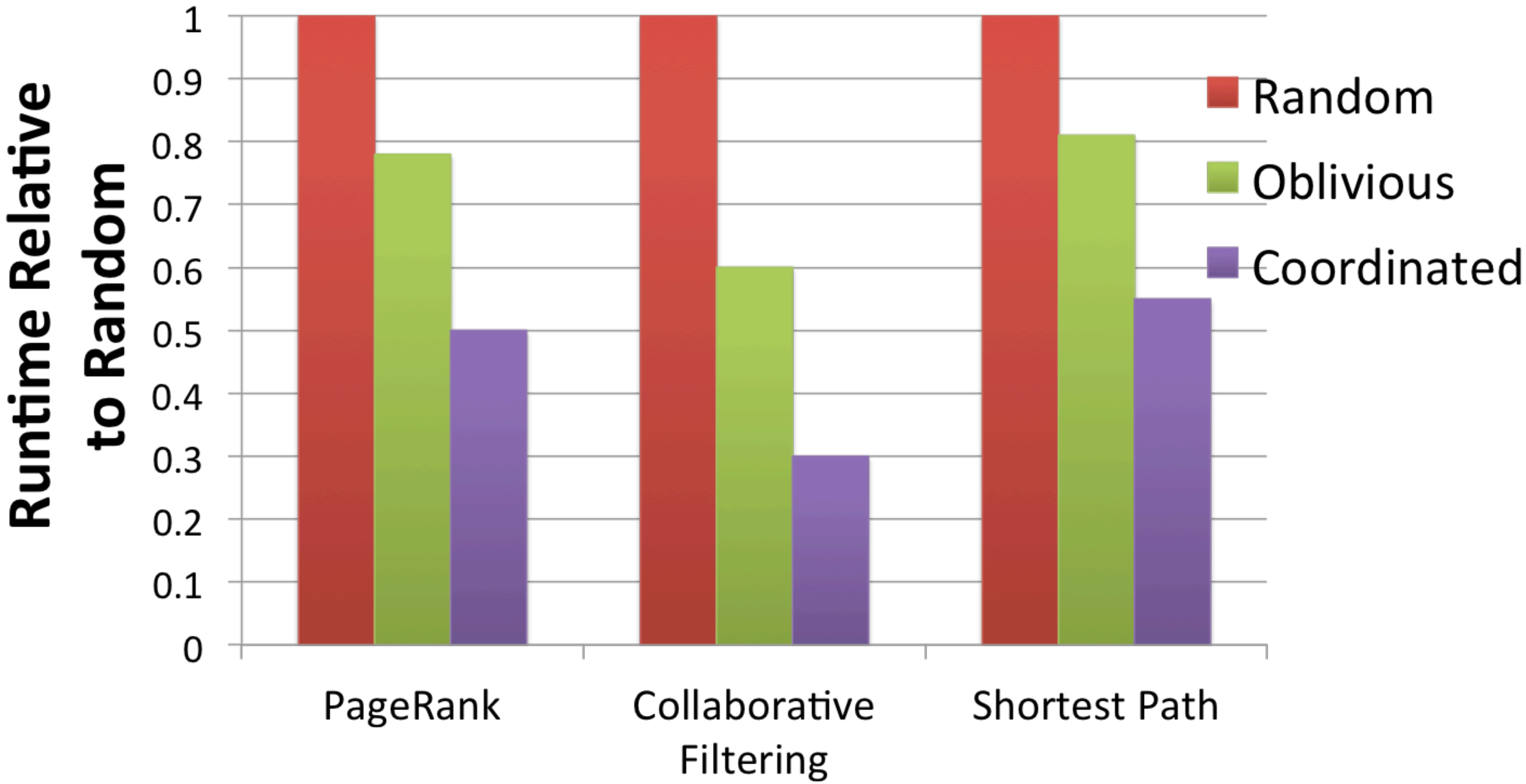
Twitter Graph: 41M vertices, 1.4B edges





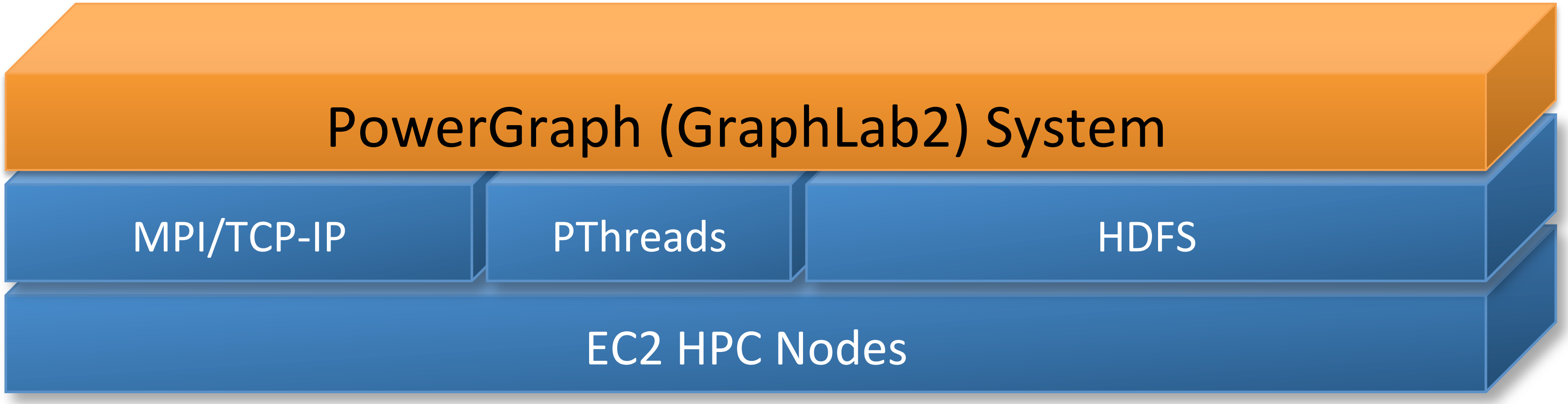
Oblivious balances cost and partitioning time.

Greedy Vertex-Cuts Improve Performance



Greedy partitioning improves computation performance.

PowerGraph System Design



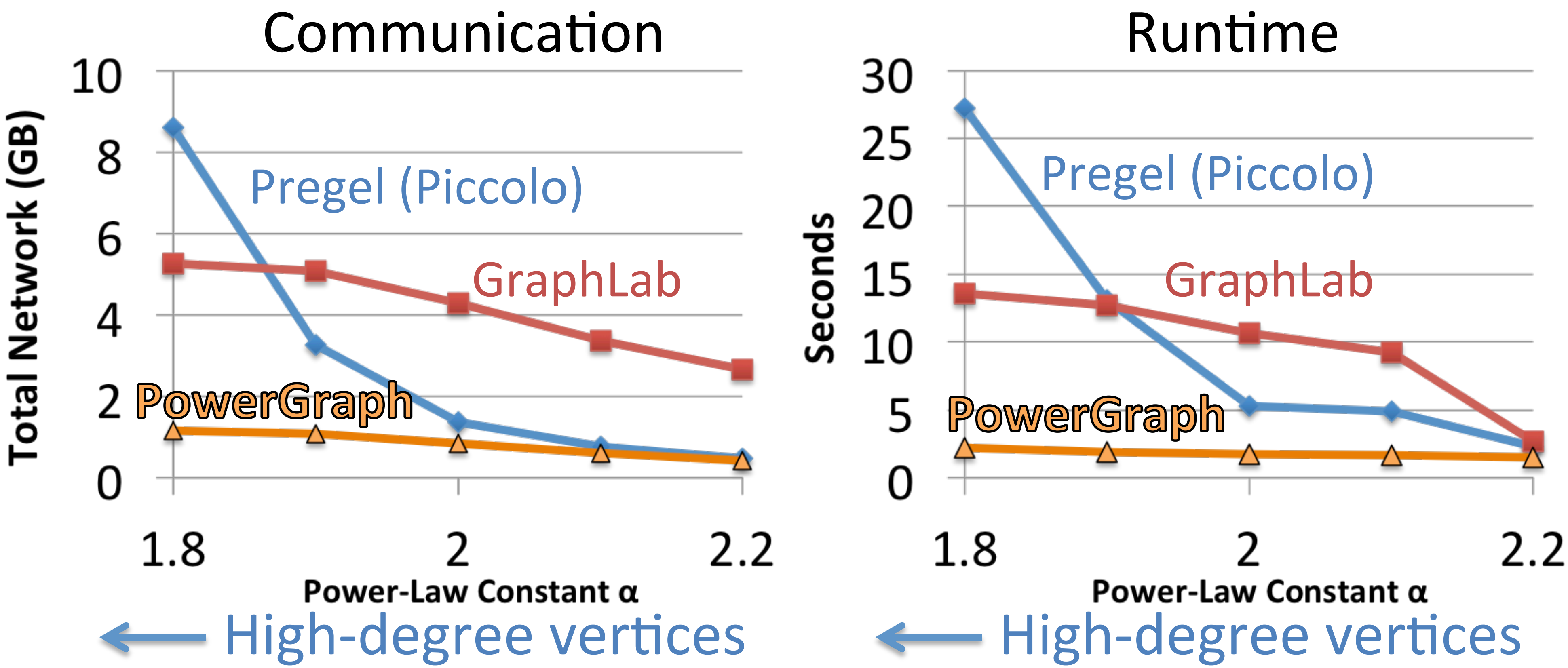
- Implemented as C++ API
- Uses HDFS for Graph Input and Output
- Fault-tolerance is achieved by check-pointing
  – Snapshot time < 5 seconds for twitter network

Implemented Many Algorithms

- **Collaborative Filtering**
  - Alternating Least Squares
  - Stochastic Gradient Descent
  - SVD
  - Non-negative MF
- **Statistical Inference**
  - Loopy Belief Propagation
  - Max-Product Linear Programs
  - Gibbs Sampling
- **Graph Analytics**
  - PageRank
  - Triangle Counting
  - Shortest Path
  - Graph Coloring
  - K-core Decomposition
- **Computer Vision**
  - Image stitching
- **Language Modeling**
  - LDA

Comparison with GraphLab & Pregel

- PageRank on Synthetic Power-Law Graphs:

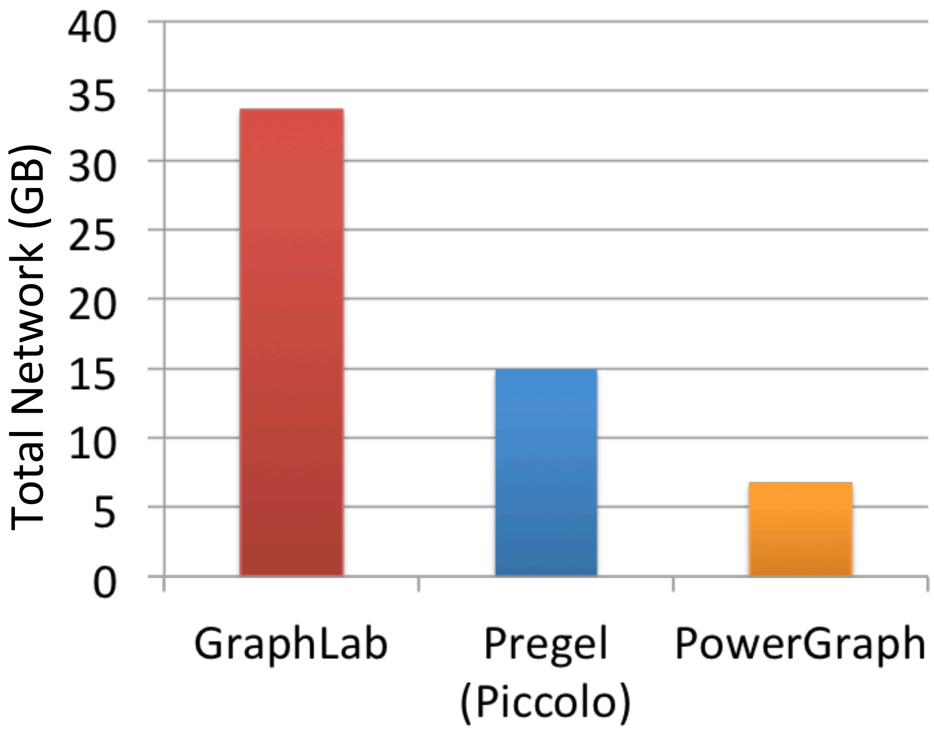


PowerGraph is robust to **high-degree** vertices.

PageRank on the Twitter Follower Graph

Natural Graph with 40M Users, 1.4 Billion Links

Communication



Runtime

Reduces Communication

Runs Faster

32 Nodes x 8 Cores (EC2 HPC cc1.4x)

PowerGraph is Scalable

Yahoo Altavista Web Graph (2002):

One of the largest publicly available web graphs

1.4 Billion Webpages, 6.6 Billion Links

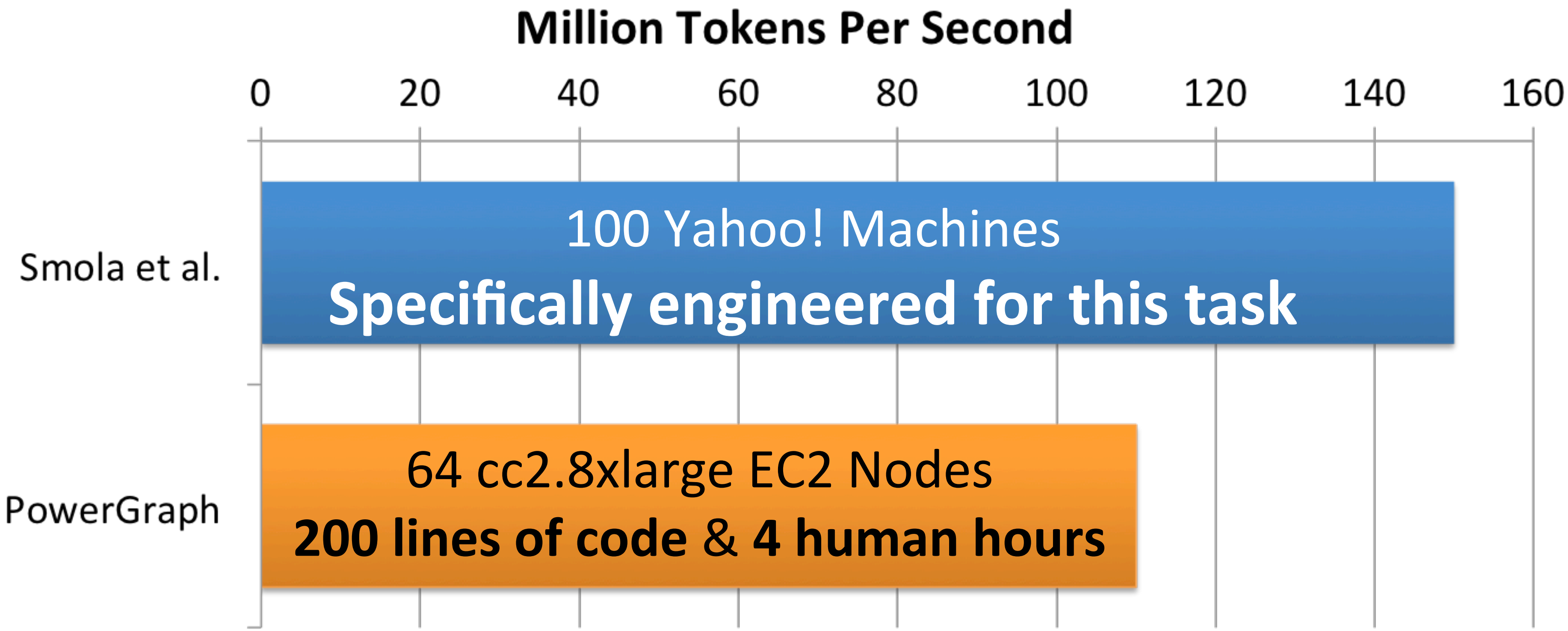
7 Seconds per Iter.

1B links processed per second

30 lines of user code

Topic Modeling

- English language Wikipedia
  - 2.6M Documents, 8.3M Words, 500M Tokens
  - Computationally intensive algorithm

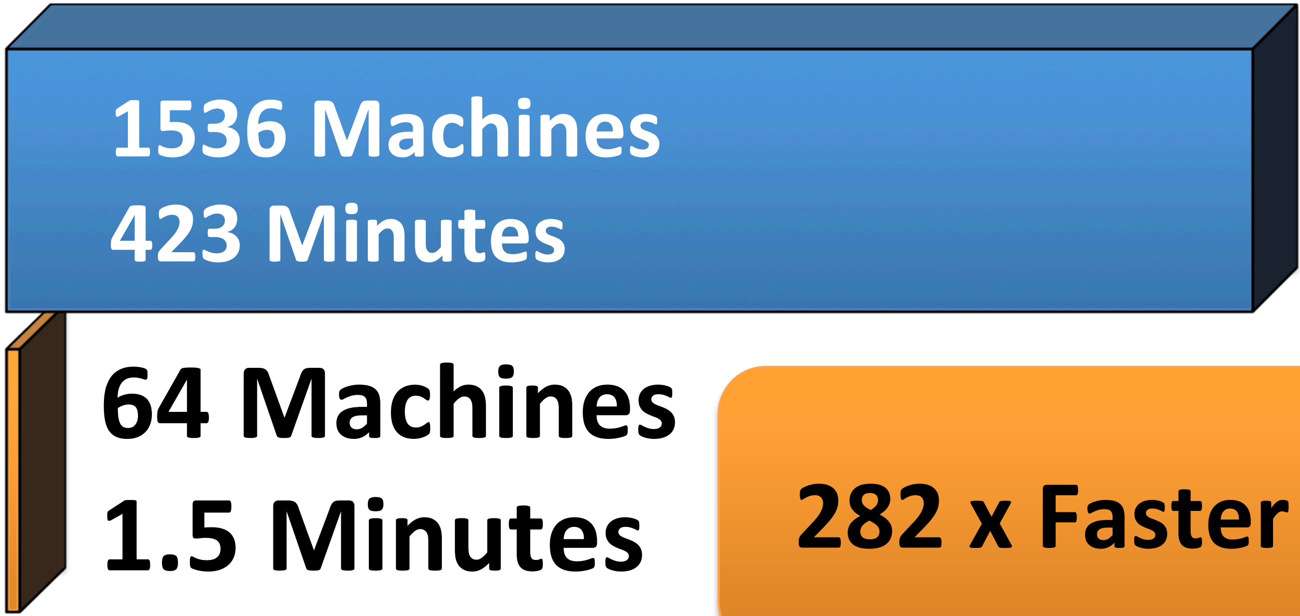


Triangle Counting on The Twitter Graph

Identify individuals with **strong communities.**

Counted: 34.8 Billion Triangles

Hadoop
[WWW’11]



PowerGraph

64 Machines
1.5 Minutes

282 x Faster

Why? **Wrong Abstraction** →
Broadcast O(degree^2) messages per Vertex

S. Suri and S. Vassilvitskii, “Counting triangles and the curse of the last reducer,” WWW’11

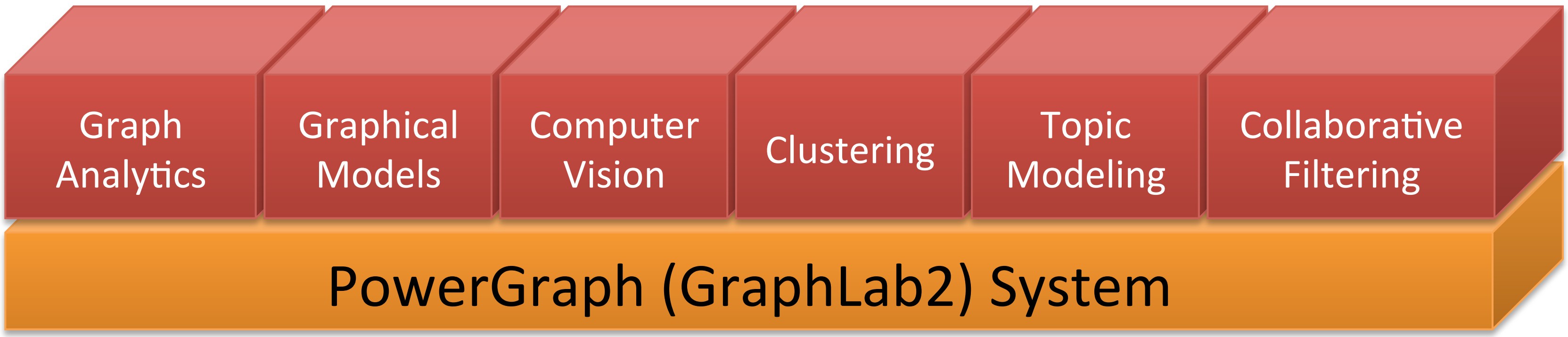
Summary

- *Problem*: Computation on **Natural Graphs** is challenging
  - High-degree vertices
  - Low-quality edge-cuts

- *Solution*: **PowerGraph System**
  - **GAS Decomposition**: split vertex programs
  - **Vertex-partitioning**: distribute natural graphs

- PowerGraph **theoretically** and **experimentally** outperforms existing graph-parallel systems.

Machine Learning and **Data-Mining** Toolkits



PowerGraph

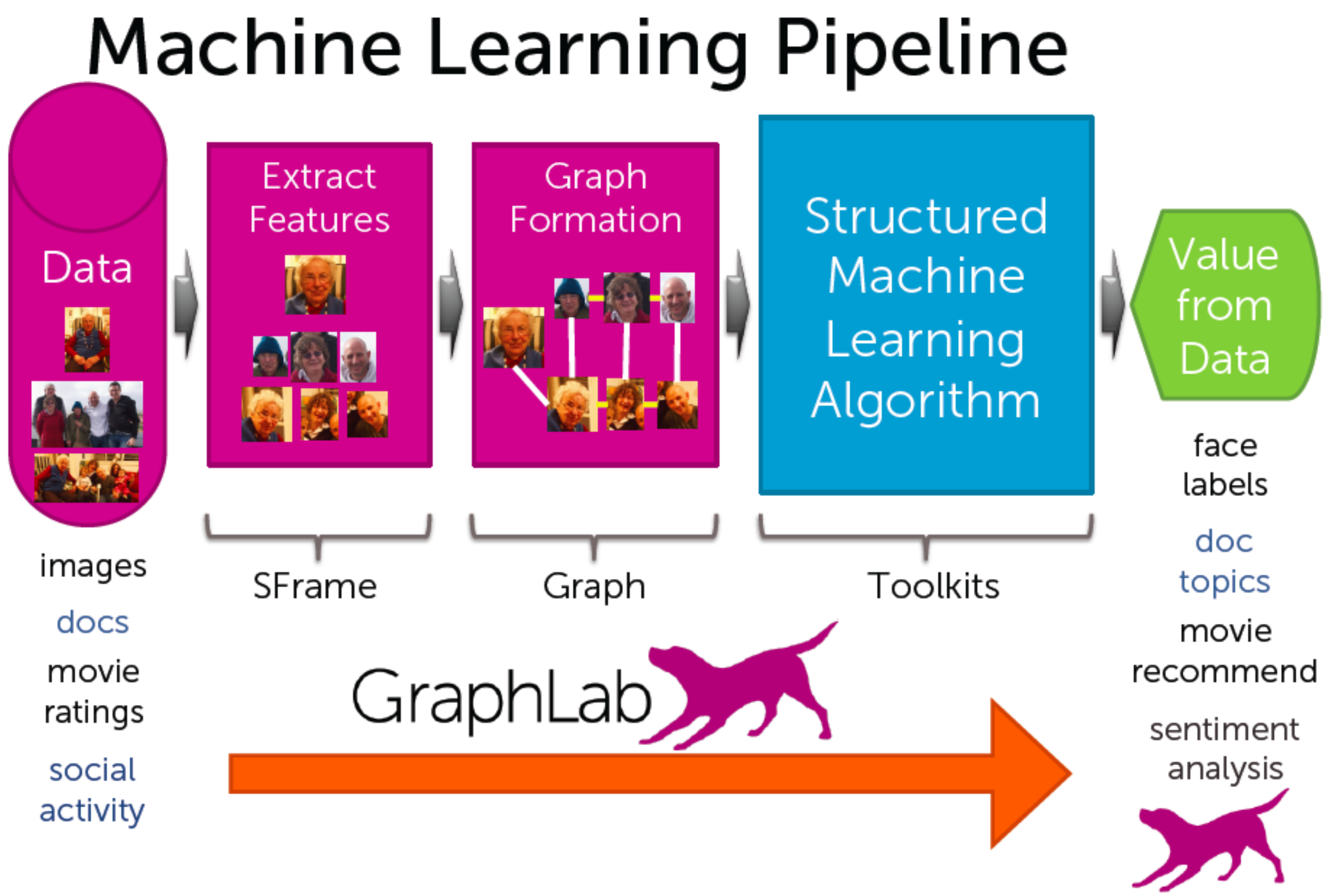
is GraphLab Version 2.1

Apache 2 License

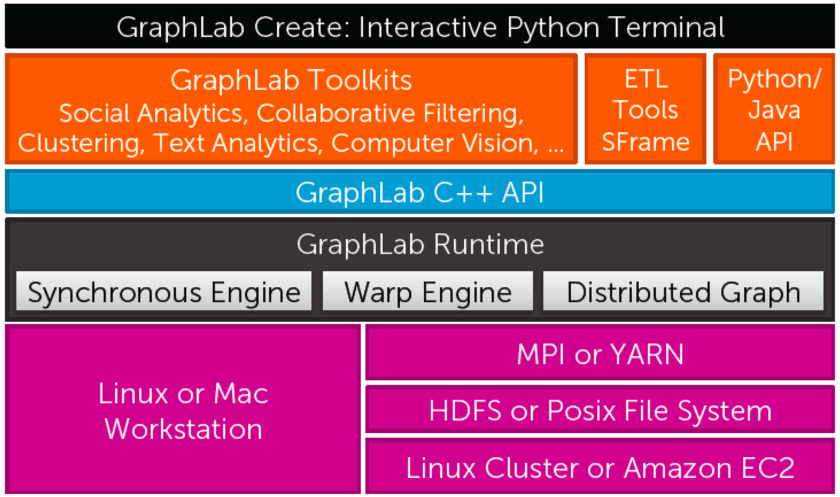
http://graphlab.org

Documentation… Code… Tutorials… (more on the way)

GraphLab for Big Learning (MLDM) Applications



Summary: Different Versions of GraphLab



- GraphLab 1.0 (phased out):
  - Designed to run on closely-coupled, shared-memory multicore machine, performed poorly with PowerLaw Graphs.
- GraphChi: Doing BigData with Small Machine:
  - enables a Single PC to process graphs with billions of edges
- GraphLab (Ver2.x) or so-called the PowerGraph
  - Model targets for seriously-imbalanced node degrees found in practical (Natural) graphs and support parallel processing on Share-Nothing Cluster architecture
  - Taking the split-vertex instead split-edge approach
- GraphCreate (Beta Release since March 2014)
  - allows you to code in your PC using Python but deploy to run over Cloud-based shared-nothing clusters.

Source: Wikipedia (Japanese rock garden)