Case Study 4: Collaborative Filtering

Graph-Parallel Problems

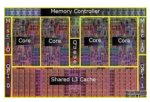
Synchronous v. Asynchronous Computation

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Carlos Guestrin, guest lecturer
May 14th, 2015

ML in the Context of Parallel Architectures

use more processors

GPUs


Multicore


Clusters

Clouds


Supercomputers

- But scalable ML in these systems is hard, especially in terms of:
  1. Programmability
  2. Data distribution
  3. Failures

we'll go through these ideas...
explored with MapReduce

Move Towards Higher-Level Abstraction

- Distributed computing challenges are hard and annoying!
  1. Programmability
  2. Data distribution
  3. Failures
- High-level abstractions try to simplify distributed programming by hiding challenges:
  - Provide different levels of robustness to failures, optimizing data movement and communication, protect against race conditions…
  - Generally, you are still on your own WRT designing parallel algorithms
- Some common parallel abstractions:
  - Lower-level:
    - Pthreads: abstraction for distributed threads on single machine
    - MPI: abstraction for distributed communication in a cluster of computers
  - Higher-level:
    - Map-Reduce (Hadoop: open-source version): mostly data-parallel problems
    - GraphLab: for graph-structured distributed problems

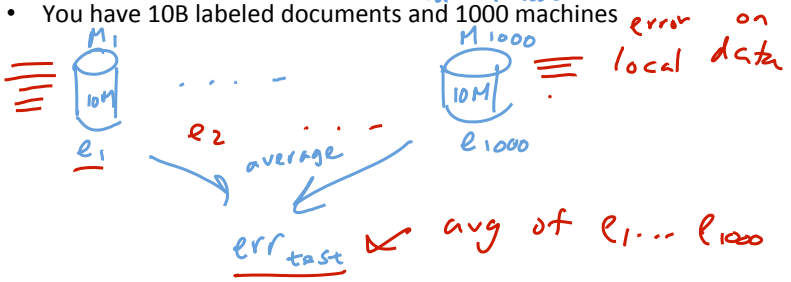
this quarter

Simplest Type of Parallelism: Data Parallel Problems

- You have already learned a classifier
  - What's the test error?

$$err_{test} = \frac{1}{N_{test}} \sum \left| y^{(i)} - sign(w^* \cdot x^{(i)}) \right|$$

- You have 10B labeled documents and 1000 machines



- Problems that can be broken into independent subproblems are called data-parallel (or embarrassingly parallel)
- Map-Reduce is a great tool for this…
  - Focus of today's lecture
  - but first a simple example

Data Parallelism (MapReduce)

*Solve a huge number of **independent** subproblems, e.g., extract features in images*

Map-Reduce Abstraction

- Map: Transform a data element
  - Data-parallel over elements, e.g., documents
  - Generate (key,value) pairs
    - "value" can be any data type

('uw', 17)

In our example: ('uw', 1) ('Mary', 1) ('uw', 1)

Example: word count

```
map(document)
  for word in doc
    emit (word, 1)
```

key, value

- Reduce: Take all values associated with a key and aggregate
  - Aggregate values for each key
  - Must be commutative-associate operation
  - Data-parallel over keys
  - Generate (key,value) pairs

reduce('uw', [1, 17, 10, 12])

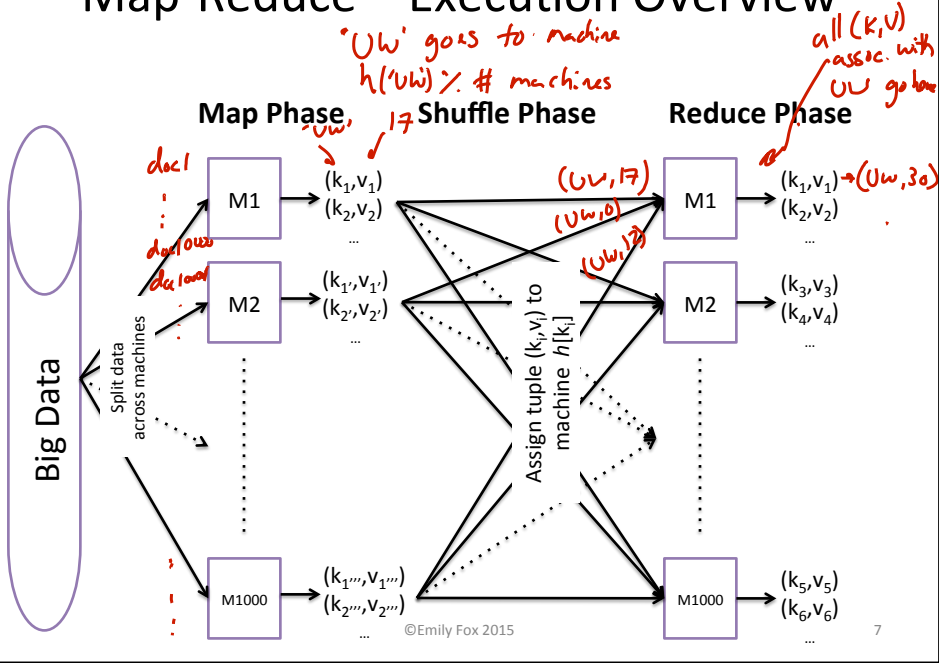
emit('uw', 30)

key, total count

```
Reduce(word, count:list(int))
  c=0          key, list of counts
  for i in count
    c+= count[i]
  emit(word, c)
                 total
```

- Map-Reduce has long history in functional programming
  - But popularized by Google, and subsequently by open-source Hadoop implementation from Yahoo!

Map-Reduce – Execution Overview



Issues with Map-Reduce Abstraction

- Often all data gets moved around cluster
  - Very bad for iterative settings
- Definition of Map & Reduce functions can be unintuitive in many apps
  - Graphs are challenging
- Computation is synchronous

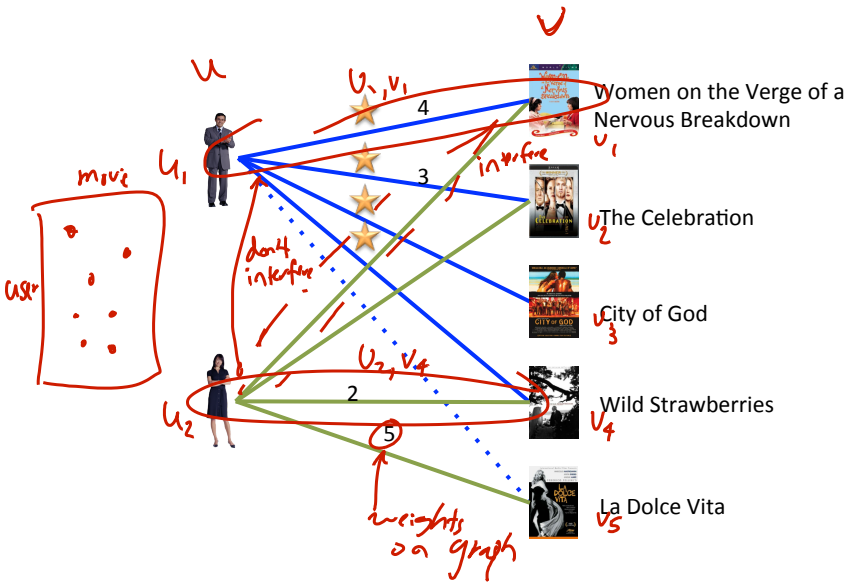
SGD for Matrix Factorization in Map-Reduce?

$$\begin{bmatrix} L_u^{(t+1)} \\ R_v^{(t+1)} \end{bmatrix} \leftarrow \begin{bmatrix} (1-\eta_t\lambda_u)L_u^{(t)} - \eta_t\epsilon_t R_v^{(t)} \\ (1-\eta_t\lambda_v)R_v^{(t)} - \eta_t\epsilon_t L_u^{(t)} \end{bmatrix}$$

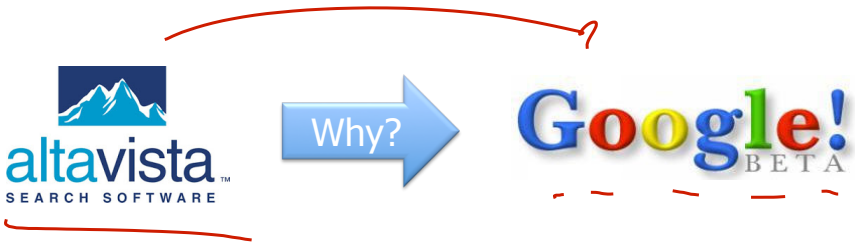
$$\epsilon_t = L_u^{(t)} \cdot R_v^{(t)} - r_{uv}$$

- Map and Reduce functions???
- Map-Reduce:
  - Data-parallel over all mappers
  - Data-parallel over reducers with same key
- Here, one update at a time!

Matrix Factorization as a Graph



Flashback to 1998



First Google advantage:
a **Graph Algorithm** & a **System to Support** it!

PageRank at Scale

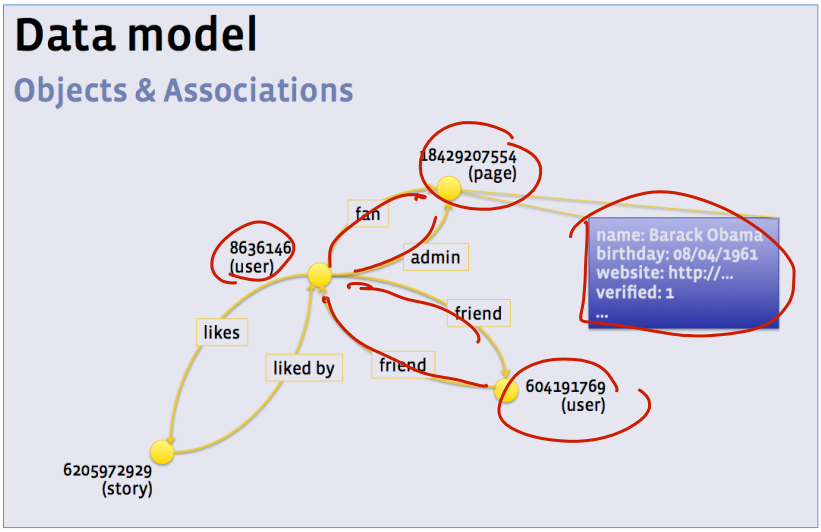
Social Media **Science** **Advertising** **Web**

- **Graphs** encode the **relationships** between:

People Products Ideas
Facts Interests

- **Big: 100 billions** of **vertices** and **edges** and rich metadata
  - Facebook (10/2012): 1B users, 144B friendships
  - Twitter (2011): 15B follower edges

Facebook Graph



Slide from Facebook Engineering presentation

Label a Face and Propagate

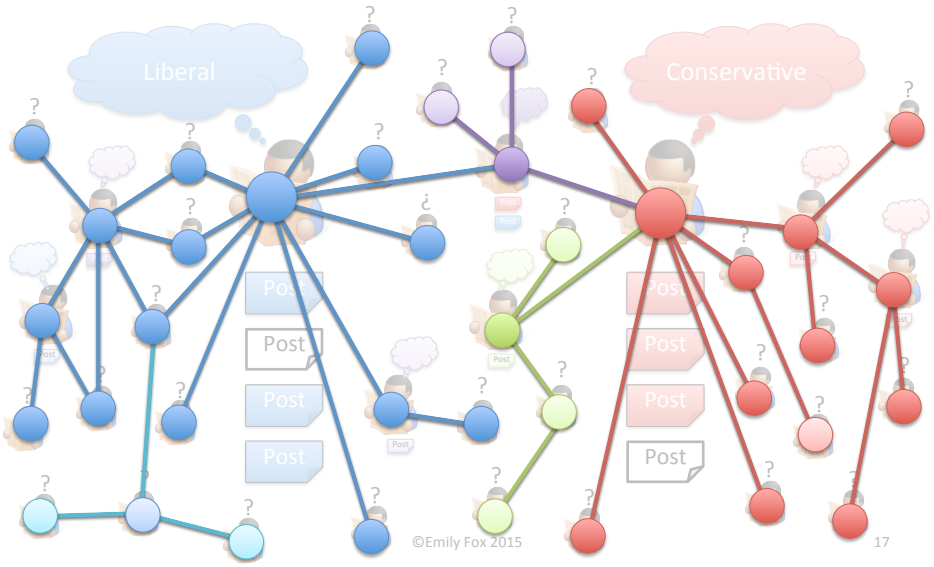
Pairwise similarity not enough…

15

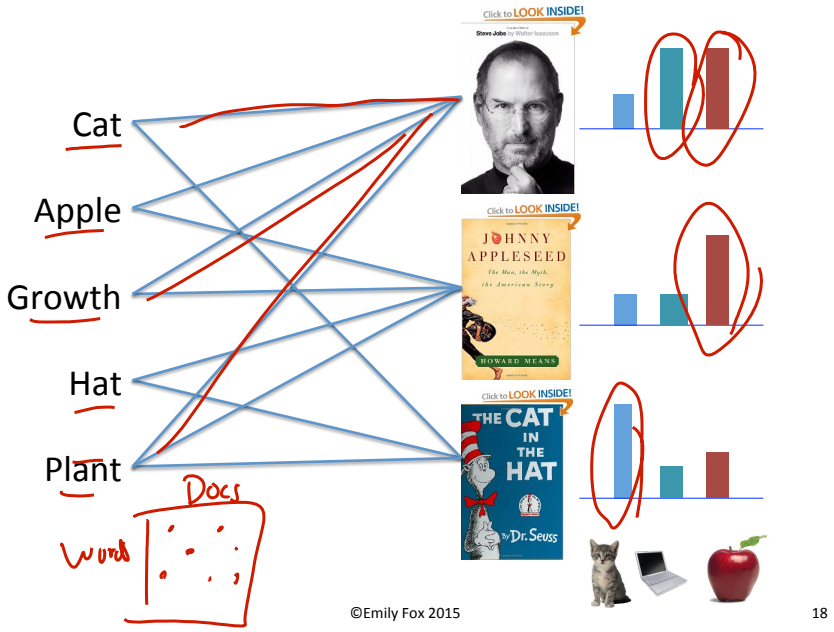
Propagate Similarities & Co-occurrences for Accurate Predictions

16

Example: *Estimate Political Bias*



Topic Modeling (e.g., LDA)



ML Tasks Beyond Data-Parallelism

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

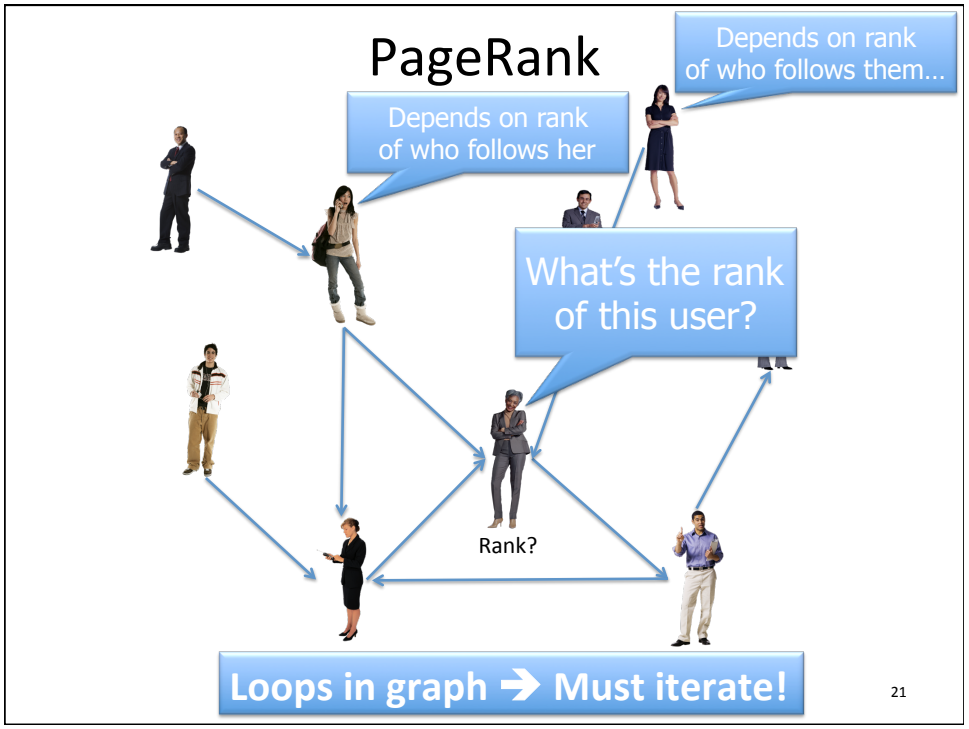
Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

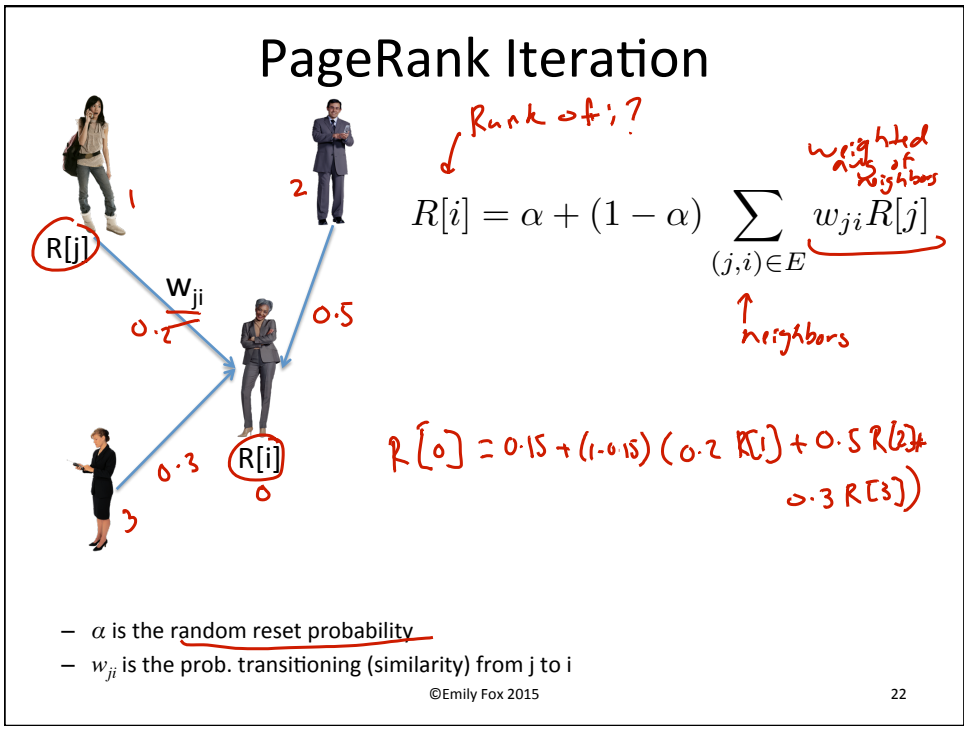
Semi-Supervised Learning
Label Propagation
CoEM

Collaborative Filtering
Tensor Factorization

Graph Analysis
PageRank
Triangle Counting

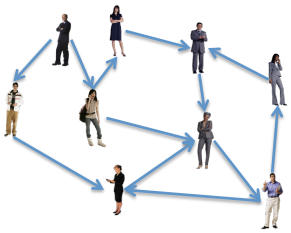
Example of a Graph-Parallel Algorithm


PageRank
Depends on rank of who follows them…
Depends on rank of who follows her
What's the rank of this user?
Rank?
Loops in graph → Must iterate!
21

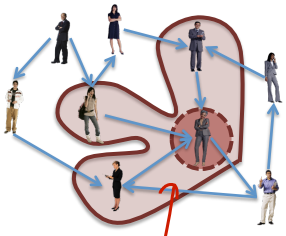

PageRank Iteration
R[j]
w_{ji}
0.2
0.5
0.3
R[i]
1
2
3
0
Rank of i?
weighted avg of neighbors
$R[i] = \alpha + (1-\alpha) \sum_{(j,i)\in E} w_{ji} R[j]$
neighbors
R[0] = 0.15 + (1-0.15)(0.2 R[1] + 0.5 R[2] + 0.3 R[3])
– α is the random reset probability
– w_{ji} is the prob. transitioning (similarity) from j to i
©Emily Fox 2015
22

Properties of Graph Parallel Algorithms

Dependency Graph



Local Updates



Iterative Computation

R[o] ← weighted avg of nbrs

iterate until convergence

Addressing Graph-Parallel ML

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Graph-Parallel Abstraction

Graphical Models
Gibbs Sampling
Belief Propagation
Variational Opt.

Semi-Supervised Learning
Label Propagation
CoEM

Collaborative Filtering
Tensor Factorization

Data-Mining
PageRank
Triangle Counting

Key Question.

Graph Computation:

Synchronous

v.

Asynchronous

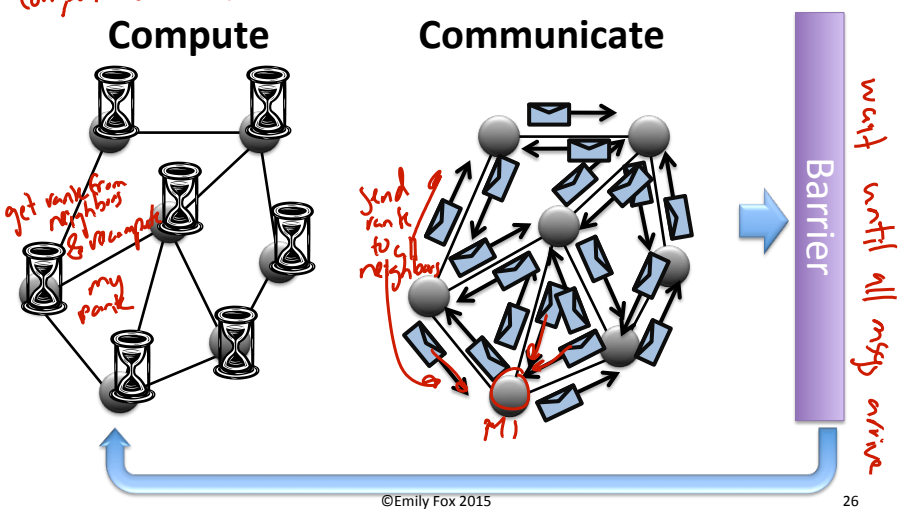
Bulk Synchronous Parallel Model: Pregel (Giraph)

[Valiant '90]

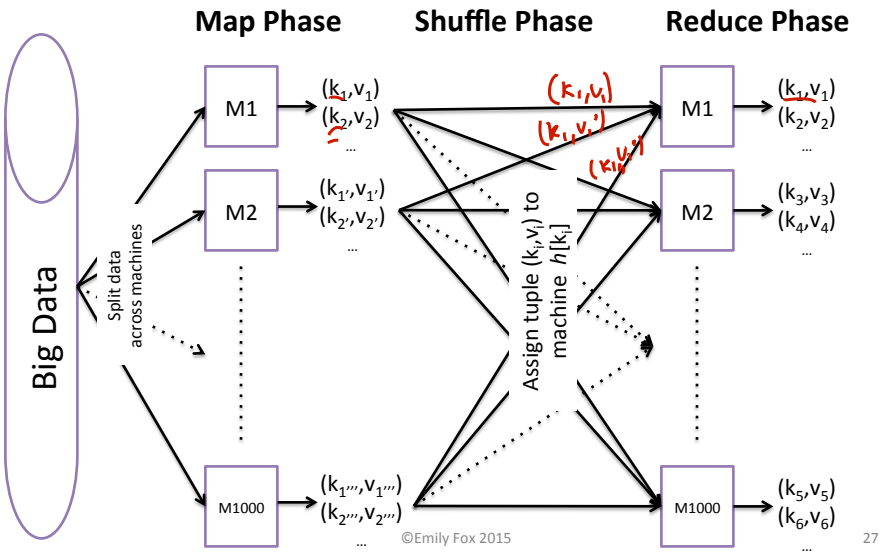
in parallel locally compute something

send messages to all neighbors

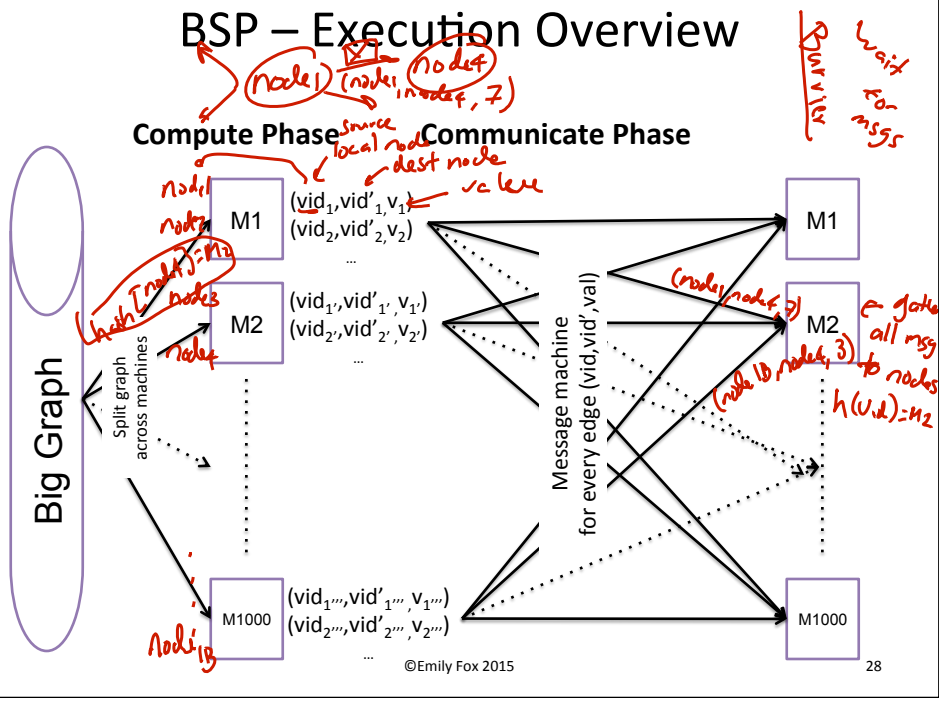
Compute **Communicate**



Map-Reduce – Execution Overview



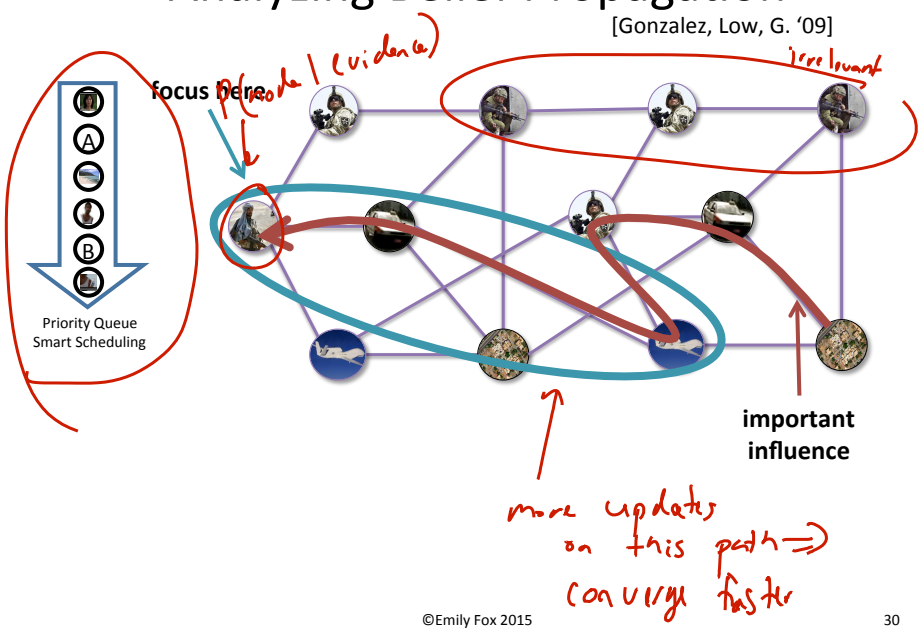
BSP – Execution Overview



Bulk synchronous parallel model ***provably inefficient*** *for some ML tasks*

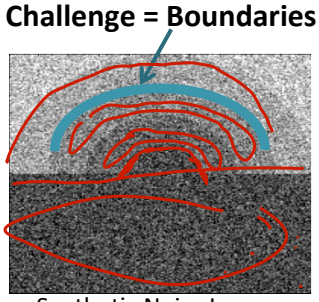
Analyzing Belief Propagation

[Gonzalez, Low, G. '09]

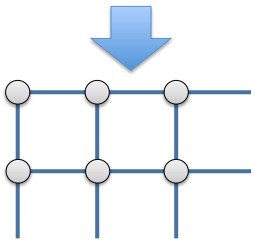


Asynchronous Belief Propagation

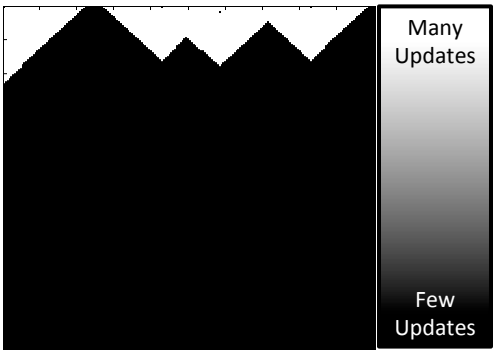
Challenge = Boundaries



Synthetic Noisy Image



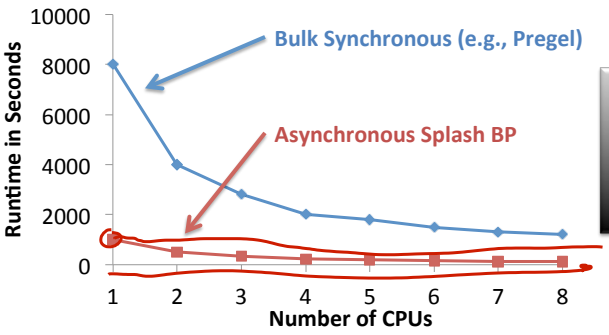
Graphical Model



Cumulative Vertex Updates

Algorithm identifies and focuses on hidden sequential structure

BSP ML Problem: Synchronous Algorithms can be **Inefficient**



Theorem:
Bulk Synchronous BP O(#vertices) slower than Asynchronous BP

Synchronous v. Asynchronous

- Bulk synchronous processing:
  - Computation in phases
    - All vertices participate in a phase
      - Though OK to say no-op
    - All messages are sent
  - Simpler to build, like Map-Reduce
    - No worries about race conditions, barrier guarantees data consistency
    - Simpler to make fault-tolerant, save data on barrier
  - Slower convergence for many ML problems
  - In matrix-land, called Jacobi Iteration
  - Implemented by Google Pregel 2010

- Asynchronous processing:
  - Vertices see latest information from neighbors
    - Most closely related to sequential execution
  - Harder to build:
    - Race conditions can happen all the time
      - Must protect against this issue
    - More complex fault tolerance
    - When are you done?
    - Must implement scheduler over vertices
  - Faster convergence for many ML problems
  - In matrix-land, called Gauss-Seidel Iteration
  - Implemented by GraphLab 2010, 2012

Case Study 4: Collaborative Filtering

GraphLab

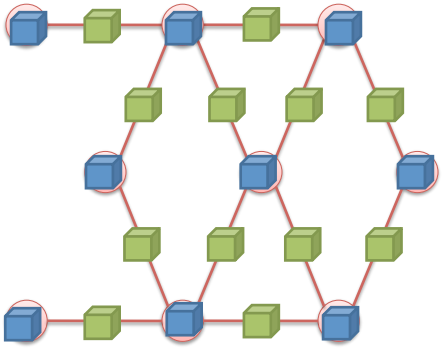
Machine Learning for Big Data
CSE547/STAT548, University of Washington

Carlos Guestrin, guest lecturer

May 14th, 2015

Data Graph

Data associated with vertices and edges



Graph:
- Social Network

Vertex Data:
- User profile text
- Current interests estimates

Edge Data:
- Similarity weights

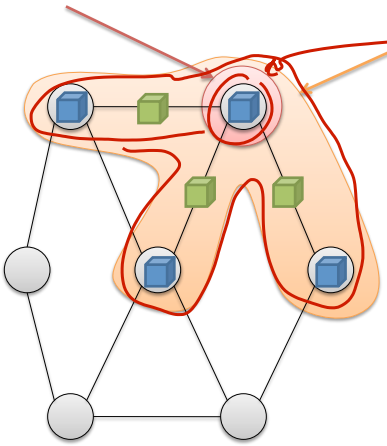
How do we *program* **graph** computation?

"Think like a Vertex."

-Malewicz et al. [SIGMOD'10]

Update Functions

User-defined program: applied to **vertex** transforms data in **scope** of vertex



vertex index
neighbors

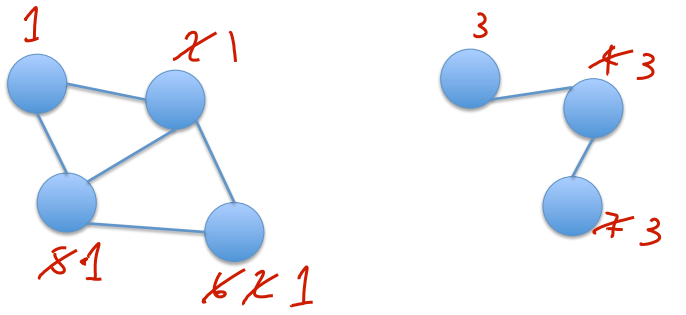
pagerank(i, scope){

$R[i] = \alpha + (1-\alpha) \sum_{j \in Scope.neighbors} Scope.edge(i,j).weight \cdot Scope.neighbor(j).Rank$

contained in scope of compute

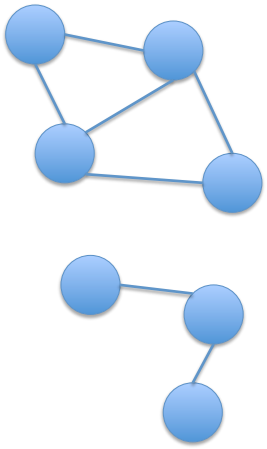
}

Update Function Example: Connected Components



1. initialize: vertex component = vertex_id
2. iterate
   pick a node i:
   component(i) = min (self, neighbors)
   if change, add neighbors to queue

Update Function Example: Connected Components



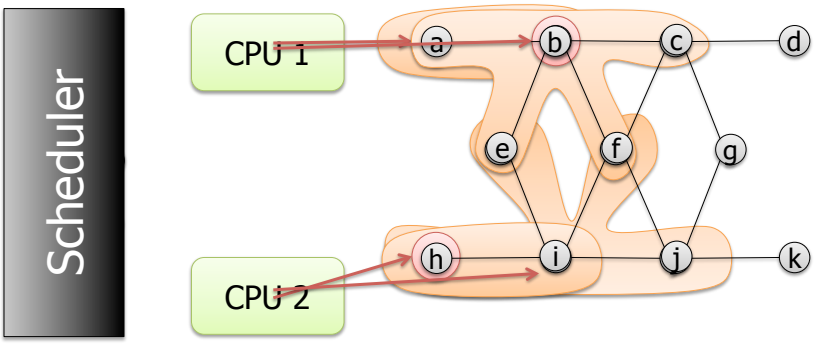
```
init: component[i] = i

update(i, scope):
    component[i] = min(
        component[i],
        min_{j ∈ neighbors} (component[j]))

; if component(i) changes !
{   schedule neighbors of i !;
```


The Scheduler

The **scheduler** determines order vertices are updated

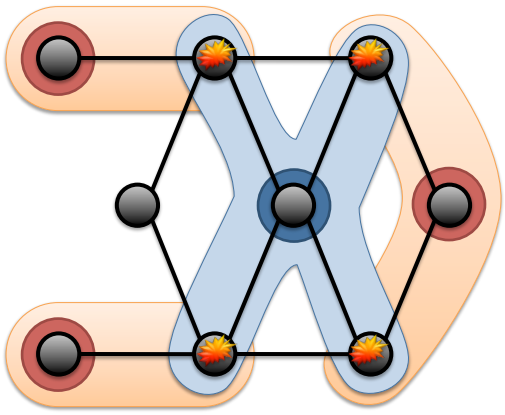


Example Schedulers

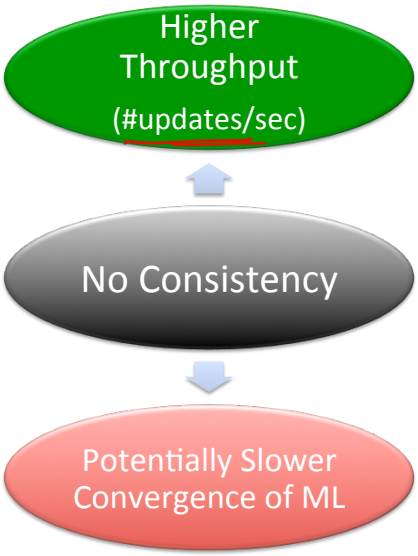
- Round-robin
- Selective scheduling (skipping):
  - round robin but jump over un-scheduled vertice
- FIFO
- Prioritize scheduling
  - Hard to implement in a distributed fashion
    - Approximations used (each machine has its own priority queue)

Ensuring Race-Free Code

How much can computation **overlap**?

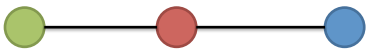


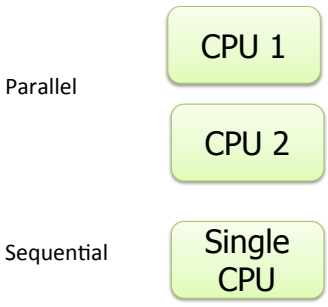
Need for Consistency?



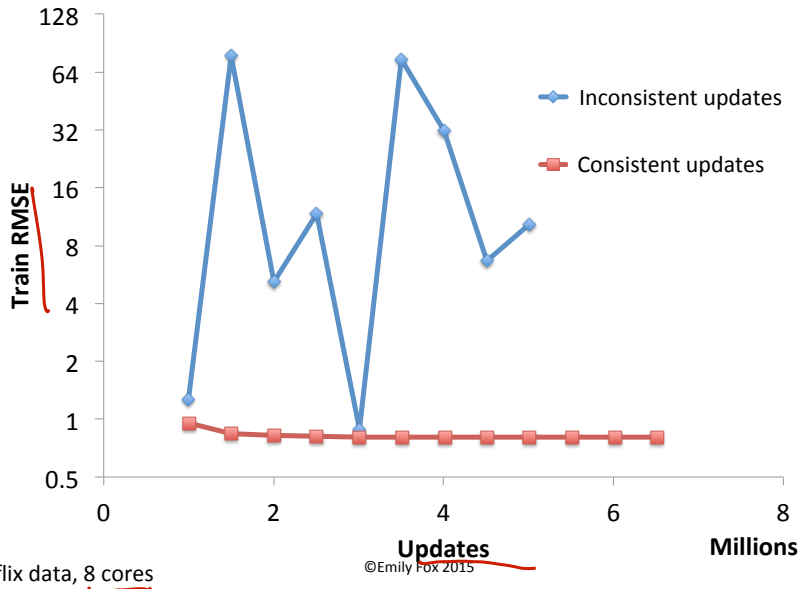
GraphLab Ensures **Sequential Consistency**

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





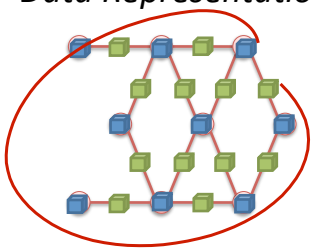
Consistency in Collaborative Filtering



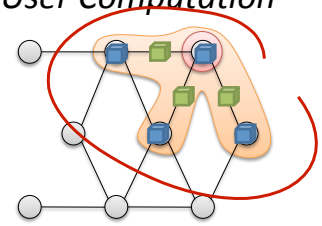
Netflix data, 8 cores

The GraphLab Framework

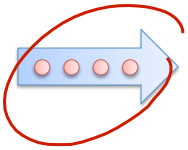
Graph Based
Data Representation



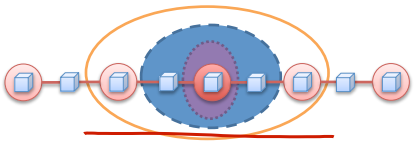
Update Functions
User Computation



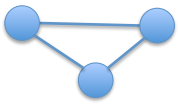
Scheduler



Consistency Model

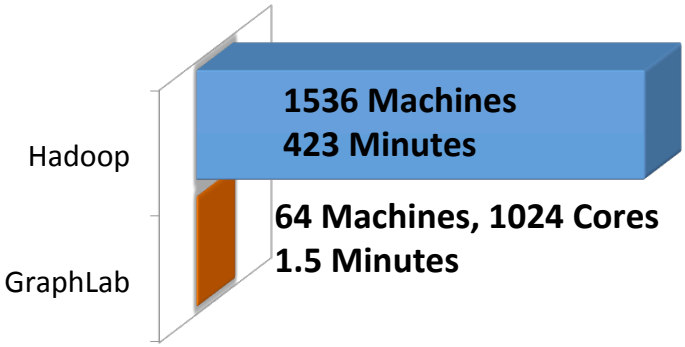


Triangle Counting in Twitter Graph



40M Users
1.2B Edges

Total:
34.8 Billion Triangles

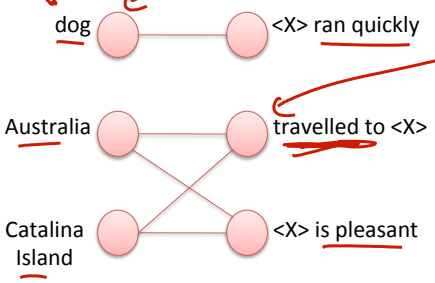


Hadoop results from [Suri & Vassilvitskii '11]

CoEM (Jones et al., 2005)

Named Entity Recognition Task

Is "Dog" an animal?
Is "Catalina" a place?

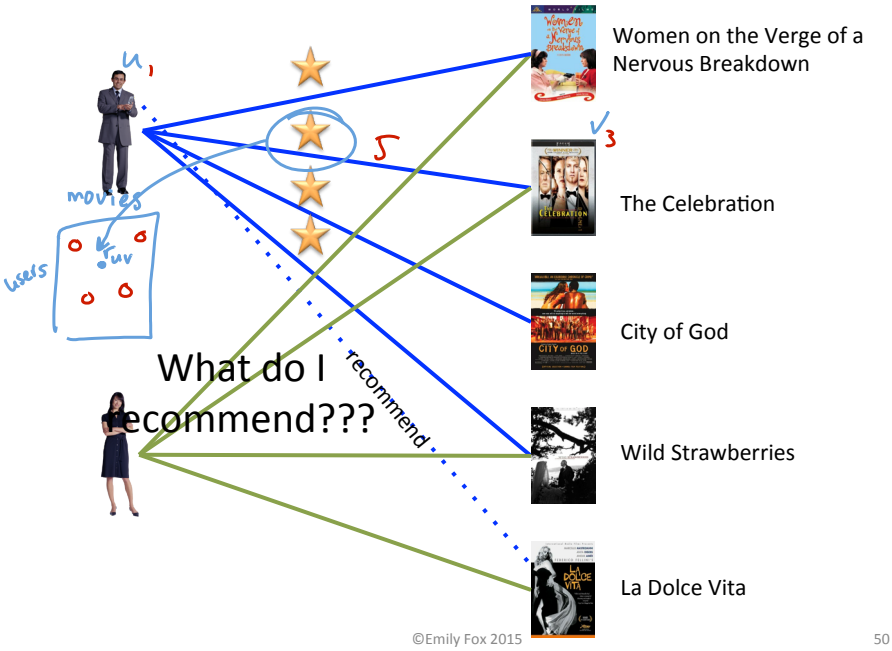


semi supervised learning
label a few nodes
dist over type
{ animal, place, ... }
dist over type of X
{ animal, place, ... }
for this context
initialize: randomly, smartly
→ iterative:
EM-like alg

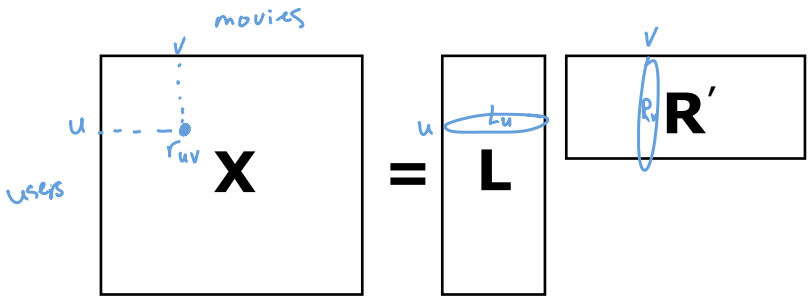
Never Ending Learner Project (CoEM)

Vertices: 2 Million
Edges: 200 Million

Hadoop	95 Cores	7.5 hrs
Distributed GraphLab 2012	**32 EC2 machines**	**80 secs**



Interpreting Low-Rank Matrix Completion (aka Matrix Factorization)



$r_{uv} \approx L_u \cdot R_v$

factor

L_u = [| | |]

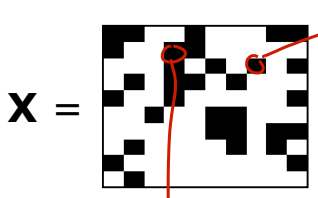
R_v = [| | |]

movie i topic i "action"

how much user u likes topic i

how much movie v is about topic i

Matrix Completion as a Graph



X_{ij} known for black cells

X_{ij} unknown for white cells

Rows index users

Columns index movies

Coordinate Descent for Matrix Factorization: Alternating Least-Squares

$$\min_{L,R} \sum_{(u,v):r_{uv}\neq ?} (L_u \cdot R_v - r_{uv})^2 + \lambda_u \|L\| + \lambda_v \|R\|$$

- Fix movie factors, optimize for user factors
  - Independent least-squares over users

$$\min_{L_u} \sum_{v \in V_u} (L_u \cdot R_v - r_{uv})^2 + \lambda_u \|L\|$$

V_u: neighbors of u

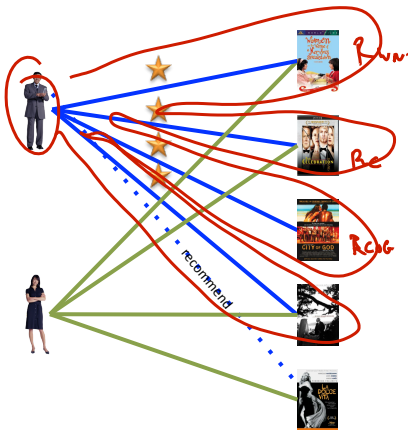
- Fix user factors, optimize for movie factors
  - Independent least-squares over movies

$$\min_{R_v} \sum_{u \in U_v} (L_u \cdot R_v - r_{uv})^2 + \lambda_v \|R\|$$

- System may be underdetermined: use regularization
- Converges to local optima

Alternating Least Squares Update Function

$$\min_{L_u} \sum_{v \in V_u} (L_u \cdot R_v - r_{uv})^2 \qquad \min_{R_v} \sum_{u \in U_v} (L_u \cdot R_v - r_{uv})^2$$

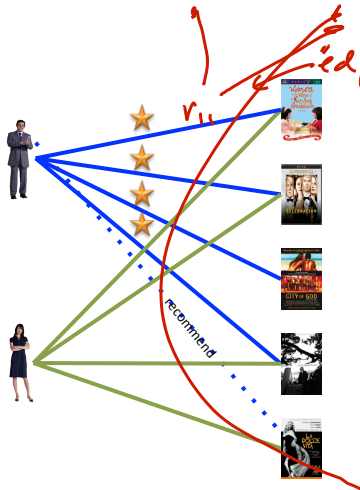



```
update (u, scope):
  // goal: estimate L_u
  // from scope gather factors of neighbors
  X = [ R_v ]
  // read all ratings from edges
  Y = [ r_uv ]
  // solve a local least squares problem
  e.g. L2 reg:
  L_u = (X^T X + λ_u I)^{-1} X^T Y
```


SGD for Matrix Factorization in GraphLab

$$\epsilon_t = L_u^{(t)} \cdot R_v^{(t)} - r_{uv}$$

$$\begin{bmatrix} L_u^{(t+1)} \\ R_v^{(t+1)} \end{bmatrix} \leftarrow \begin{bmatrix} (1-\eta_t\lambda_u)L_u^{(t)} - \eta_t\epsilon_t R_v^{(t)} \\ (1-\eta_t\lambda_v)R_v^{(t)} - \eta_t\epsilon_t L_u^{(t)} \end{bmatrix}$$



edge update

GraphLab operates on vertices

update (u, scope):
// perform SGD for
every edge connected
to u
for each (u,v) edge in scope
do

Out-of-core computation

- Often data doesn't fit in memory
  - Must use disk/SSDs
- Although random accesses to disk are very slow, high performance is possible with optimized memory layout. Implemented in:
  - GraphChi
  - GraphLab Create

'Mac mini' Same running time Hadoop Cluster

- Example performance of GraphLab Create:
  - Common Crawl Graph (3.5 billion Nodes and 128 billion Edges)
  - PageRank iteration in 9 mins on single, commodity machine

What you need to know…

- Data-parallel versus graph-parallel computation
- Bulk synchronous processing versus asynchronous processing
- GraphLab system for graph-parallel computation
  - Data representation
  - Update functions
  - Scheduling
  - Consistency model
- ALS, SGD for matrix factorization in GraphLab

Reading

- Papers under "Case Study IV: **Parallel Learning with GraphLab**"
- Optional:
  - Parallel Splash BP
    http://www.ml.cmu.edu/research/dap-papers/dap-gonzalez.pdf

