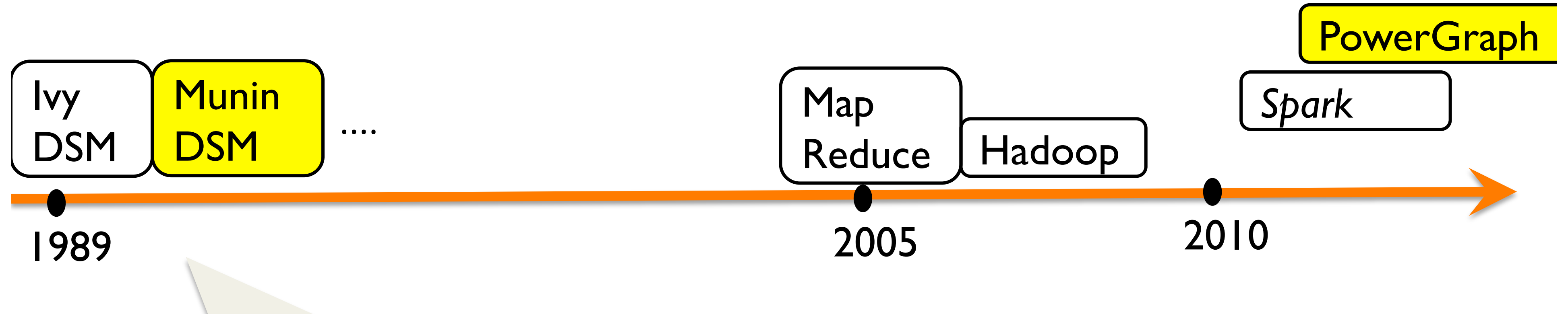
# DSM and Graph Computation Frameworks

Jinyang Li

(GraphLab slides from Gonzalez' OSDI talk)

# Distributed Computation




Distributed computation in the 90s focus on the distributed shared memory model

# Distributed shared memory

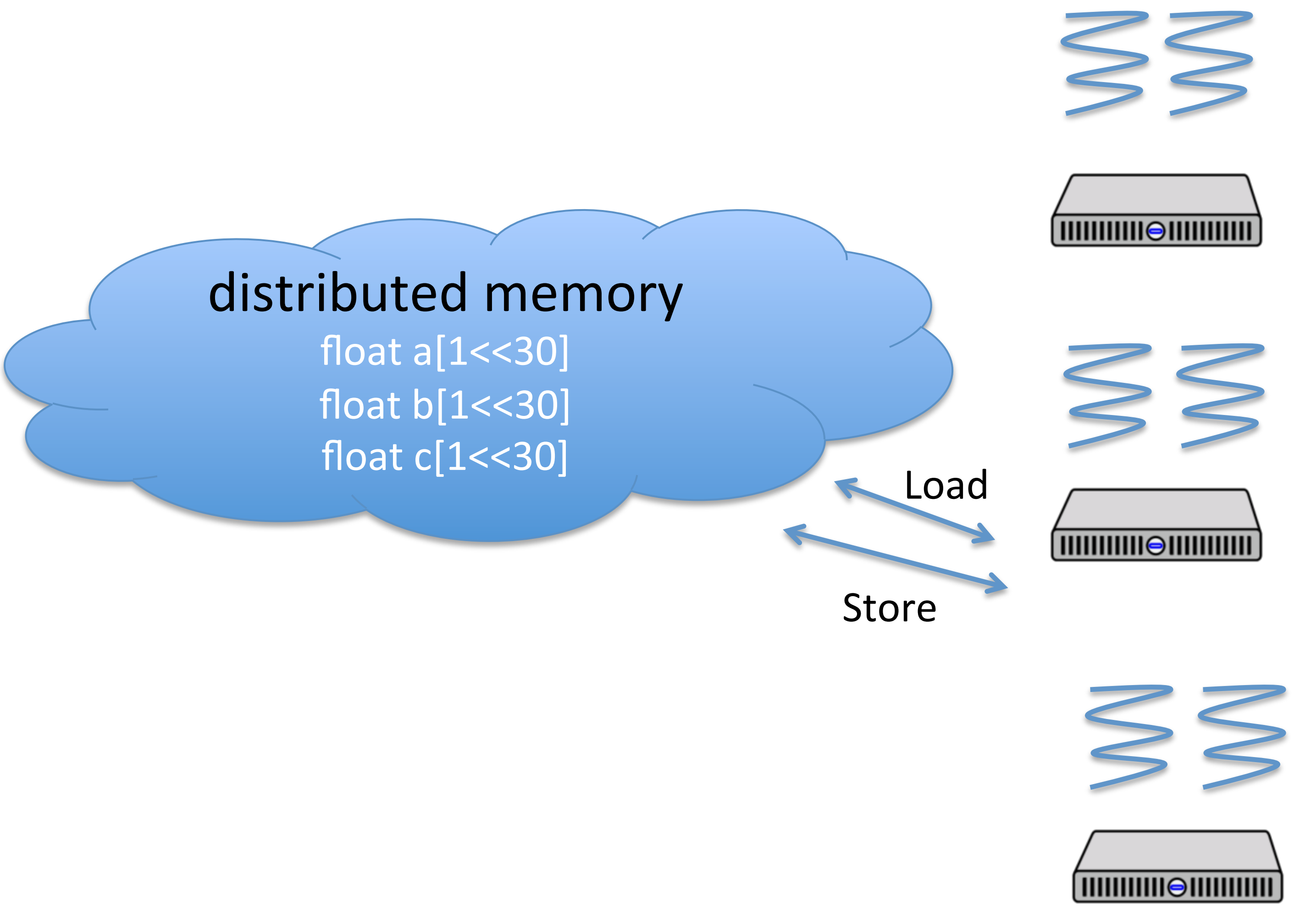
Goal:

- Write any distributed computation the way you'd write a single-machine multi-threaded computation

# Example: adding two arrays

```
float a[1<<30];
float b[1<<30];
float c[1<<30];

void addChunk(thread_id idx)
{
        long long start = (1 << 20) * idx;
        for (int i = start; i < start+(1<<20); i++ ) {
                c[i] =  a[i] + b[i];
        }
}
void
main() {
   //launch 1024 threads, each invoking function addChunk
    launchThreads(1024, addChunk);
}
```

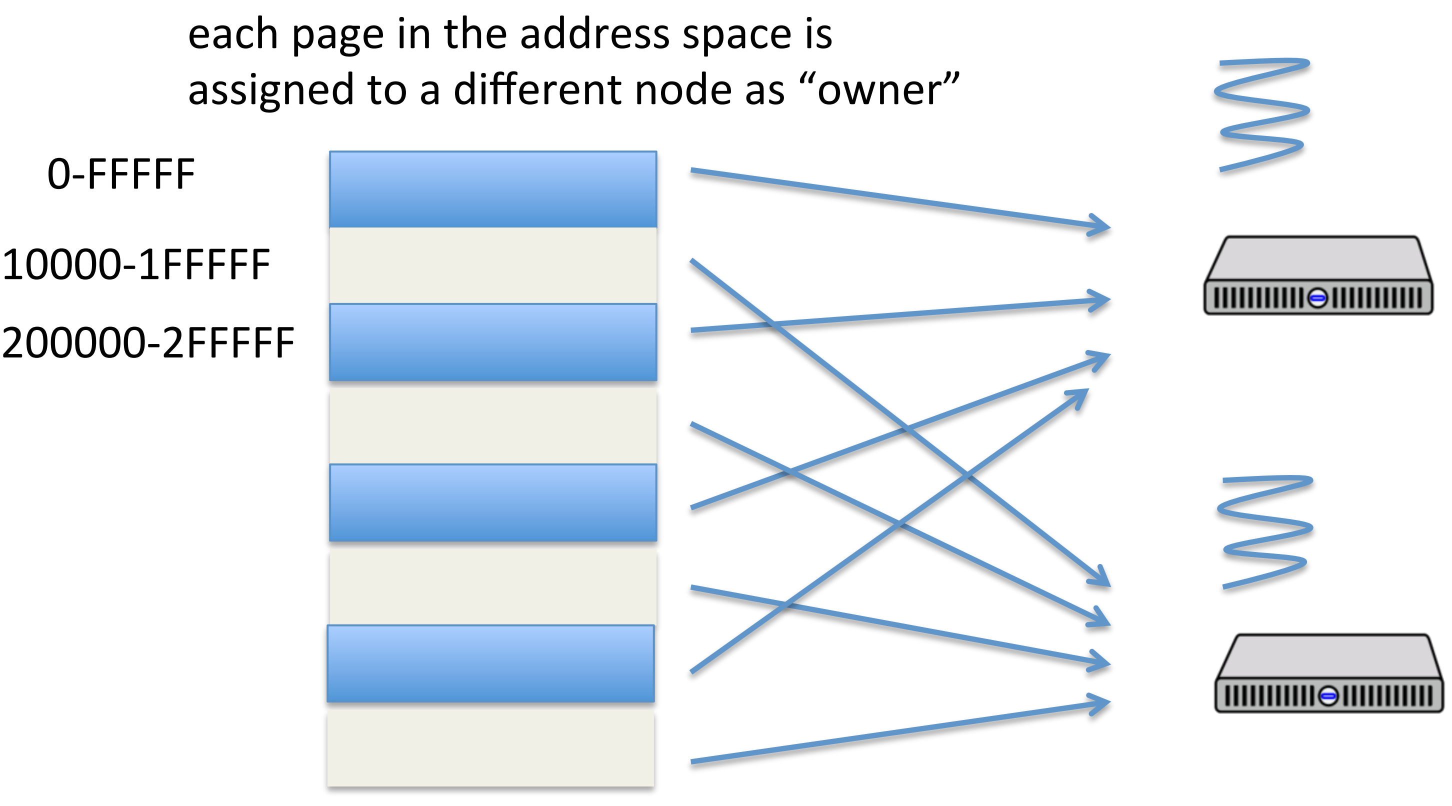
# Distributed shared memory enabled distributed multi-threading



# Advantages of the DSM model

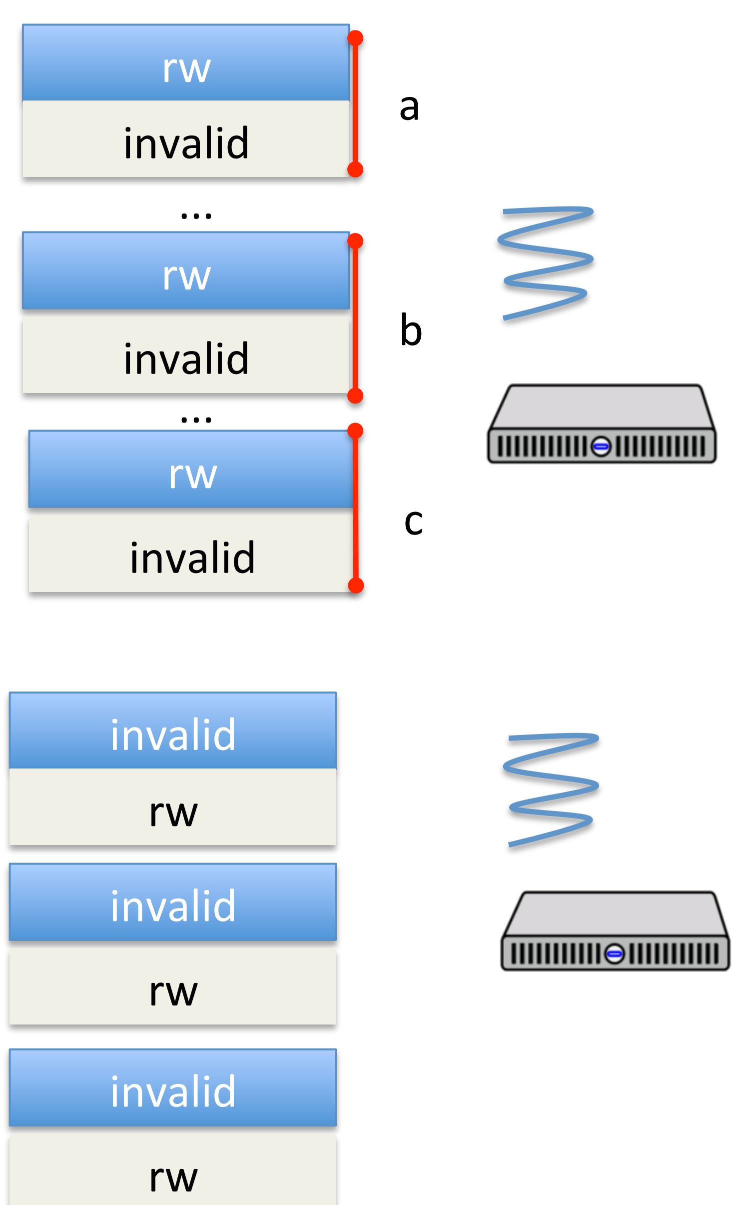
- Familiar programming model
  - shared variables, locks.
- General purpose
  - Any type of computation can be supported
    - unlike MapReduce, Spark
  - Language agnostic
- Allow re-use of existing apps and library written for single machine

# Supporting DSM: conventional approach

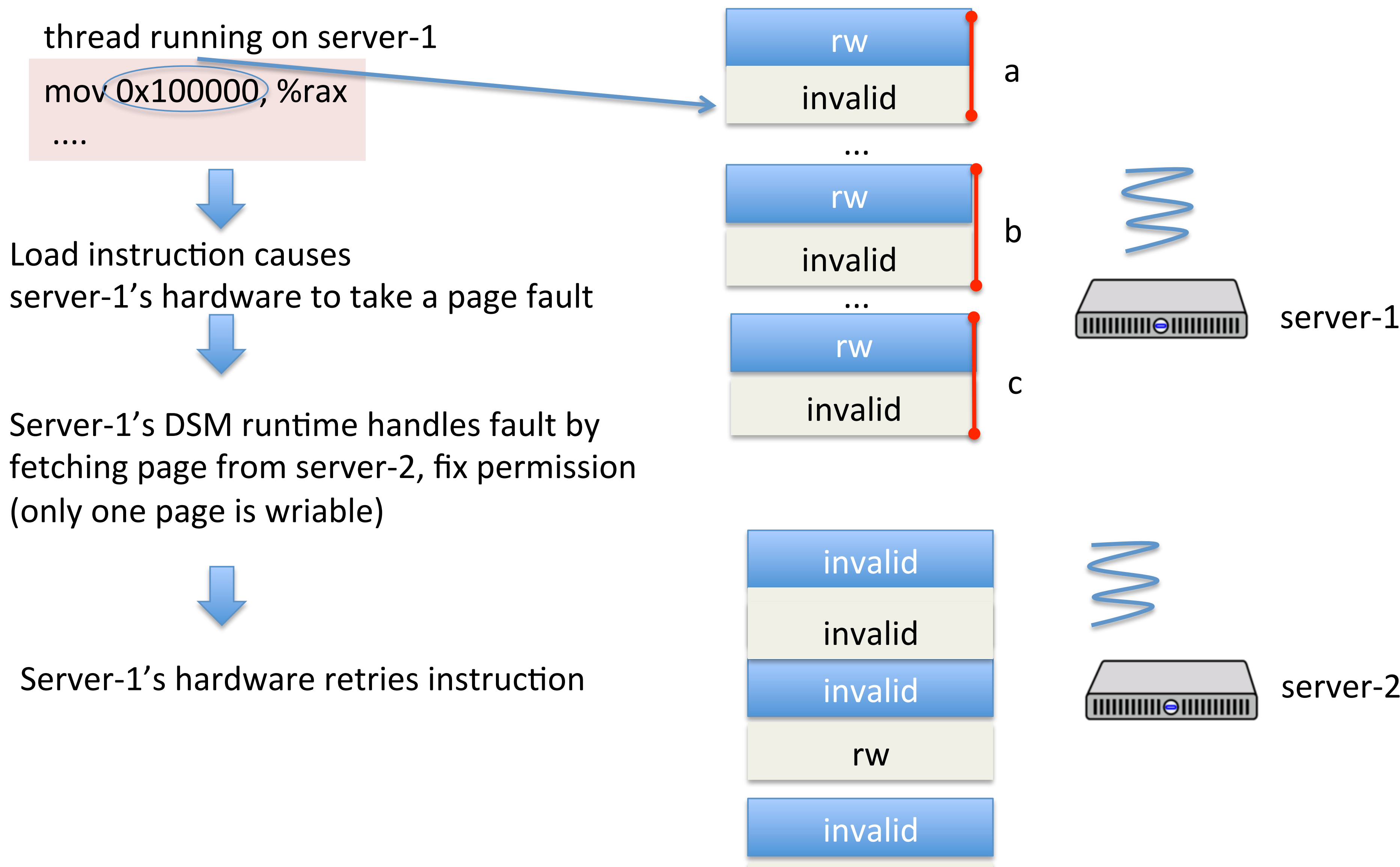


# Supporting DSM: conventional approach

```
for (i = start; i<start+(1<<20); i++) {
  c[i] = a[i]+b[i];
}
```



# Supporting DSM: conventional approach



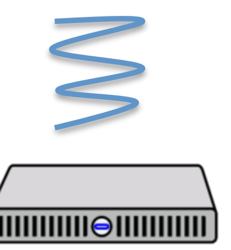
# DSM challenges

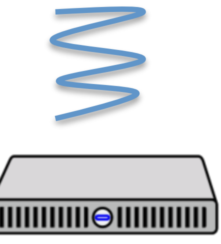
- Memory consistency model
  - What should a read observe?
- Performance
  - Is it fast? Is it scalable?

# Memory consistency affects program correctness

```
x = 1
if y == 0 {
    print “yes”
}
```

```
y = 1
if x == 0 {
    print “yes”
}
```





- Will both threads print “yes”?
  - under sequential consistency?
  - under Go’s memory model?

# Munin’s memory model

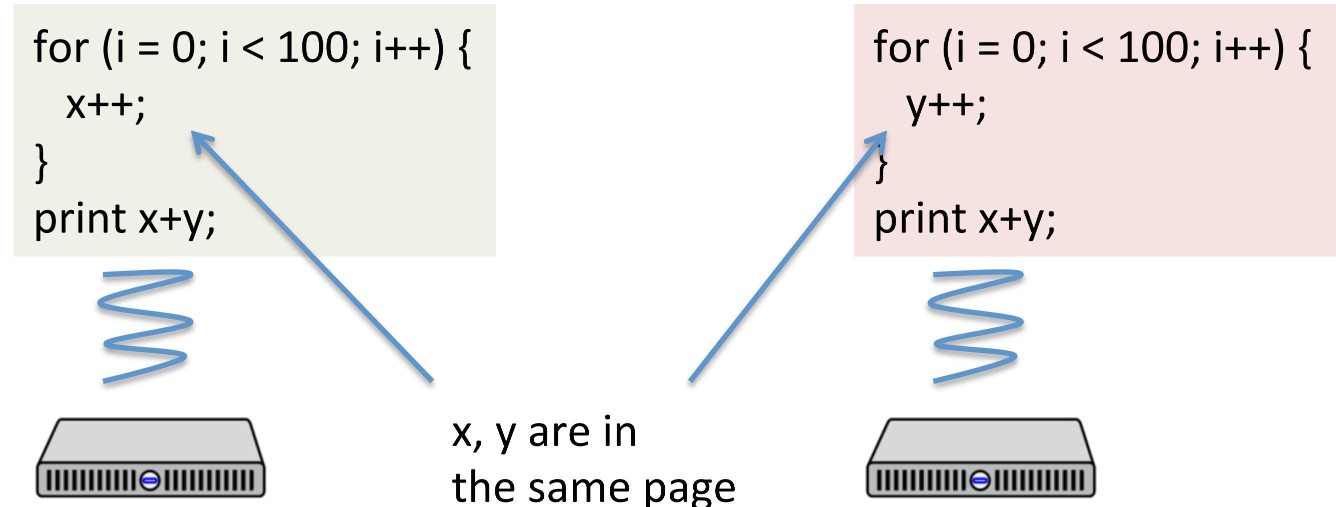
- Release consistency (RC)
  - Weaker than sequential consistency
- Key idea:
  - Access of shared data are commonly protected by synchronization primitives.
  - Sync primitives: Acquire (aka Lock), Release (aka Unlock)
- RC is a partial order:
  - All sync primitives are totally ordered
  - With a thread, the ordering of ordinary memory access w.r.t. synchronization primitive must be preserved

# Why Release Consistency

- Release consistency is more efficient to implement
- A server's writes need not be visible to others until the next synchronization primitive

# How RC addresses false sharing

- A main DSM challenge: false sharing




False sharing leads to ping-ponging and write-amplification:

- To write one-byte to x, S1 transfers whole page from S2, invalidates the page at S2.
- To write one-byte to y, S2 transfers the page back from S1, invalidates the page at S1, and so on.

# How RC addresses false sharing

- A main DSM challenge: false sharing

False sharing leads to ping-ponging and write-amplification:

- To write one-byte to x, S1 transfers whole page from S2, invalidates the page at S2.
- To write one-byte to y, S2 transfers the page back from S1, invalidates the page at S1, and so on.

# How RC addresses false sharing

- A main DSM challenge: false sharing

False sharing leads to ping-ponging and write-amplification:

- To write one-byte to x, S1 transfers whole page from S2, invalidates the page at S2.
- To write one-byte to y, S2 transfers the page back from S1, invalidates the page at S1, and so on.

# Idea: Write diffs + Release Consistency

- To write, transfer a copy, but do not invalidate other writable-copies of the page

- Send out and merge diffs on release

# Release Consistency

server-1

```
Acquire(Lx)
for (i = 0; i < 100; i++) {
   x++;
}
Release(Lx)
print x+y;      <-- Acquire(Ly)
                <-- Release(Ly)
```

server-2

```
Acquire(Ly)
for (i = 0; i < 100; i++) {
   y++;
}
Release(Ly)     <-- Acquire(Lx)
print x+y;
                <-- Release(Lx)
```

- What's the possible outcomes under Munin?
  - <100, 100> <200, 100> <100, 200> <200, 200>
- What's possible after adding new acquires/ release?
- How many network transfers?

# DSM’s failure story

- DSMs rely on checkpointing to recover from failure.
- Periodically checkpoint all servers’ state.
- On recovery, load from last checkpoint and resume

# Why no DSM now?

- Masking the difference between distributed and single-machine computation is too hard
- Difference in memory fetch latency is huge
  - 100 ns vs. 10us~1 ms
- Programs that make sense in single-machine setting are too slow on DSM

# An example computation that's difficult for DSM: PageRank

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

Rank of node *i*

Weighted sum of neighbors' ranks

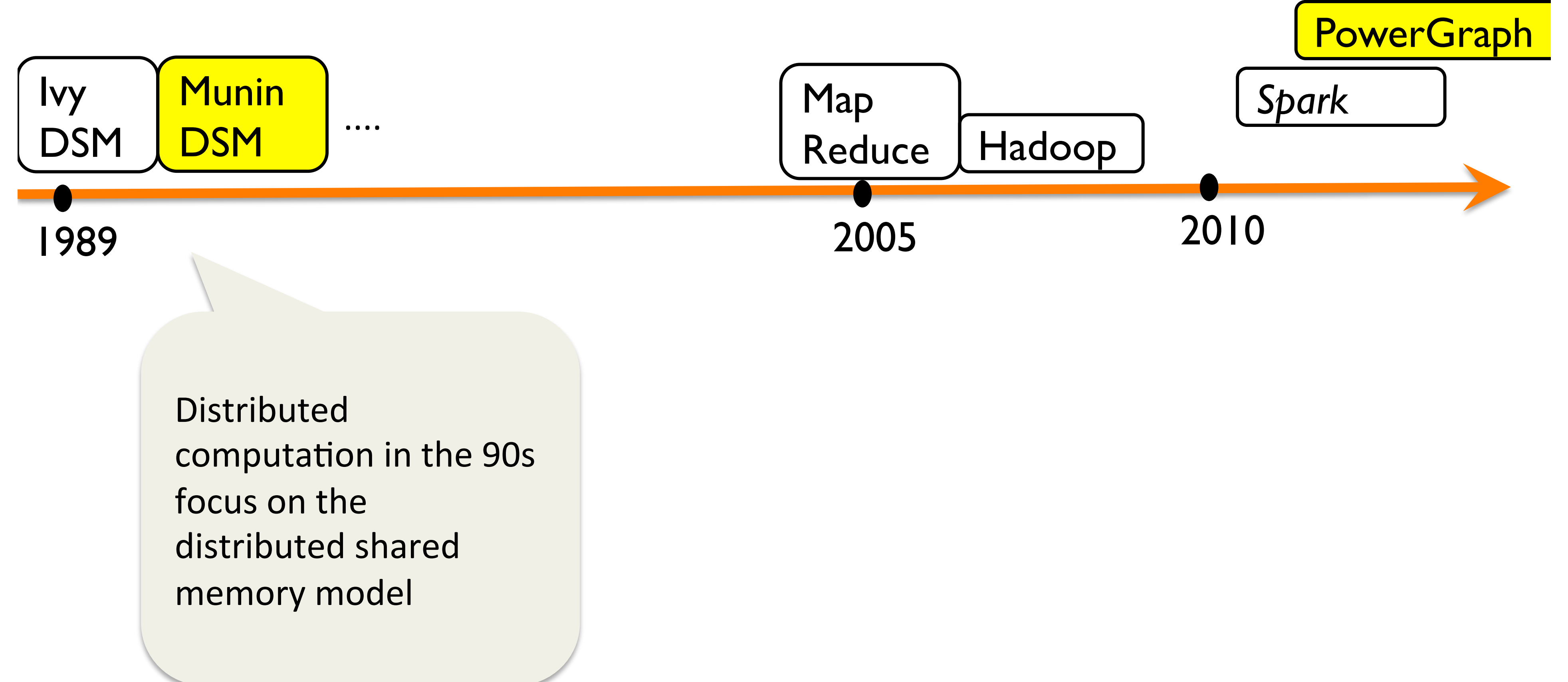
- Iterate until convergence

# Difficulty of DSM

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

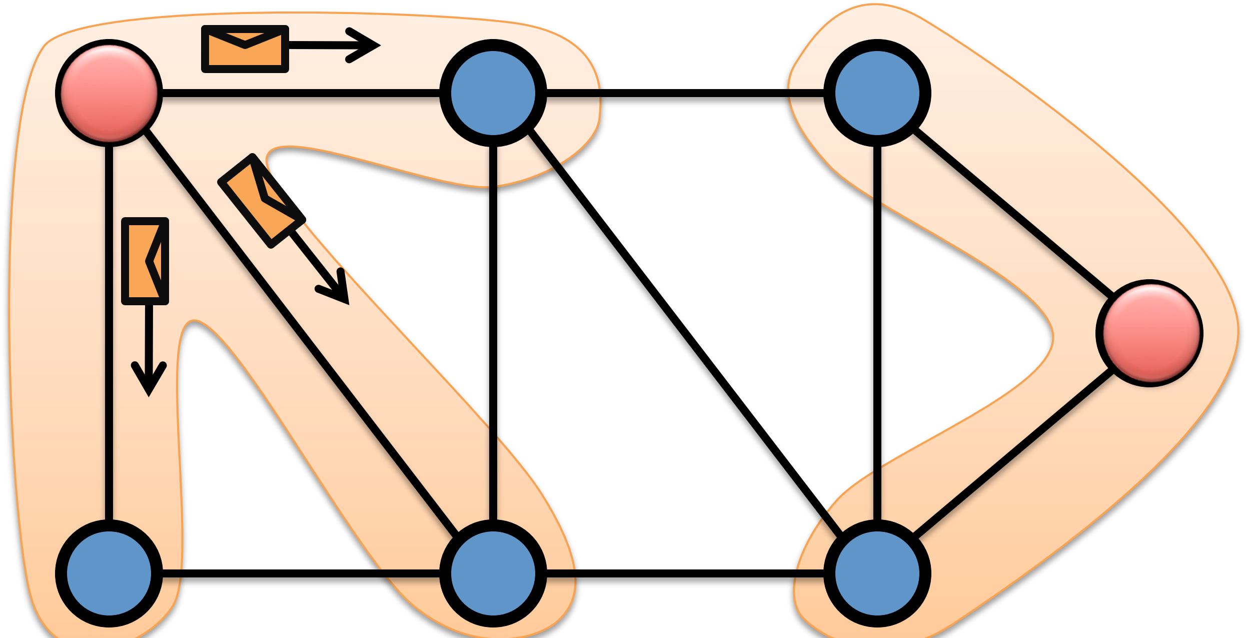
- 2 parallelization strategies:
  - Each thread calculates disjoint R[i], need to perform random (remote) reads for R[j] → too slow
  - Each thread works on disjoint R[j], computes $W_{j,i} * R[j]$, increments R[i] += $W_{j,i} * R[j]$, need to perform synchronized remote writes for R[i] → too slow

# Distributed Computation



# The **Graph-Parallel** Abstraction

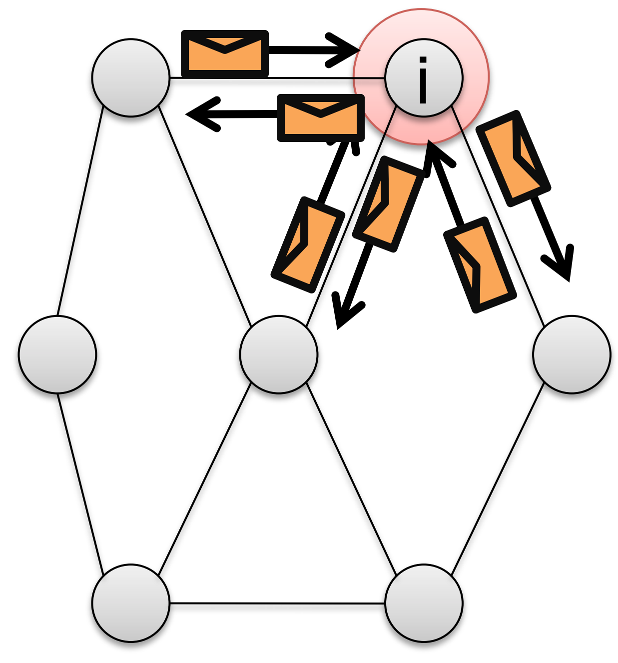
- A user-defined **Vertex-Program** runs on each vertex
- **Graph** constrains **interaction** along edges
  - Using **messages** (e.g. **Pregel** [PODC'09, SIGMOD'10])
  - Through **shared state** (e.g., **GraphLab** [UAI'10, VLDB'12])
- **Parallelism**: run multiple vertex programs simultaneously



# 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
```

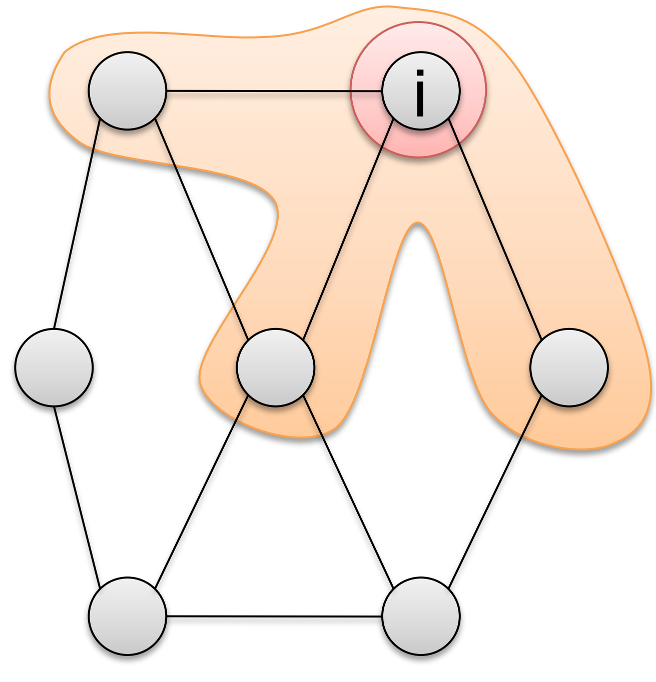



Malewicz et al. [PODC'09, SIGMOD'10]

# The GraphLab Abstraction

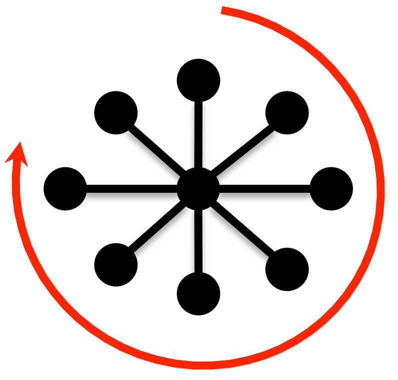
Vertex-Programs directly **read** the neighbors state

```
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
```

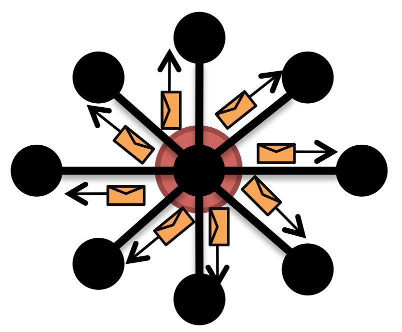



Low et al. [UAI'10, VLDB'12]

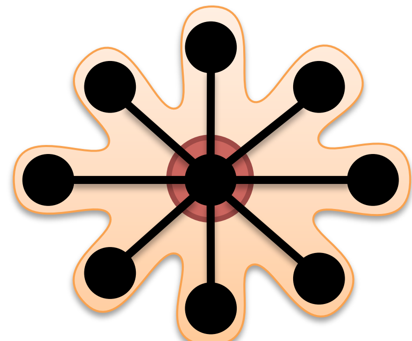
# Challenges of **High-Degree** Vertices



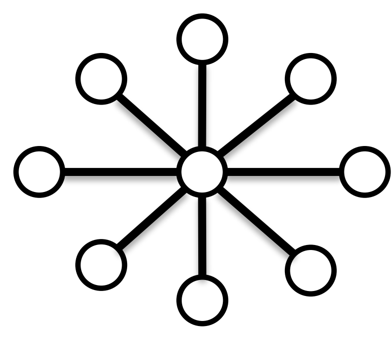
Sequentially process edges



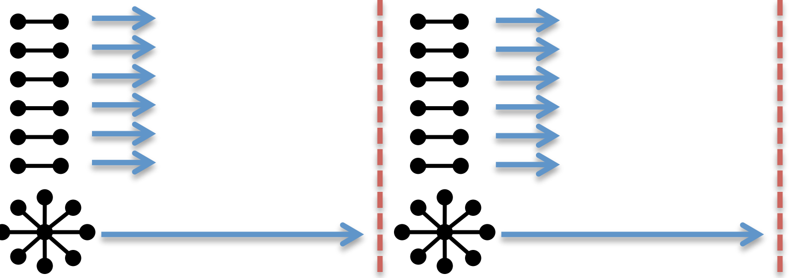
Sends many messages (Pregel)



Touches a large fraction of graph (GraphLab)



Edge meta-data too large for single machine

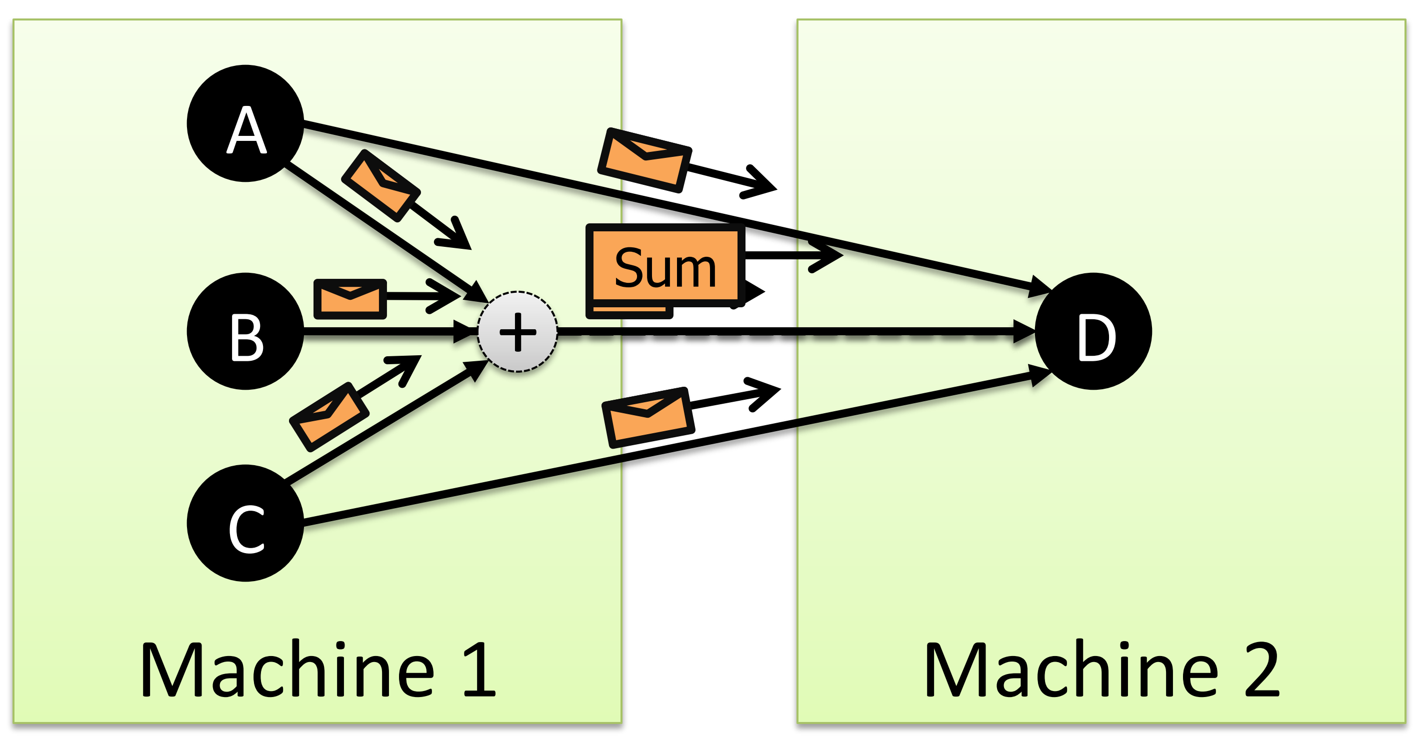


Synchronous Execution prone to stragglers (Pregel)

# Communication Overhead for High-Degree Vertices

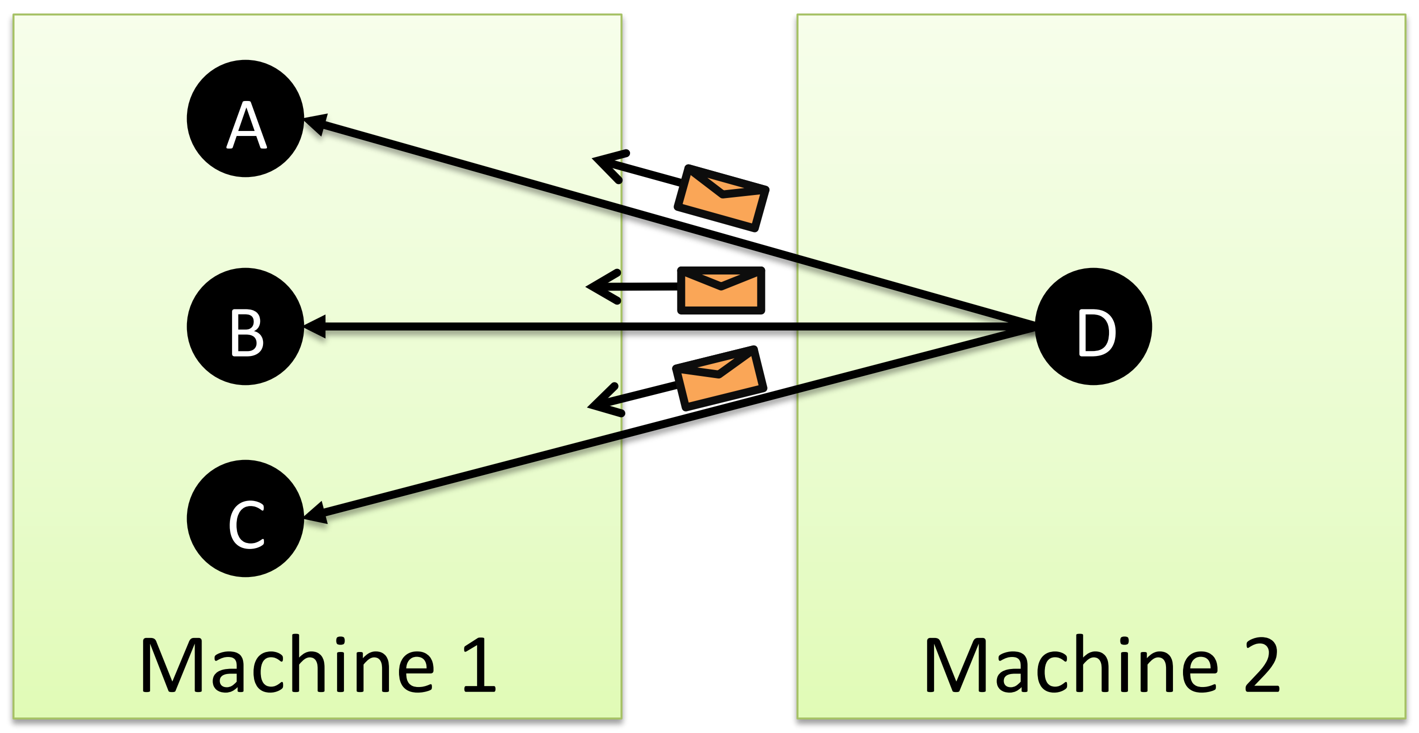
Fan-In vs. Fan-Out

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




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

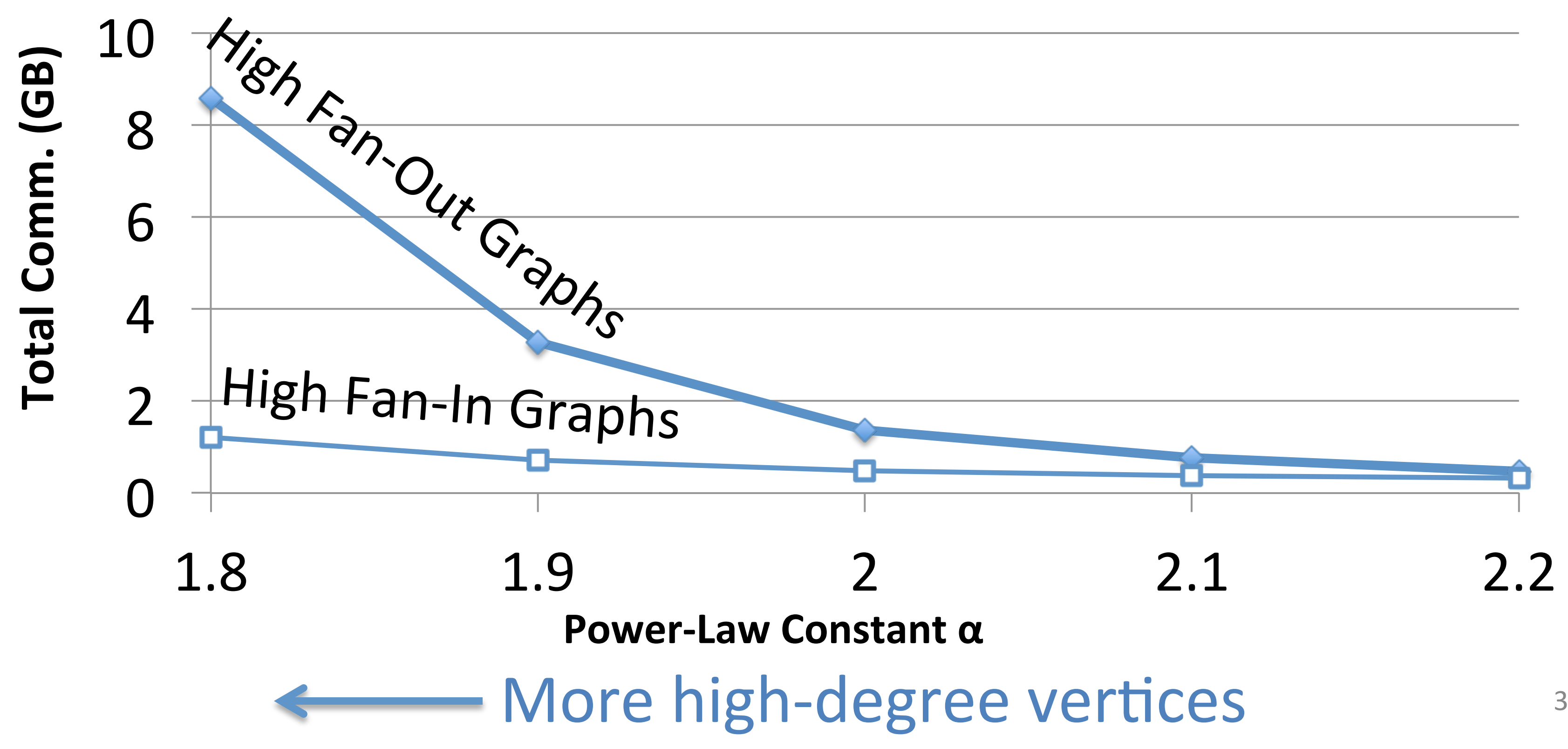
# Pregel Struggles with **Fan-Out**



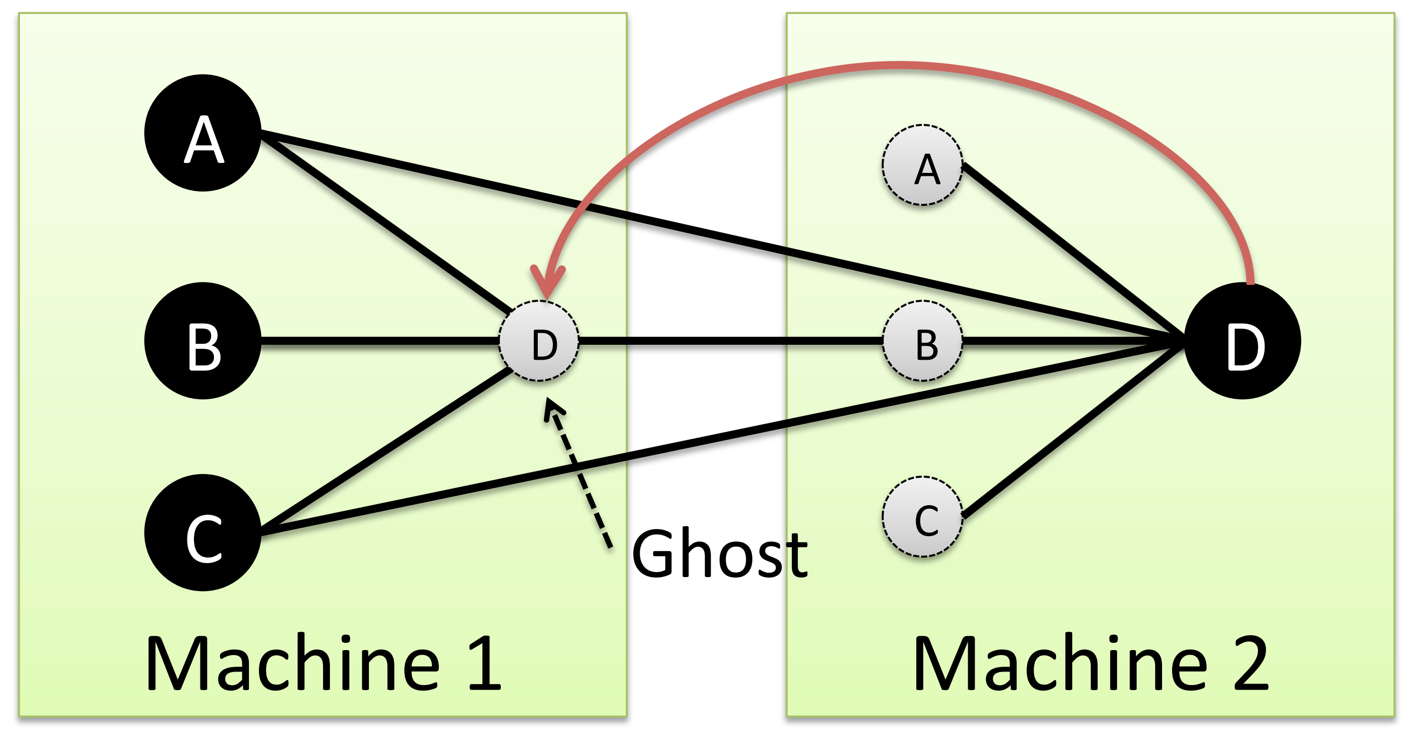

- **Broadcast** sends many copies of the same message to the same machine!

# Fan-In and Fan-Out Performance

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

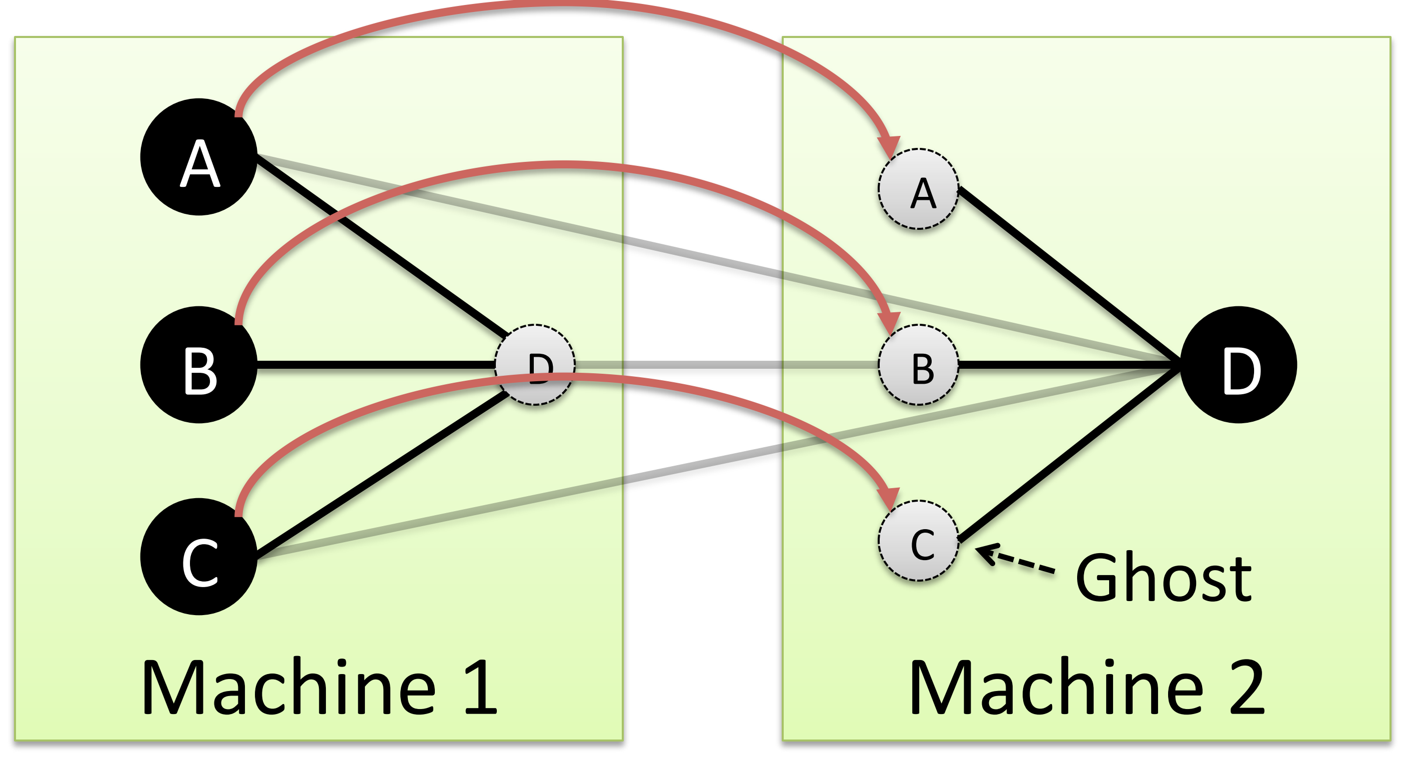


# GraphLab Ghosting




- Changes to master are synced to ghosts

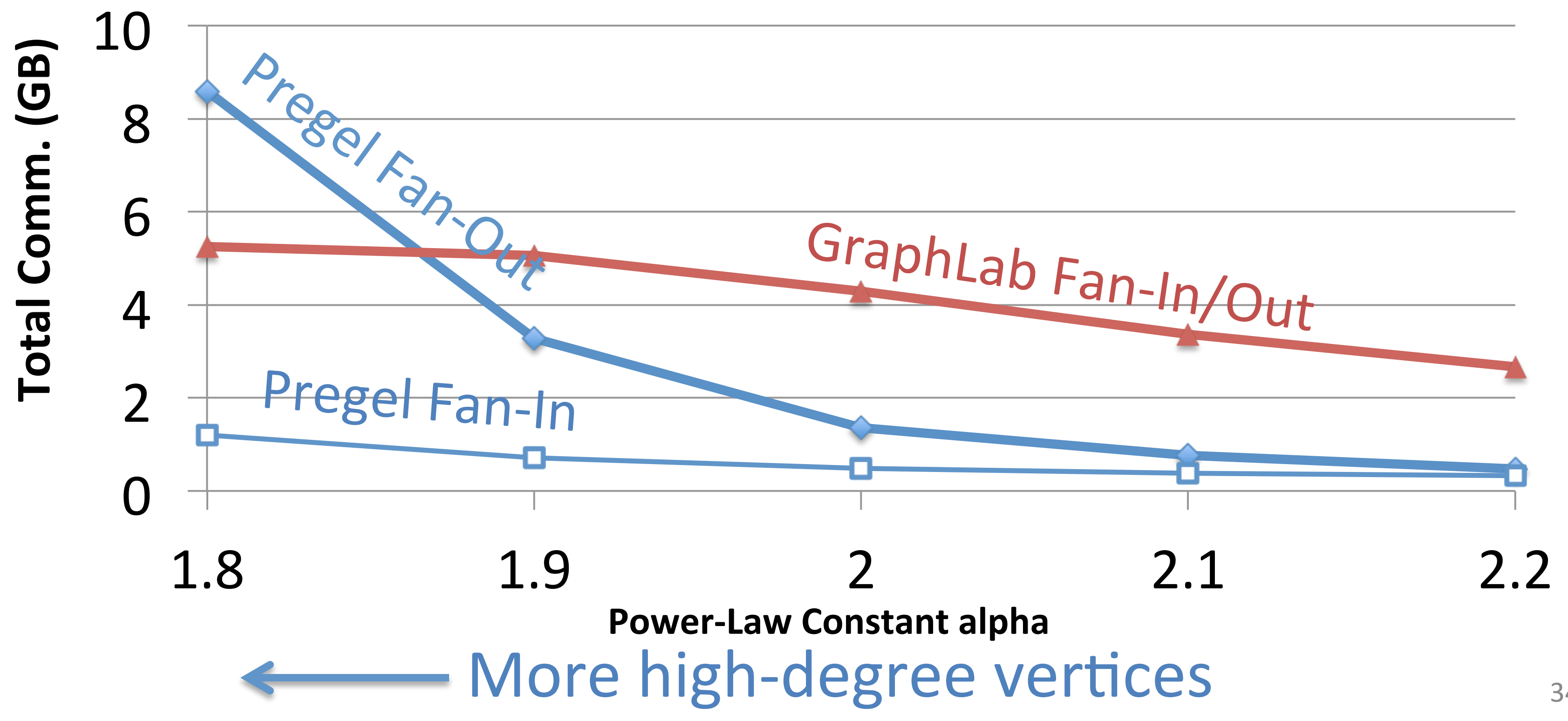
# GraphLab Ghosting




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

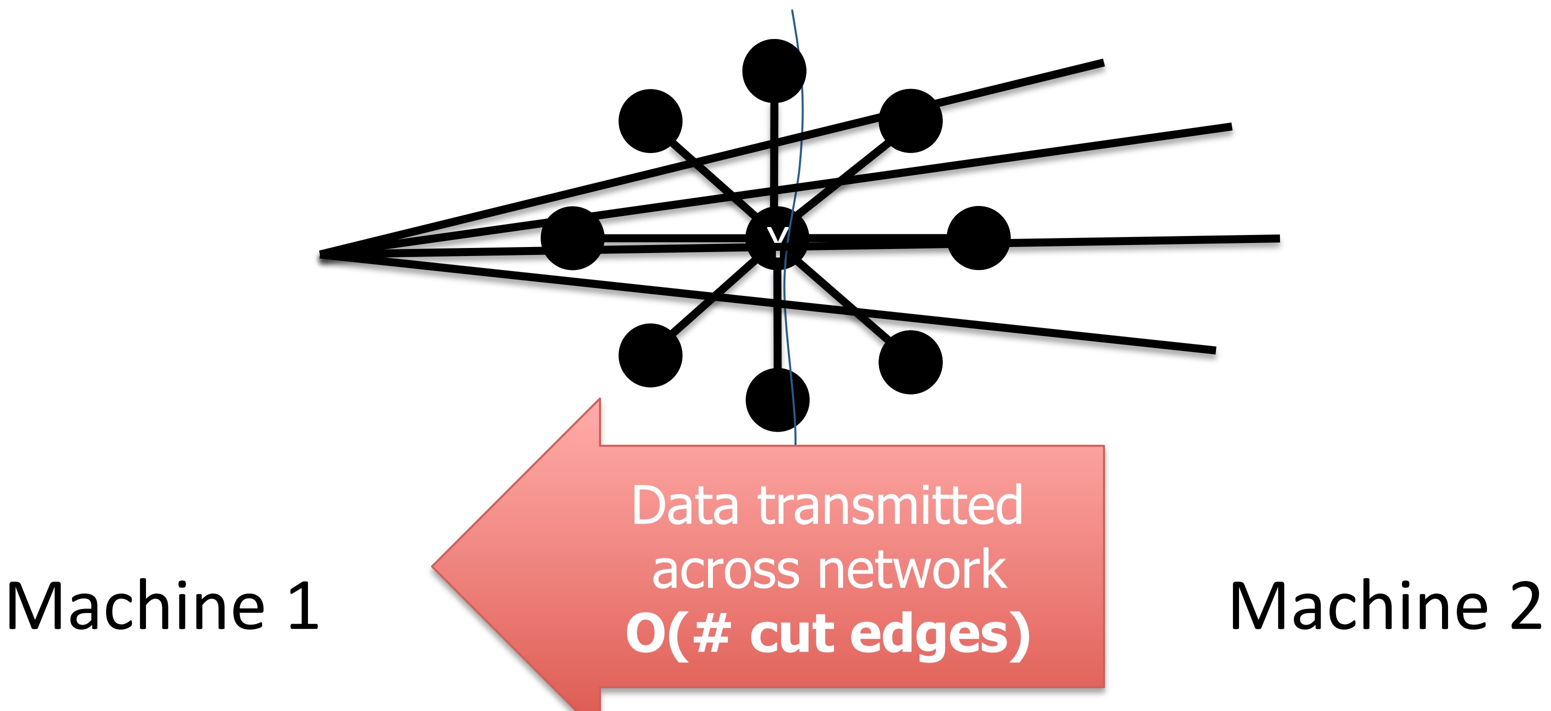
# Fan-In and Fan-Out Performance

- 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!**

# PowerGraph at a high level

- How to partition graph-computation in the face of **high-degree vertices?**
- Contributions:
  - **GAS programming model**
    - allows a single high-degree vertex to be parallelized
  - **Vertex partitioning**
    - assign edges (instead of nodes) to machines

# 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**

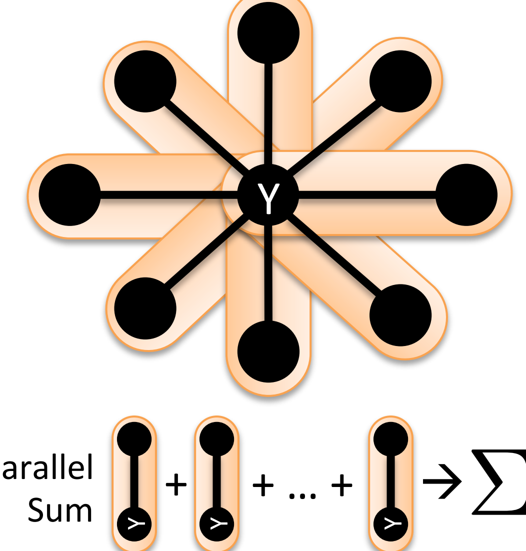
# GAS Decomposition

## 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'

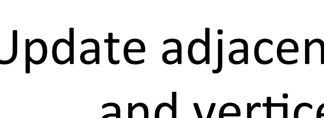
Y

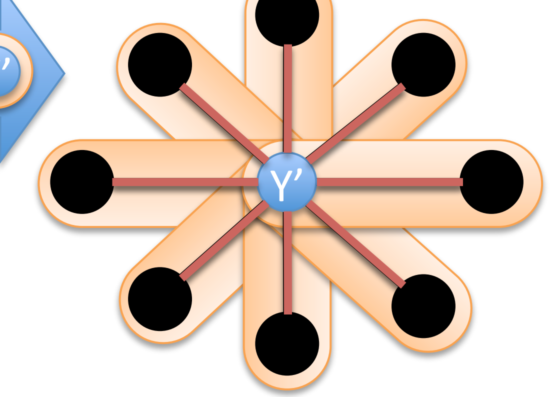
## Scatter

Update adjacent edges and vertices.

***User Defined:***

- **Scatter**( ) →






Update Edge Data & Activate Neighbors

# PageRank in PowerGraph

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

**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 ) :

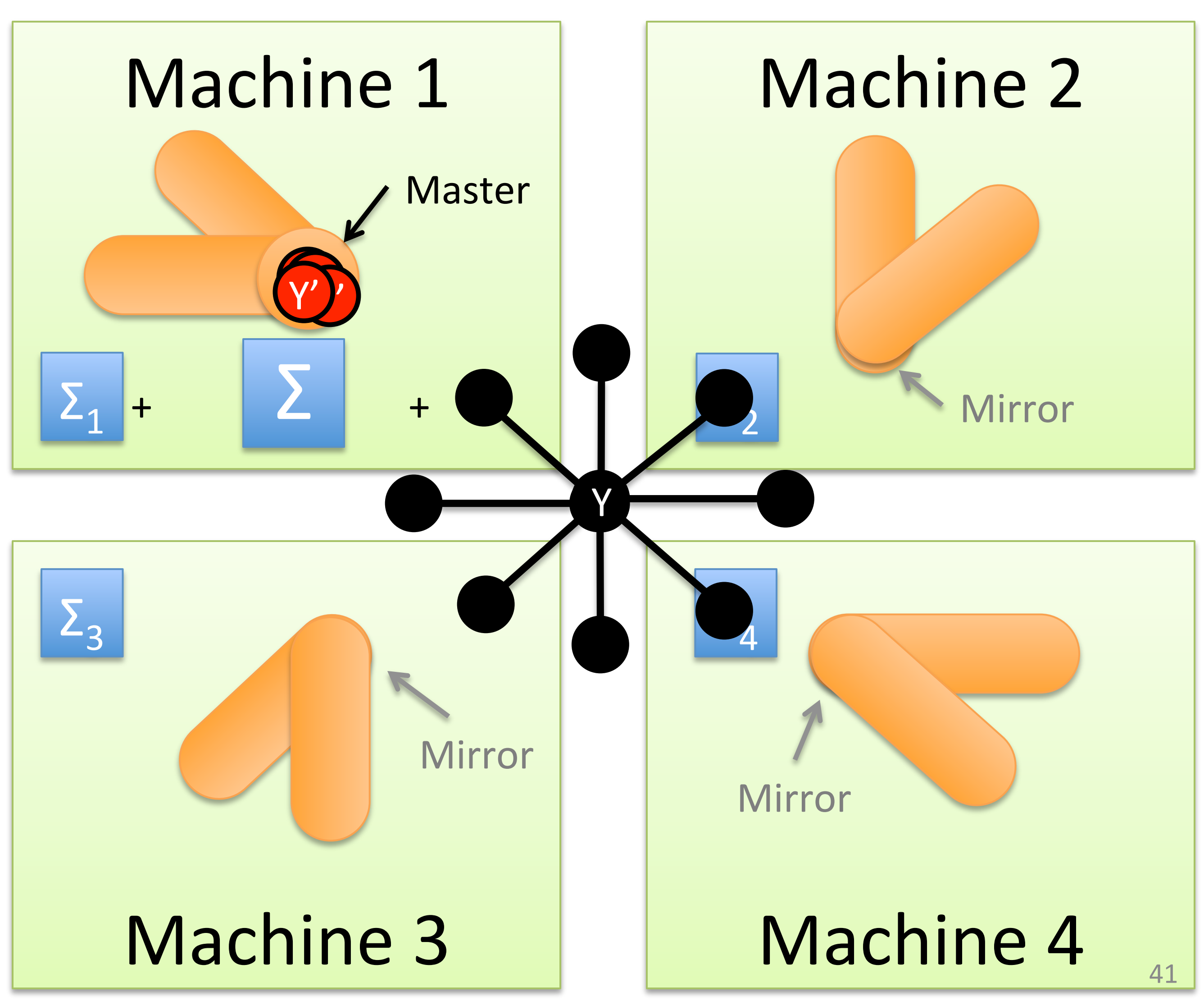
if *R[i]* changed then trigger *j* to be **recomputed**

# Distributed Execution of a PowerGraph Vertex-Program

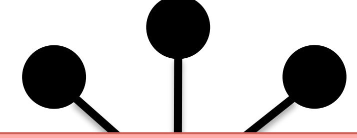
**G**ather

**A**pply

**S**catter



# Minimizing Communication in PowerGraph



Communication is linear in the number of machines each vertex spans

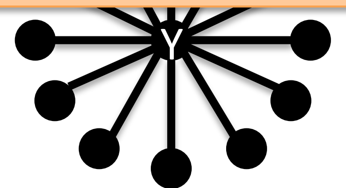
A **vertex-cut** minimizes machines each vertex spans

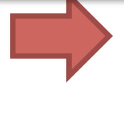
# New Approach to Partitioning

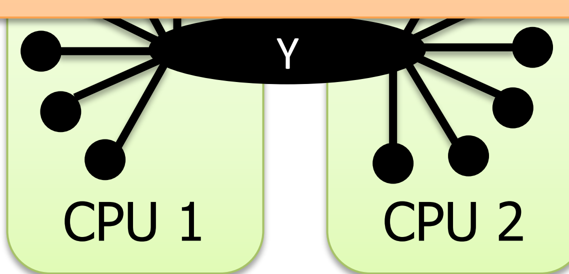
- Rather than cut edges:

*For **any edge-cut**, one 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

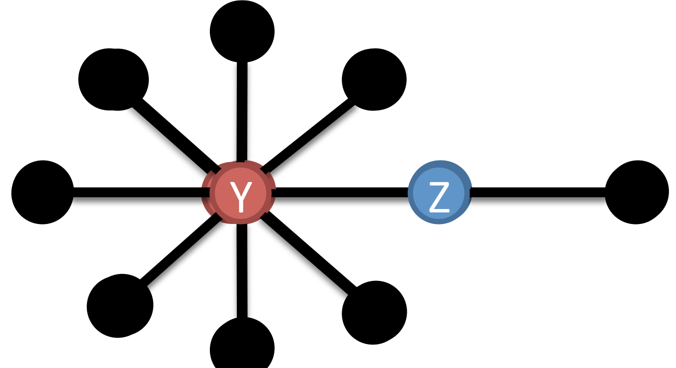
Machine 3

**Balanced Vertex-Cut**

Y Spans 3 Machines

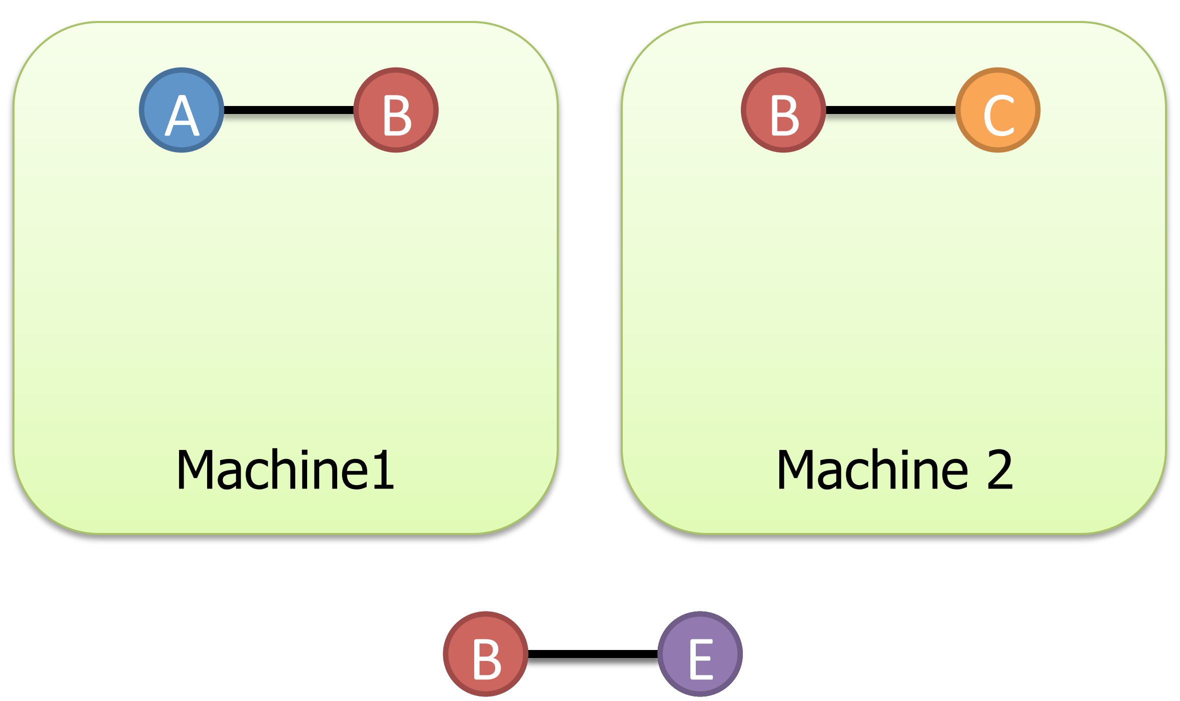
Z Spans 2 Machines

**Not cut!**



# Greedy Edge Placements

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

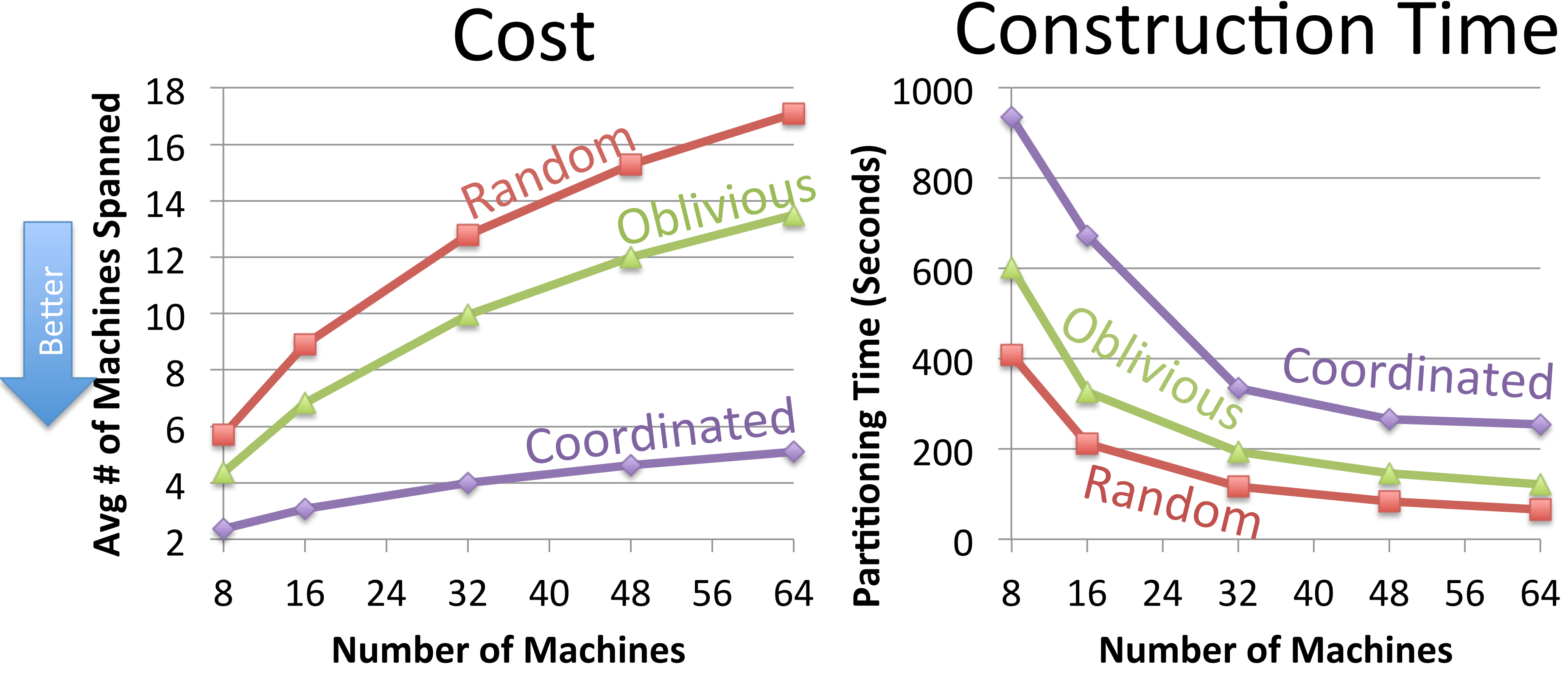


# Greedy Edge Placements

- **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

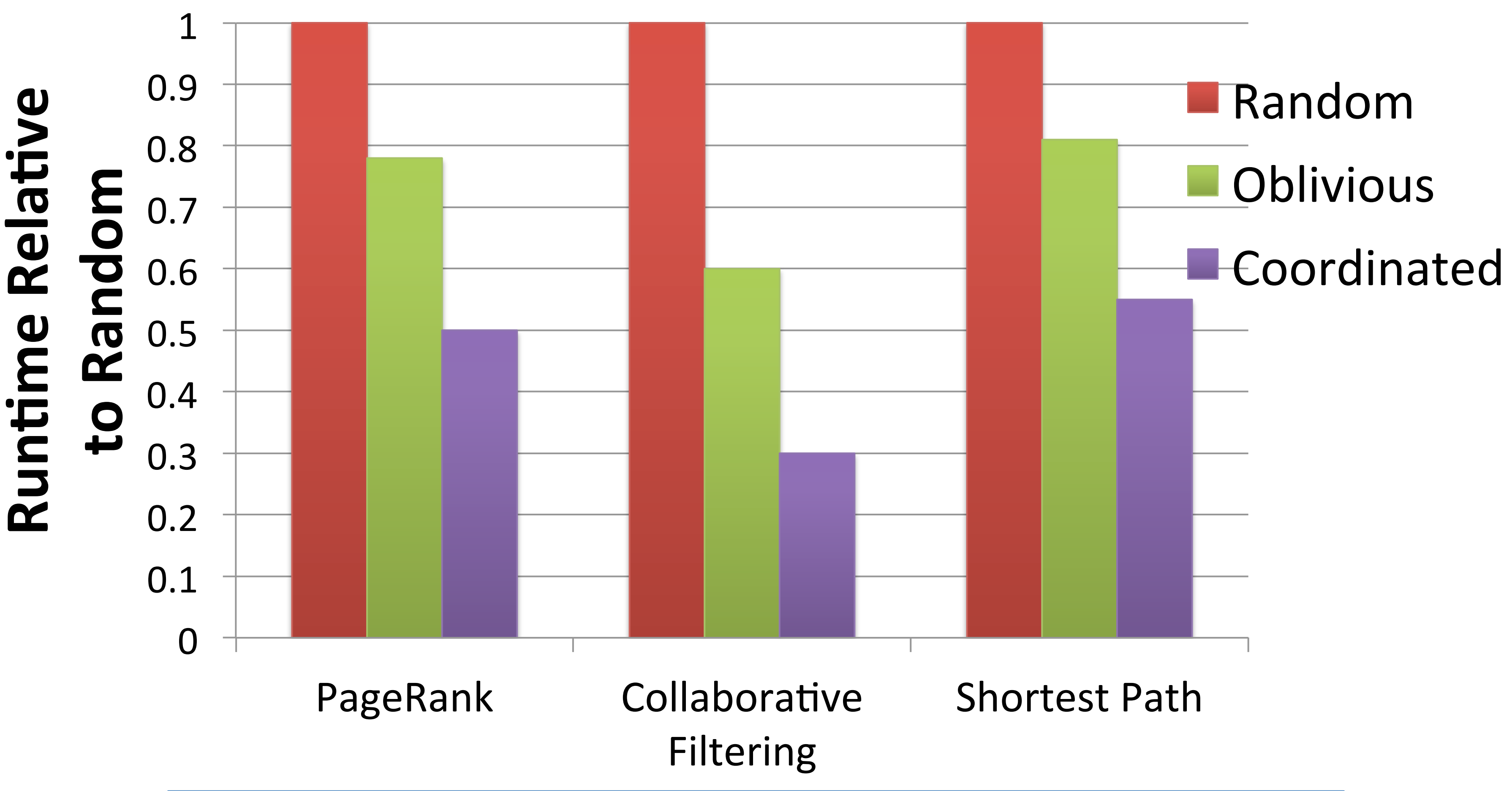
# Partitioning Performance

**Twitter Graph:** 41M vertices, 1.4B edges




**Oblivious** balances cost and partitioning time.

# Greedy Vertex-Cuts Improve Performance




**Greedy partitioning improves computation performance.**

# Summary

- DSM: use the same general single-machine model for distributed computation
  - use release consistency to improve performance
  - still hard to hide the performance difference between local and remote memory
- Graph Framework: "shared memory", but specialized for graph computation