Complex Data Processing

Rong Chen

Institute of Parallel and Distributed Systems

Shanghai Jiao Tong University

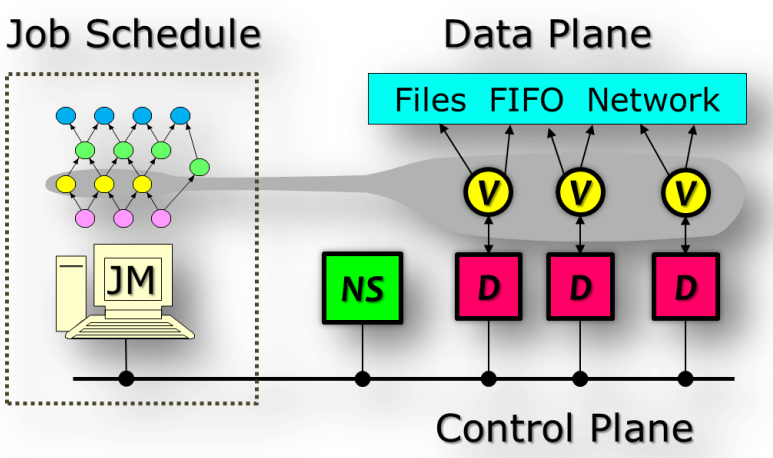
http://ipads.se.sjtu.edu.cn/rong_chen

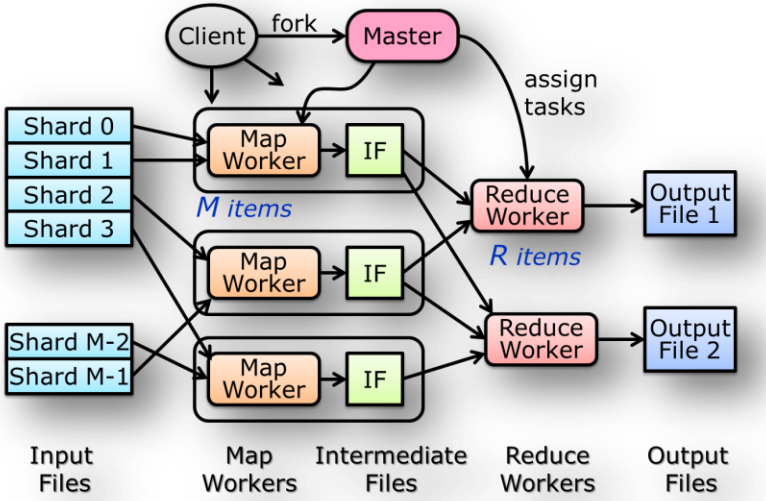
Credits: some slides from Joseph Gonzalez (CMU) and Yucheng Low (CMU)

Concurrency so far

Data-parallel Programming Model

- MapReduce & Dryad
  - Parallelism
  - Load balance
  - Locality and data distribution
  - Fault tolerance





Big Data already Happened

1 Billion Tweets Per Week

750 Million Facebook Users

6 Billion Flickr Photos

48 Hrs of Video Per Minute

Big Learning


The New York Times
SundayReview

WORLD U.S. N.Y. / REGION BUSINESS TI

NEWS ANALYSIS

The Age of Big Data

By STEVE LOHR

Published: February 11, 2012

"...growing at 50 percent a year..."

"... data a new class of economic asset, like currency or gold."

How do we understand and use Big Data?

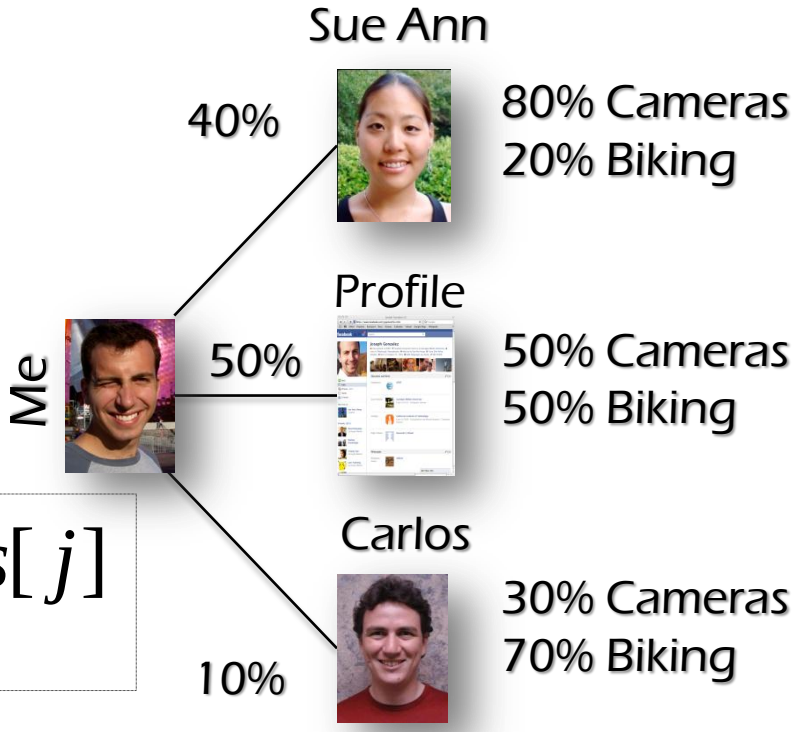
Dig Learning Example

Social Arithmetic

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

I Like: 60% Cameras, 40% Biking

Label Propagation



Recurrence Algorithm

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

iterate until convergence

Parallelism: Compute all Likes[i] in parallel

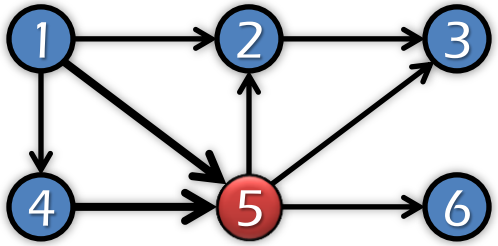
http://www.cs.cmu.edu/~zhuxj/pub/CMU-CALD-02-107.pdf

Dig Learning Example

PageRank

http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf

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



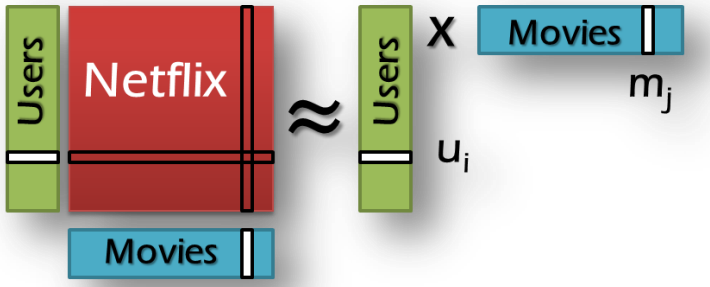
α is the random reset probability

L[j] is the number of links on page j

Alternating Least Squares

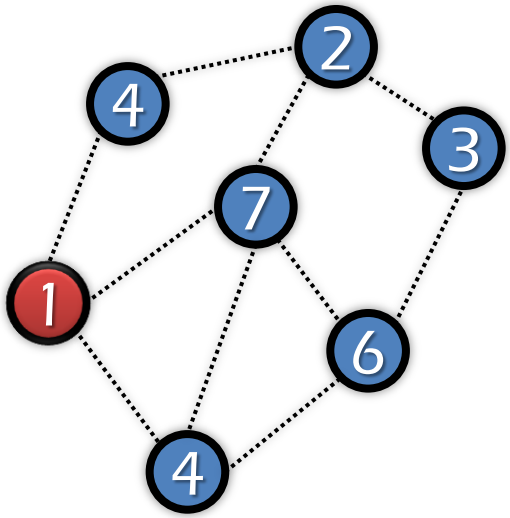
$$u_i = \arg\min_w \sum_{j\in N[i]} (r_{ij} - m_j \cdot w)^2$$

$$m_j = \arg\min_w \sum_{i\in N[j]} (r_{ij} - u_i \cdot w)^2$$

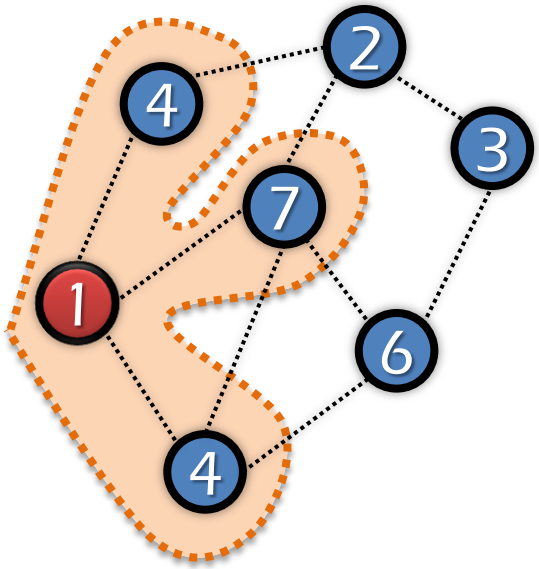


http://dl.acm.org/citation.cfm?id=1424269

Graph Parallel Algorithm



Dependency
Graph



Local
Updates

Iterative
Computation

What is the Right Tool
Graph-parallel ML

Data-Parallel

MapReduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

MapReduce

Lasso

PageRank

Label Propagation

Kernel Methods

Tensor Factorization

Belief Propagation

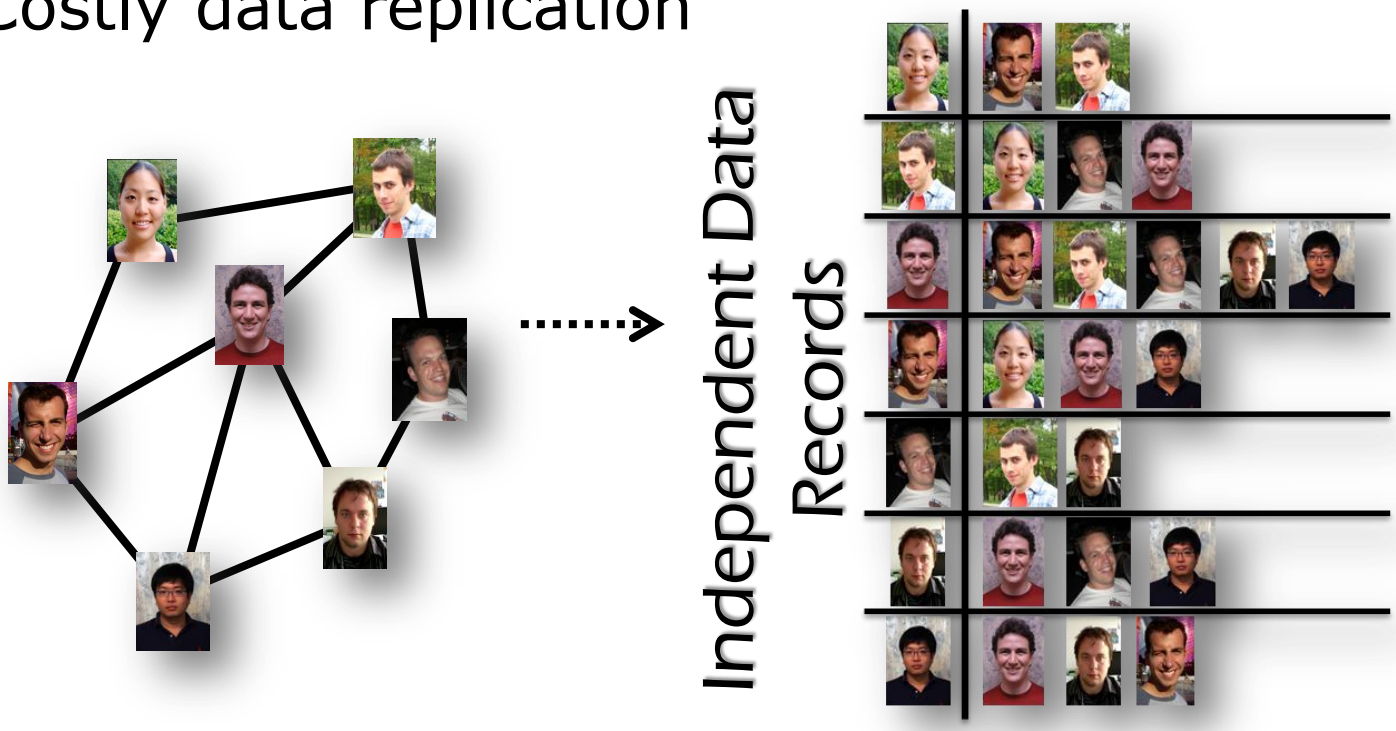
Deep Belief Networks

Neural Networks

Dependencies are Difficult

Difficult to express dependent data in MR

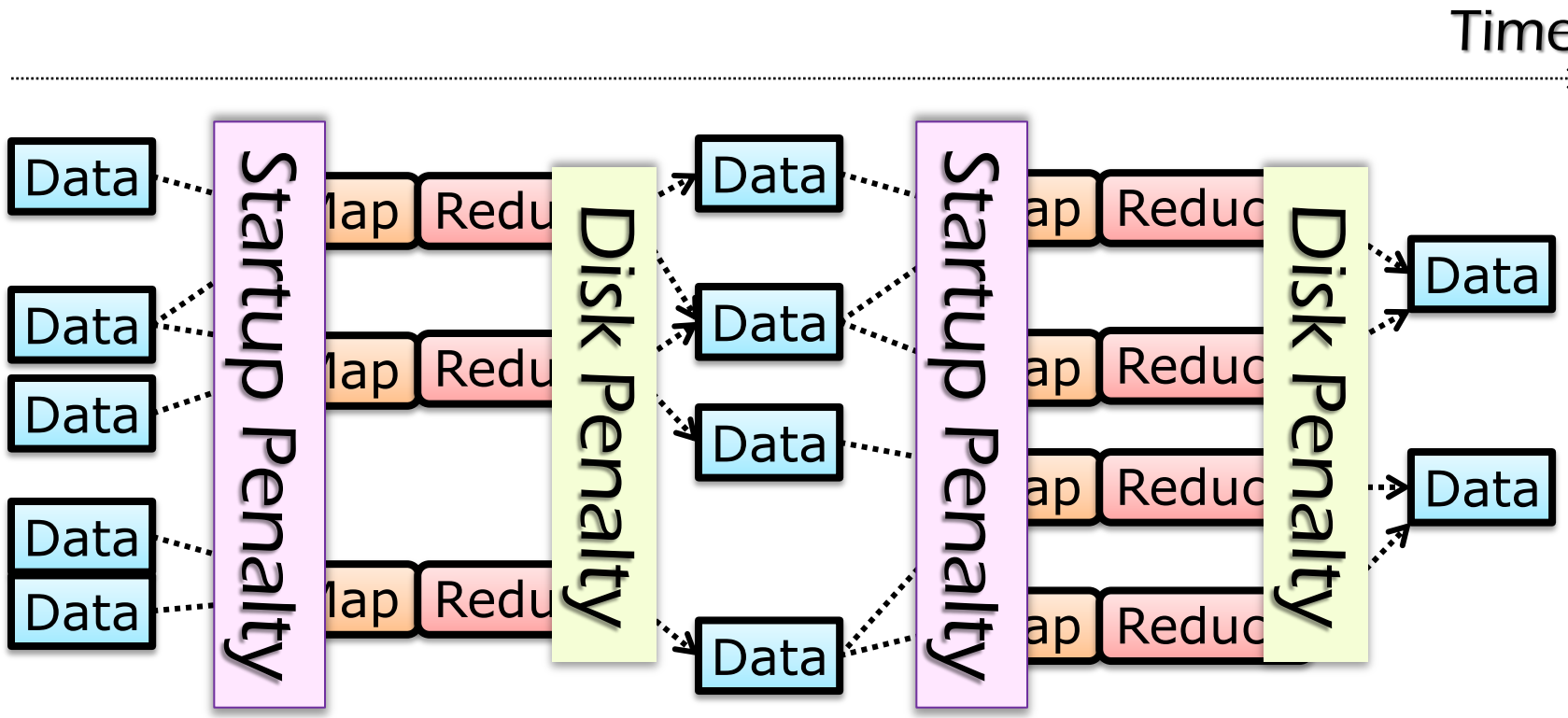
- Substantial data transformations
- User managed graph structure
- Costly data replication



Iterative Comp. is Difficult

MR runtime is not optimized for iteration

- Persistent I/O (e.g., GFS)



What is the Right Tool
Graph-parallel ML

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

MPI/Pthreads

Lasso

PageRank

Label Propagation

Kernel Methods

Tensor Factorization

Belief Propagation

Deep Belief Networks

Neural Networks

Threads, Locks and Messages

ML experts repeatedly solve the same parallel design challenges

- Impl. and debug complex parallel systems
- Tune for a specific parallel platform

Threads, Locks and Messages

Graduate Student

~~ML experts~~ repeatedly solve the **same** parallel design challenges

- Impl. and debug complex parallel systems
- Tune for a specific parallel platform

6 months later, the conference paper contains:

> *"We implemented ____ in parallel."*

The resulting **code**

- difficult to **maintain** and **extend**

couples **learning model** to **parallel impl.**

What is the Right Tool
Graph-parallel ML

Data-Parallel → Graph-Parallel

Map Reduce

Feature Extraction

Cross Validation

Computing Sufficient Statistics

Lasso

PageRank

Label Propagation

Kernel Methods

Tensor Factorization

Belief Propagation

Deep Belief Networks

Neural Networks

GraphLab Framework

Graph Based
Data Representation

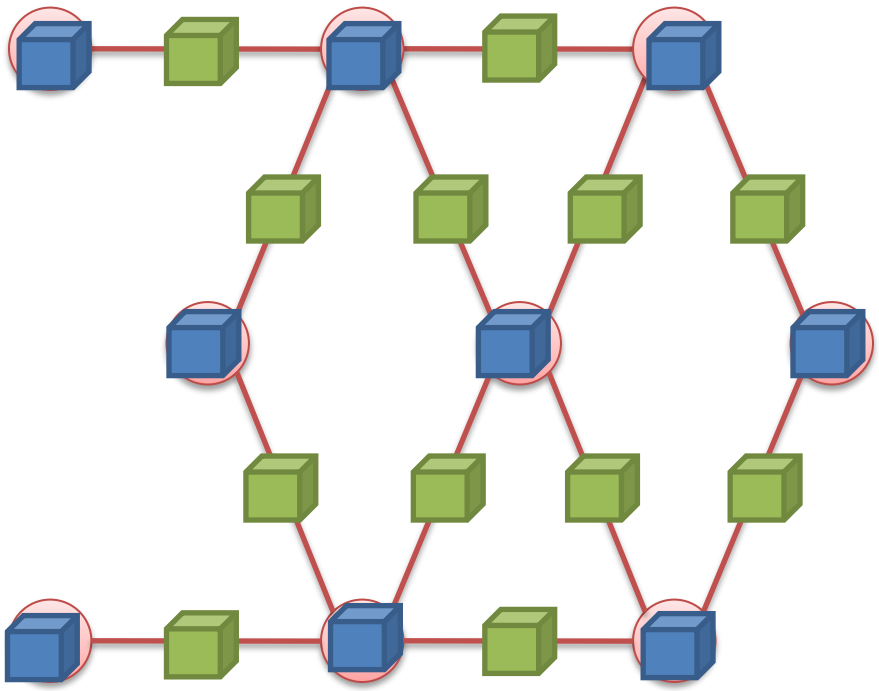
Update Functions
User Computation

Scheduler

Consistency Model

Data Graph

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



Graph: Social Network

Vertex Data:

1. User profile text
2. Current interests estimates

Edge Data:

1. Similarity weights

Implementation

All data and structure is stored in memory

- Supports fast random lookup needed for dynamic computation

Multicore Setting

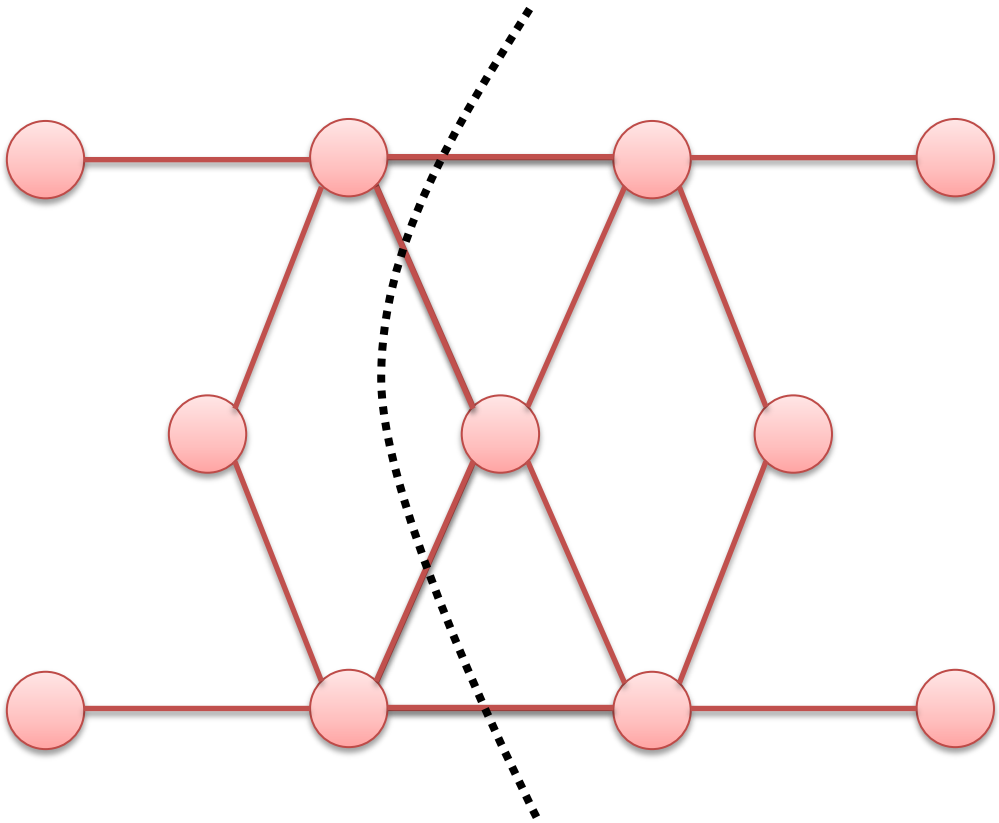
- Challenge: Fast lookup, low overhead
- Solution → dense data-structures (e.g., array)

Distributed Setting

- Challenge: Graph partitioning
- Solution → ParMetis and Random placement

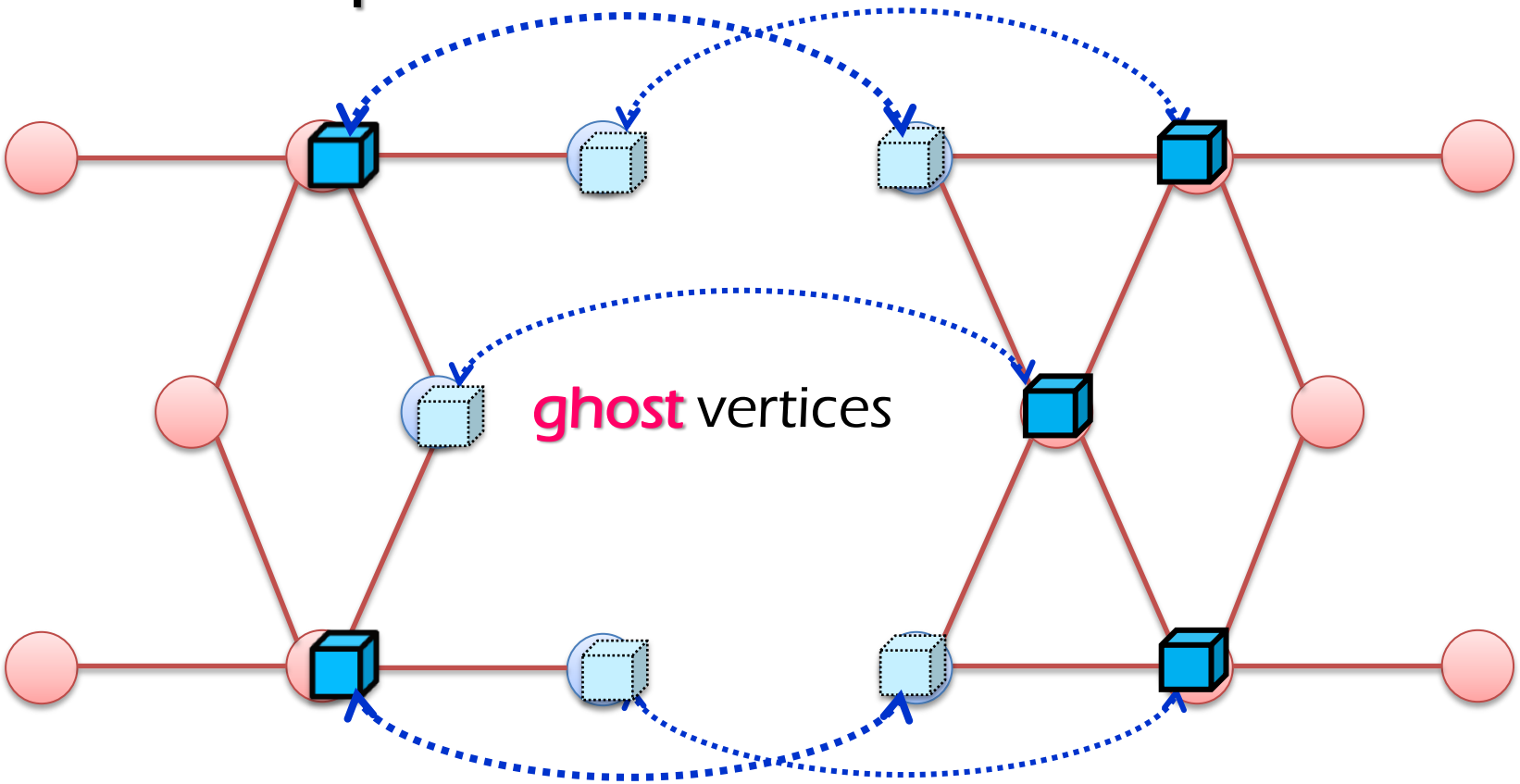
Distributed Graph

Partition the graph across multiple machines



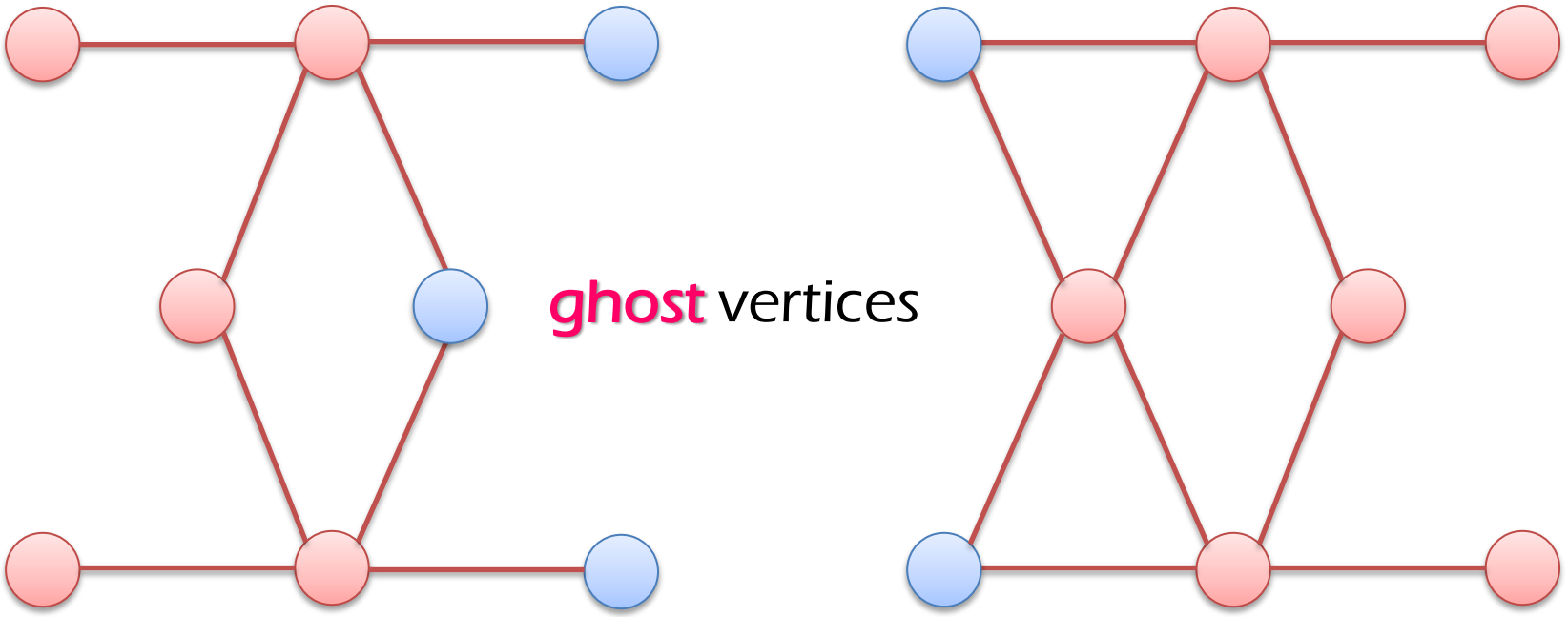
Distributed Graph

Ghost vertices maintain adjacency structure and replicate remote data



Distributed Graph

Cut **efficiently** using HPC Graph partitioning tools (e.g., ParMetis, …)



GraphLab Framework

Graph Based
Data Representation

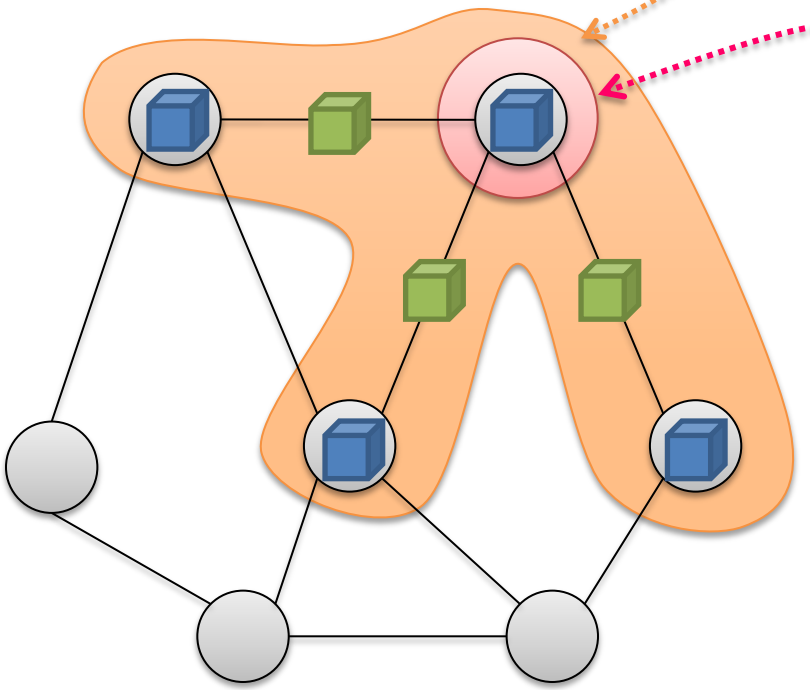
Update Functions
User Computation

Scheduler

Consistency Model

Update Function

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

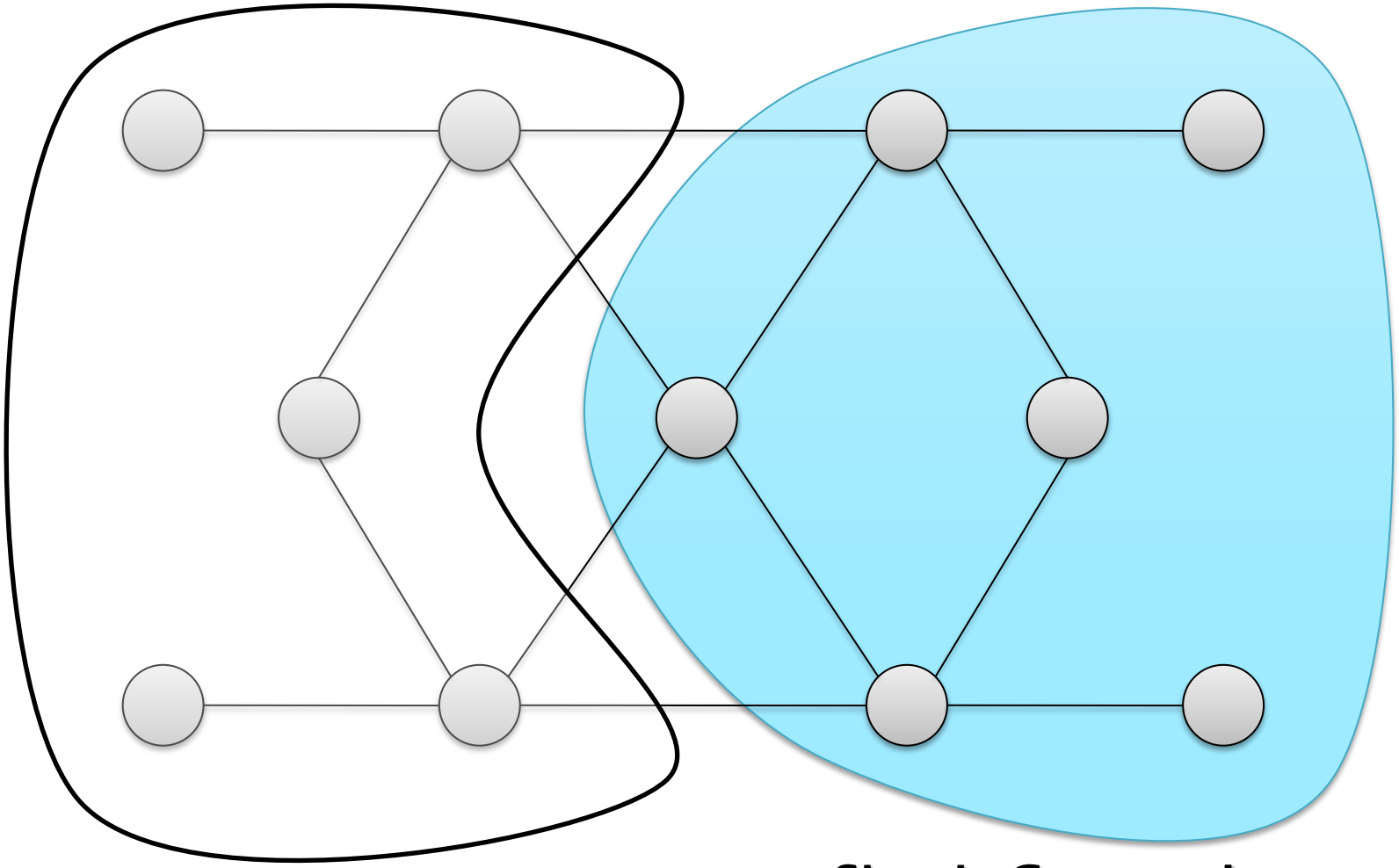


```
void PageRank(scope){
  // Get Neighborhood data
  float sum = 0, diff;
  float last = this.val;
  foreach (nbr in scope.in_nbrs)
    sum += nbr.val/nbr.nout_nbrs;

  // Update the vertex data
  this.val = 0.15 + 0.85*sum;

  // Reschedule Nbrs if needed
  diff = abs(this.val - last);
  if (diff > epsilon)
     reschedule(this.out_nbrs);
}
```

Dynamic Computation



GraphLab Framework

Graph Based
Data Representation

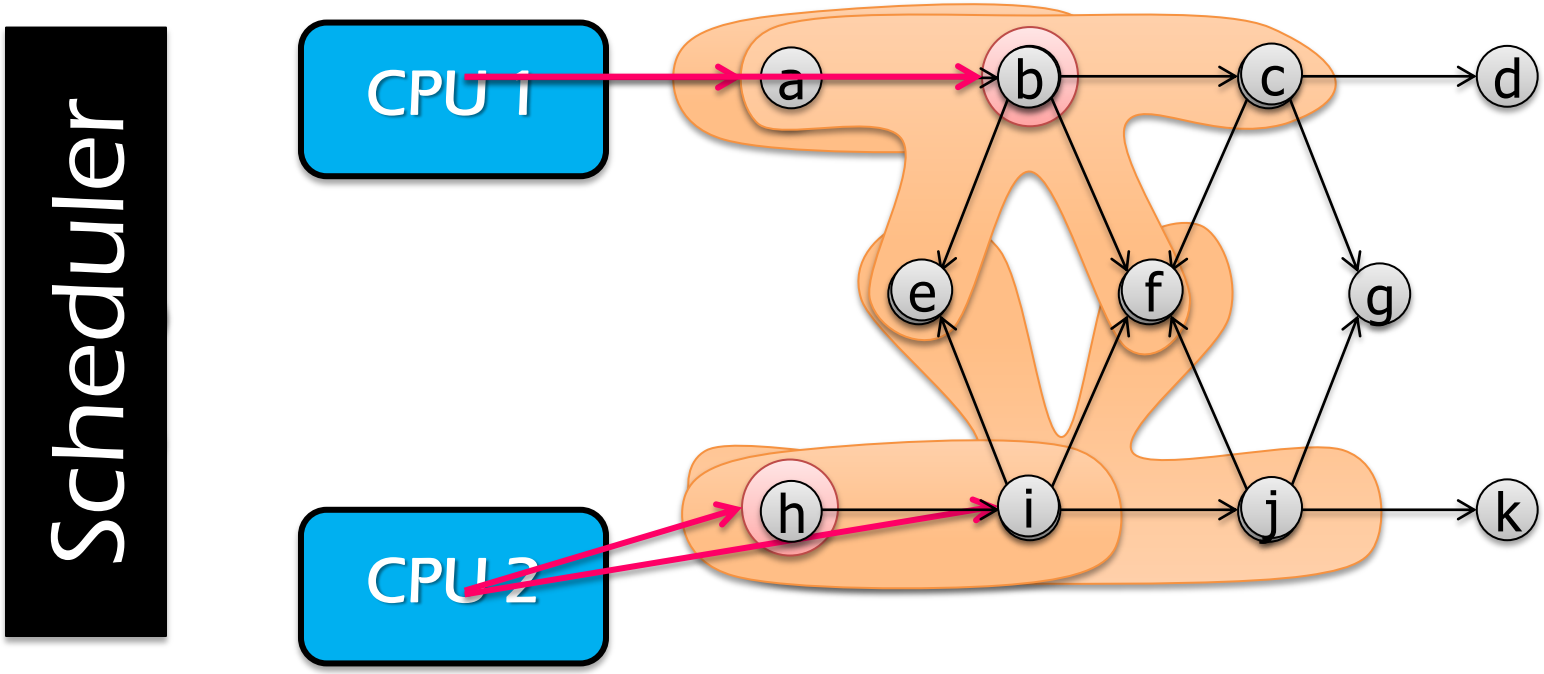
Update Functions
User Computation

Scheduler

Consistency Model

GraphLab Scheduler

The scheduler determines the order that vertices are updated



The process repeats until the scheduler is empty

GraphLab Scheduler

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

GraphLab provides several diff schedulers

- Vertices are updated

  → **Round Robin**, **FIFO**, **Priority**

GraphLab Framework

Graph Based
Data Representation

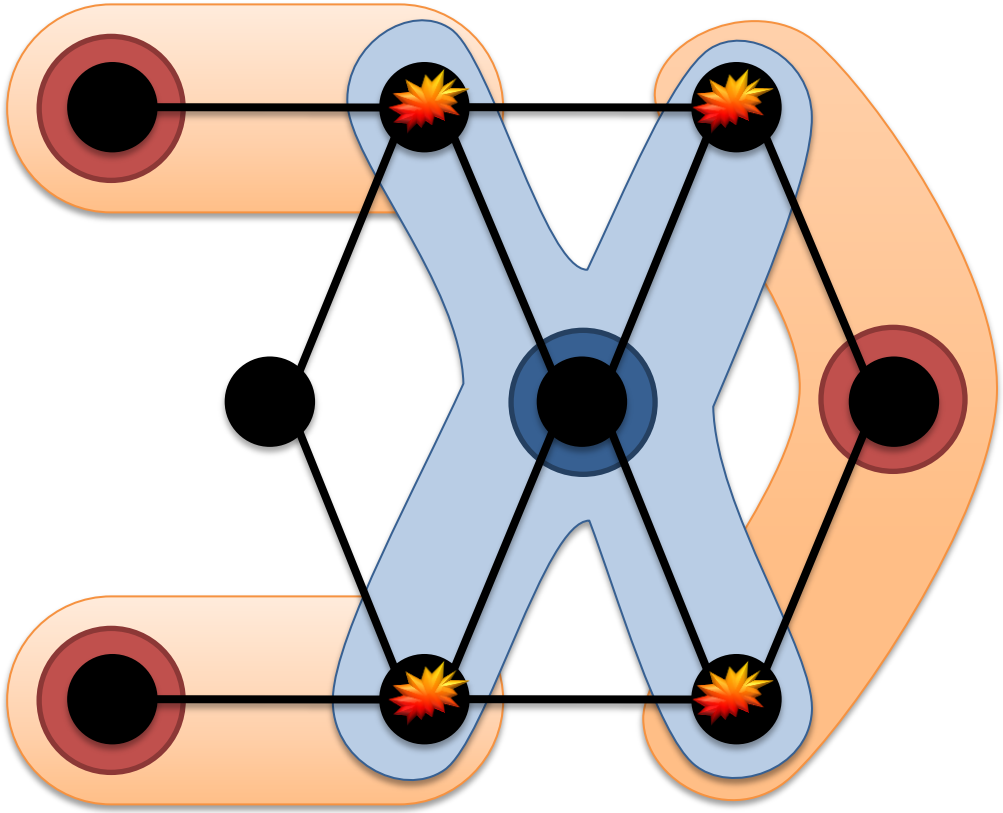
Update Functions
User Computation

Scheduler

Consistency Model

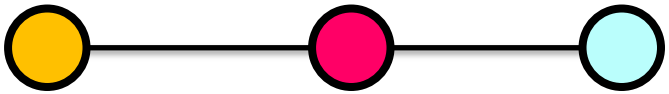
Race-free Execution

How much can computation overlap?

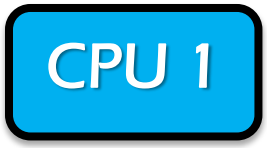


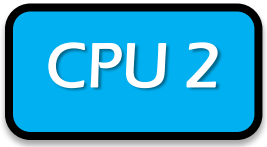
Sequential Consistency

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

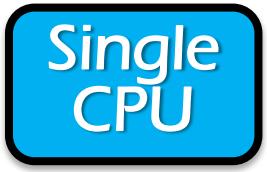


Parallel

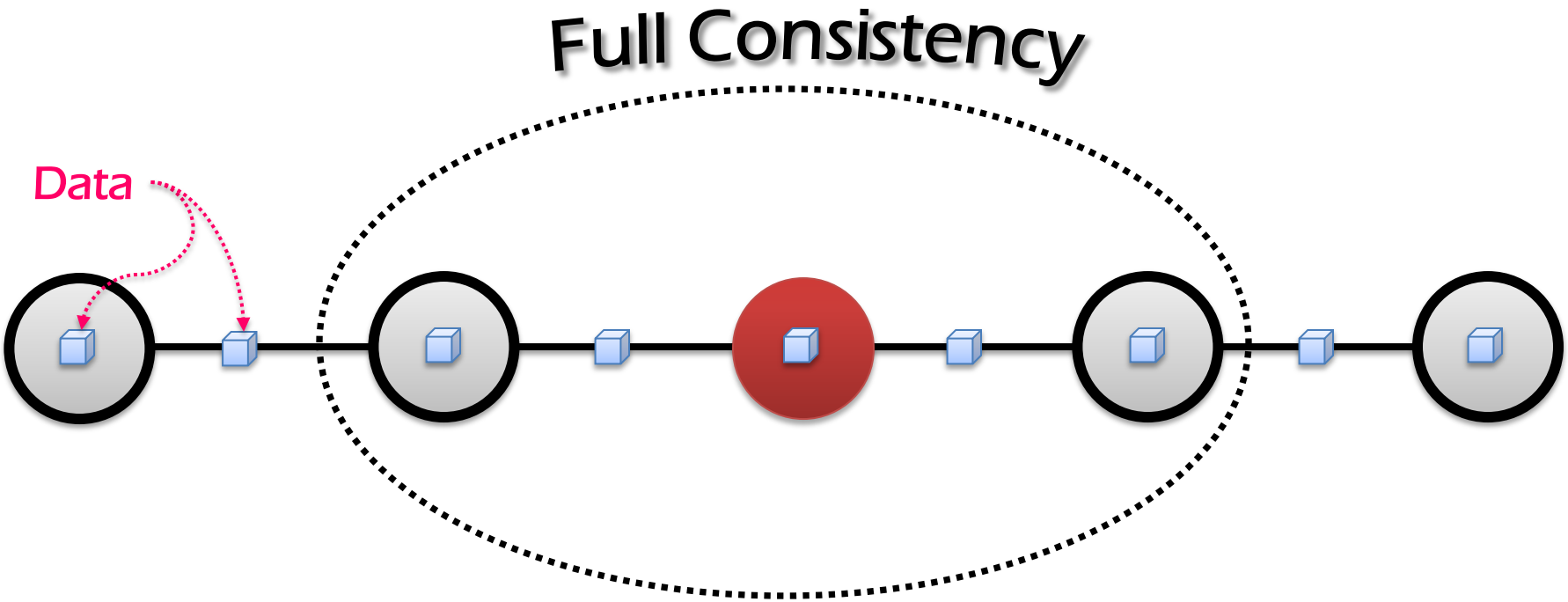




Sequential

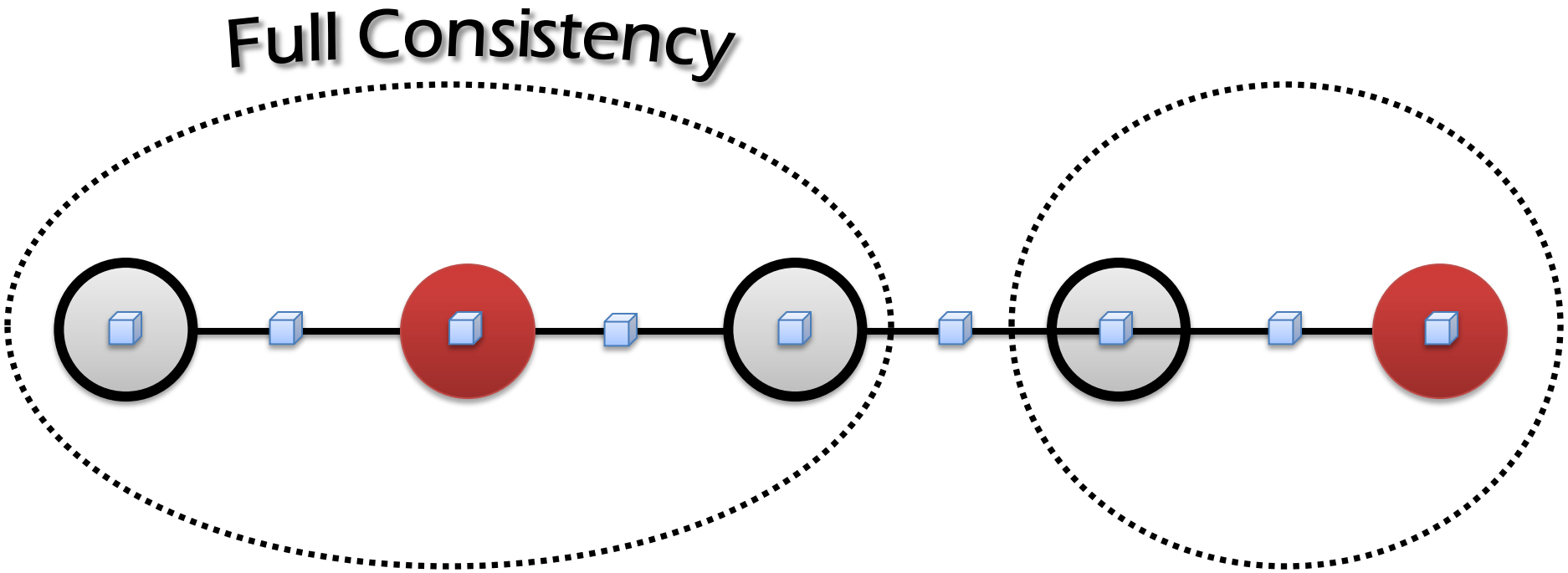


Consistency Rule

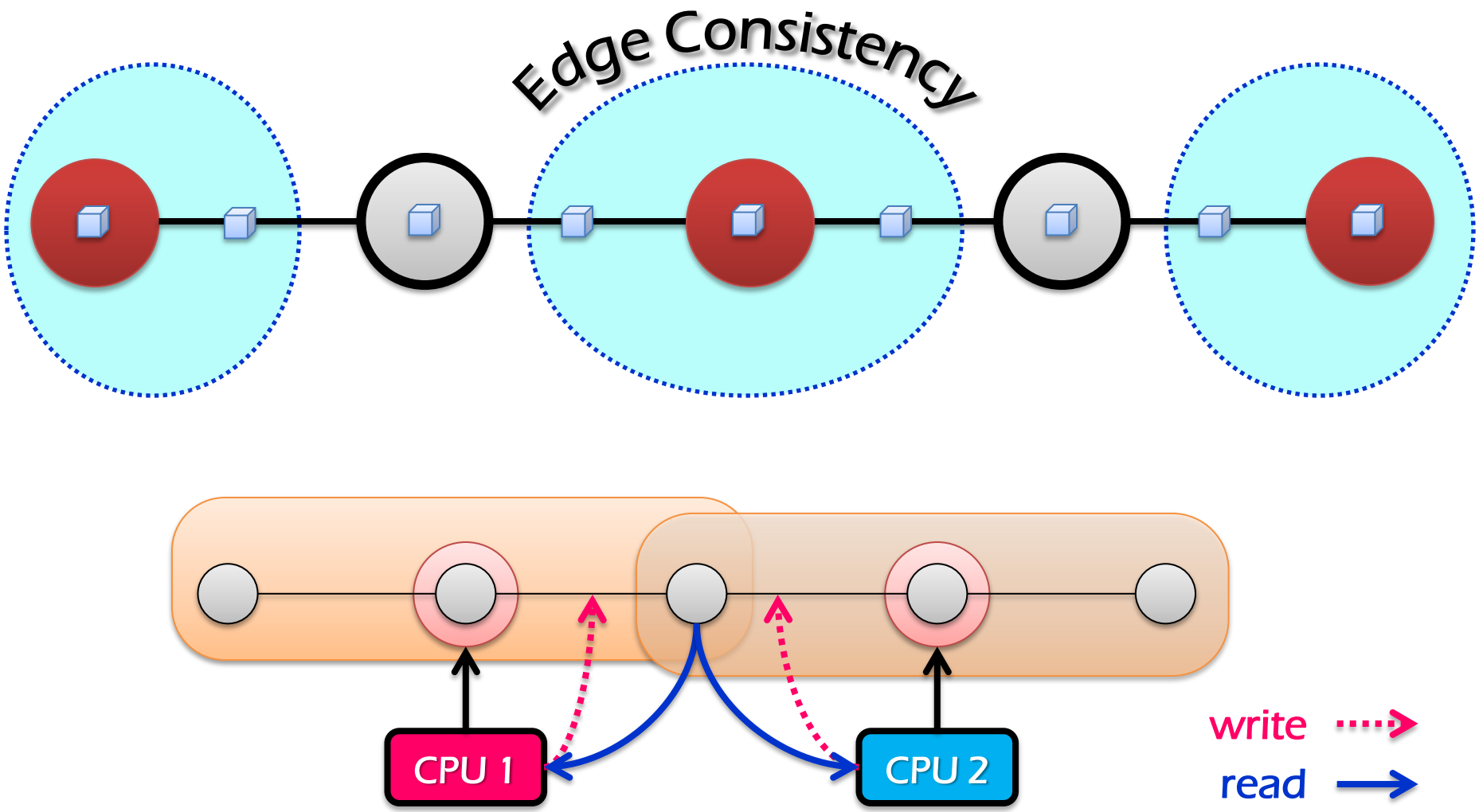


Guaranteed sequential consistency for all update functions

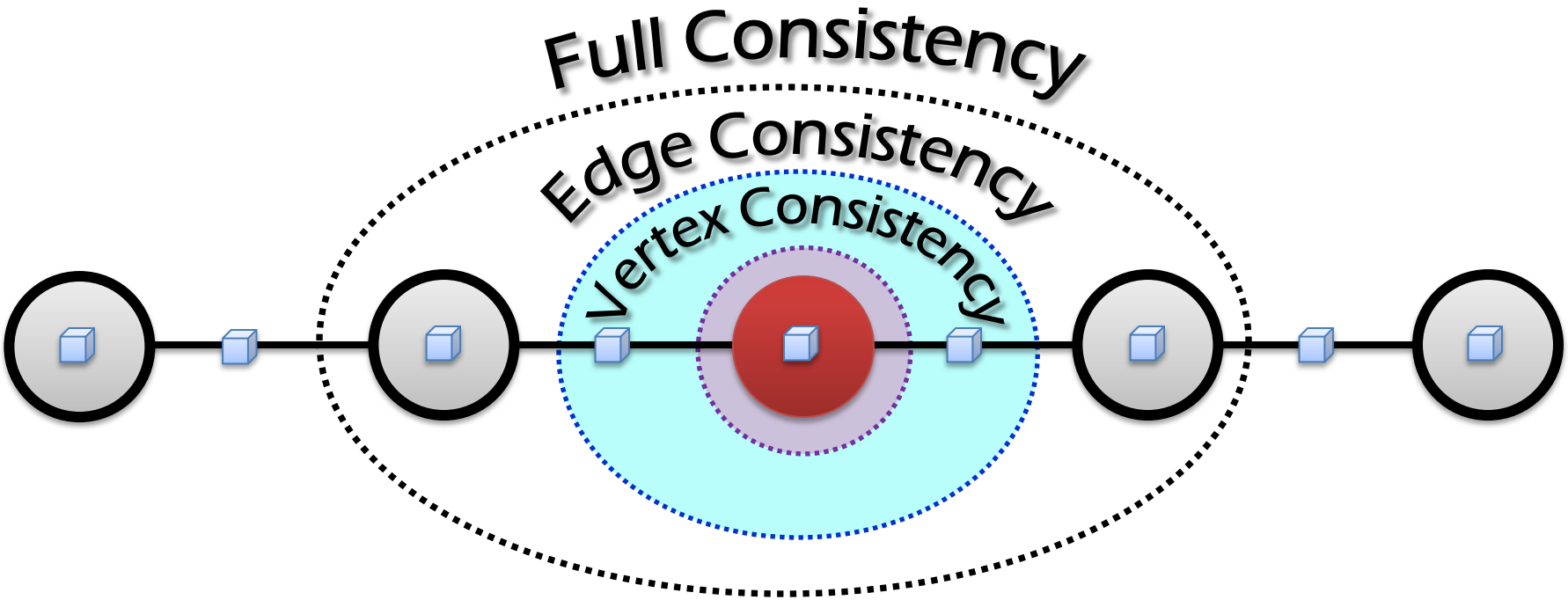
Full Consistency



Edge Consistency



More Parallelism



Consistency through Scheduling

Edge Consistency Model

- Two vertices can be updated **simultaneously** → they do not share an edge

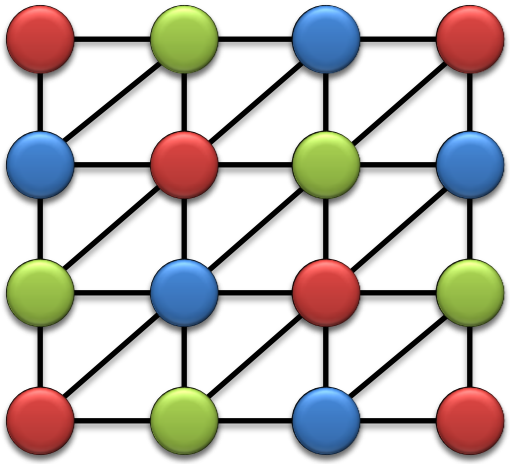
Consistency through Scheduling

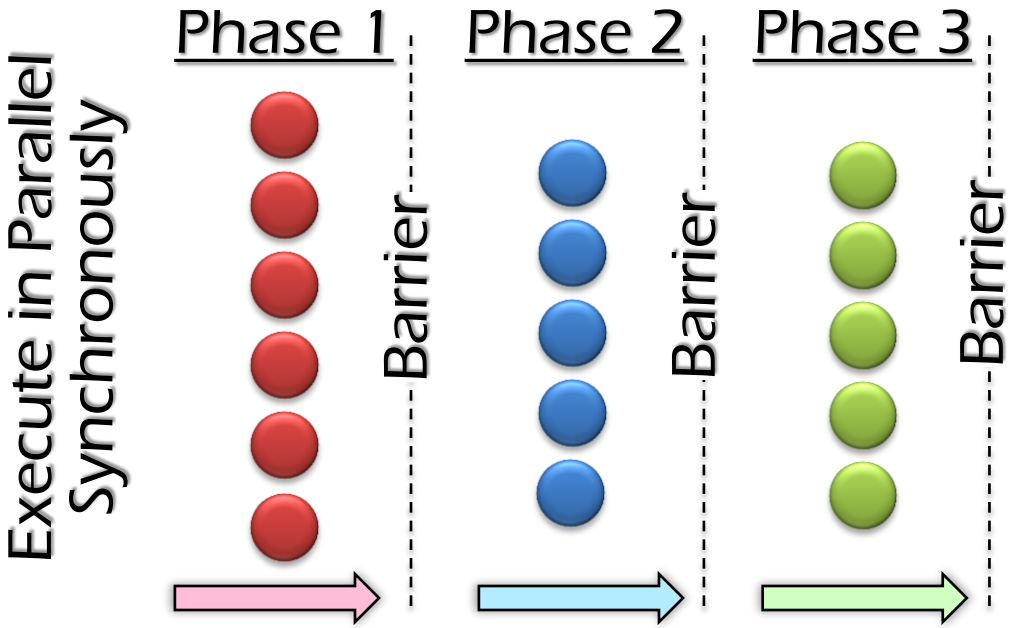
Edge Consistency Model

- Two vertices can be updated ~~simultaneously~~
  → they do not share an edge

same color

Graph Coloring



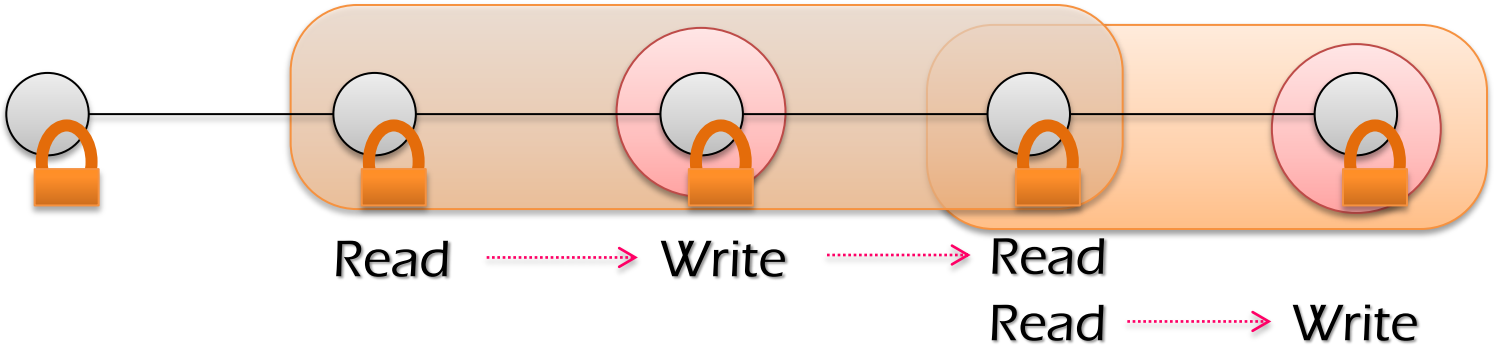


Consistency through R/W Locks

Multicore Setting: R/W Locks (Pthread)

- **Full** consistency

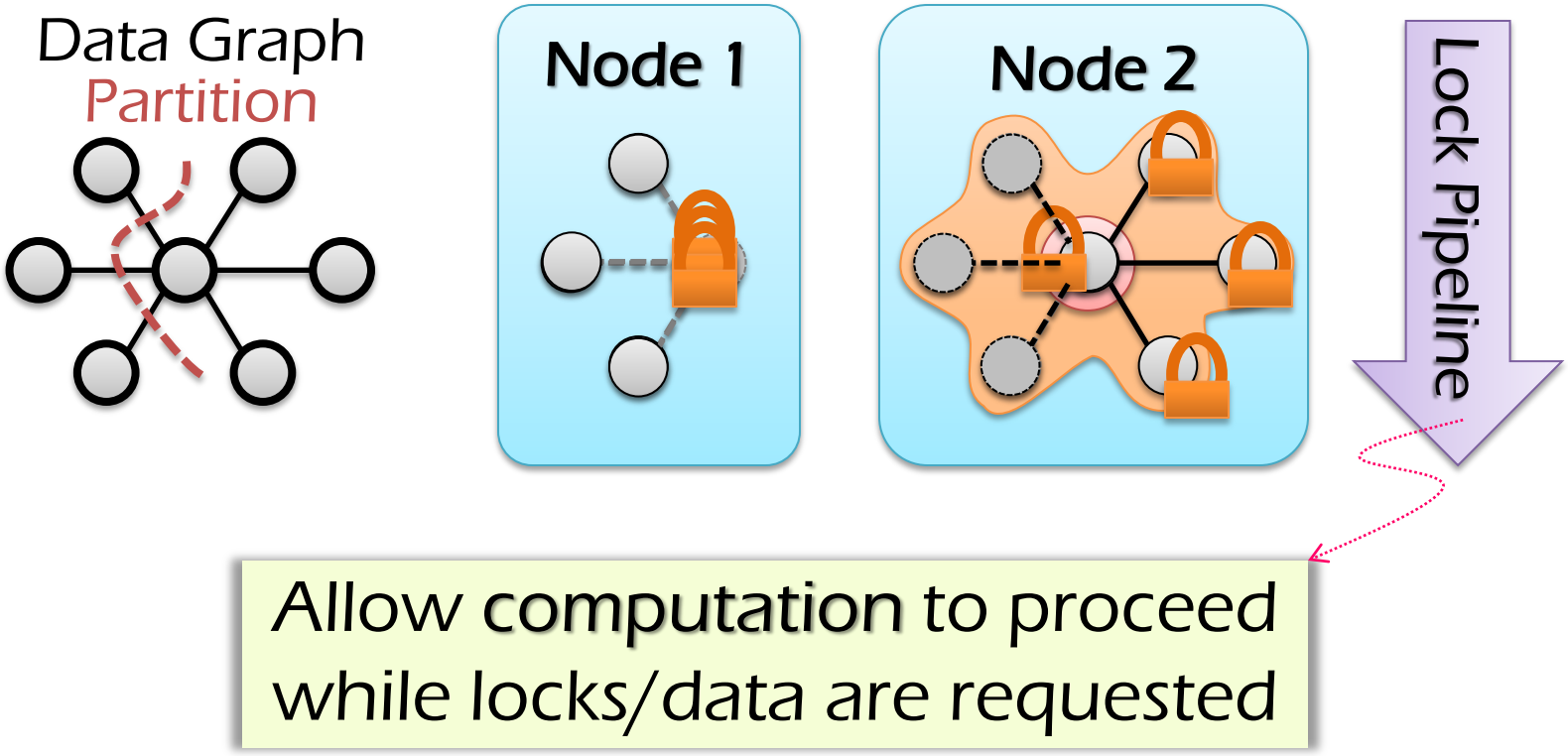
- **Edge** consistency



Consistency through R/W Locks

Distributed Setting: Distributed Locking

- Prefetch Locks and Data



Conclusion

An **abstraction** tailored to **Machine Learning**

A **distributed** and **parallel** framework which compactly expresses

- Data/computational **dependencies**
- **Iterative** computation

http://www.graphlab.org/

Presto: Distributed Machine Learning and Graph Processing with Sparse Matrices

Shivaram Venkataraman[*], Erik Bodzsar[#], Indrajit Roy[+],
Alvin AuYoung[+], Rob Schreiber[+]

[*]UC Berkeley, [+]HP Labs, [#]U Chicago

PRESTO

http://www.hpl.hp.com/research/presto.htm

Background

It is **cumbersome** to write machine learning and graph algorithms in **data-parallel** models (*e.g., MapReduce, Dryad*)

- *Top-K eigenvalues (Twitter), Dominant eigenvector (Google), Vertex Centrality (Facebook), Matrix factorization (Netflix)*

Array-based languages (*e.g., R*) are suitable for implementing complex algorithms

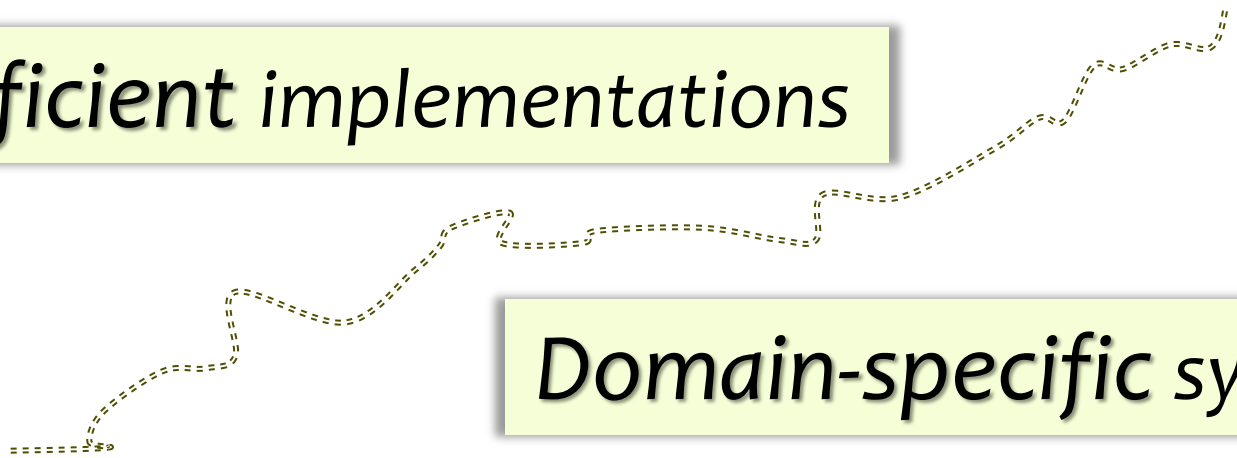
> *Randomness combined with **linear algebra** is a powerful tool for modern problems.*
>
> *-- Ravi Kannan, FCRC 2011*

Why not data-parallel systems?

The **simplicity** of the programming model (*e.g.*, *MapReduce*) or reliance on **relational** algebra (*e.g.*, *DryadLINQ*)

- Don't support **stateful** computations
- Don't retain the structure of **global shared** data
- Don't allow **point to point** communication

Inefficient implementations

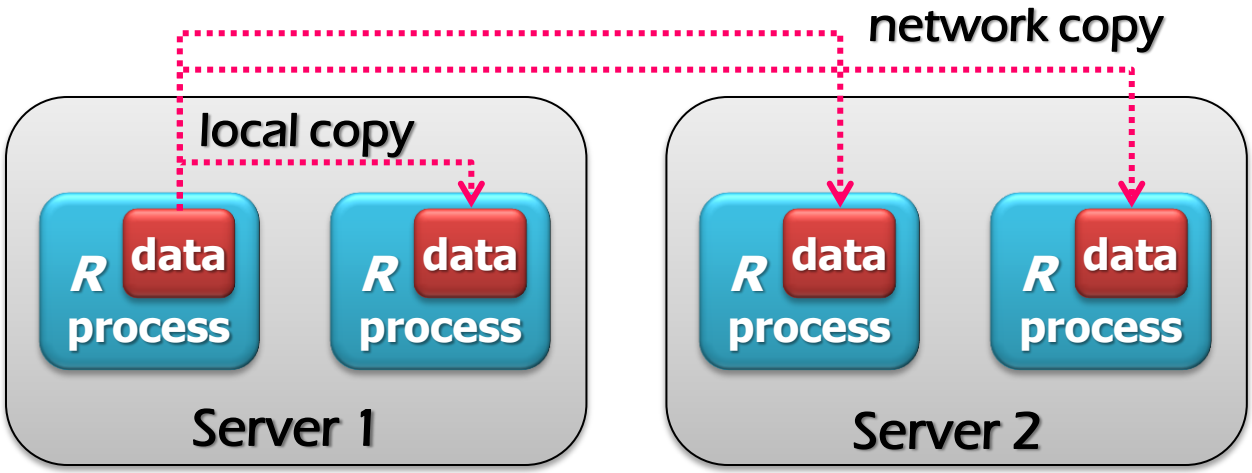


Domain-specific systems

Challenges

Effective use of *multi-cores*

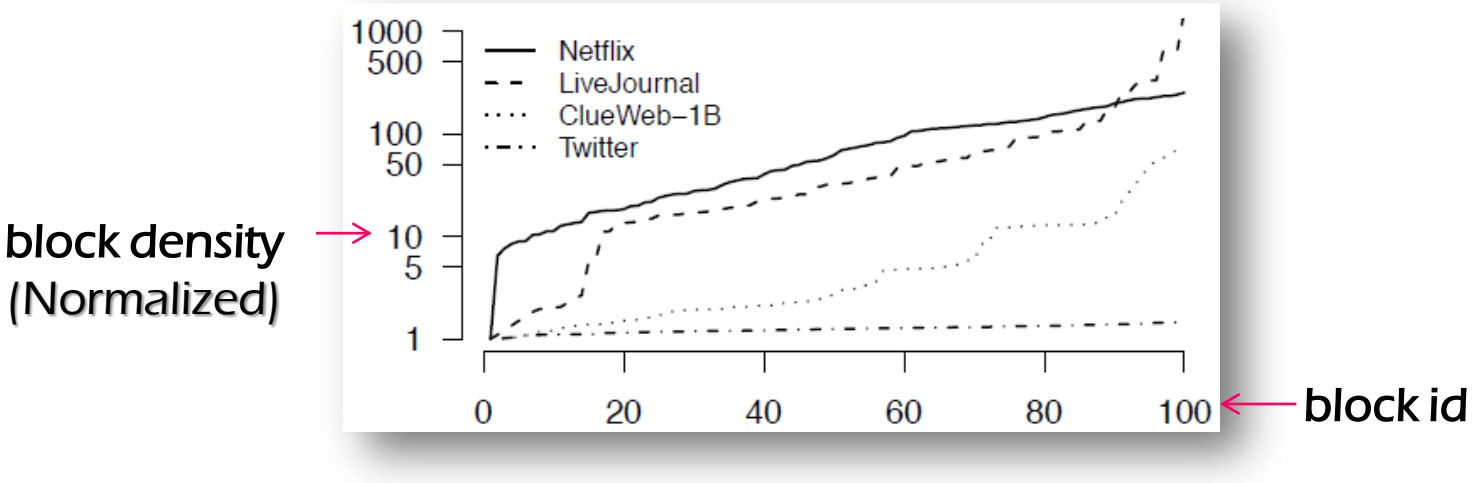
- *R* is single threaded
- "Multi-process approach" has two limitations
  - Redundant data copies
  - Network communication overhead
- Support *point-to-point* communication



Challenges

Imbalance in **sparse** computation

- Most *real-world* datasets are *sparse*
  - Netflix prize dataset:
    480K users (rows) and 17K movies (cols)
    but only *100* million (possible *8* billion ratings)
- Skew due to the *power-law* distribution
- *Computation* & *comm*. imbalance (*partition*)



Introduction to R

R provides an interactive environment to analyze data

- Conditional execution (***if***), loops (***for***, ***while***, ***repeat***), and uses ***array operators*** written in *C/C++* or *FORTRAN*

#3x3 matrix

```
> A←array(10:18,dim=c(3,3))
> A
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
```

#First row

```
> A[1,]
[1] 10 13 16
```

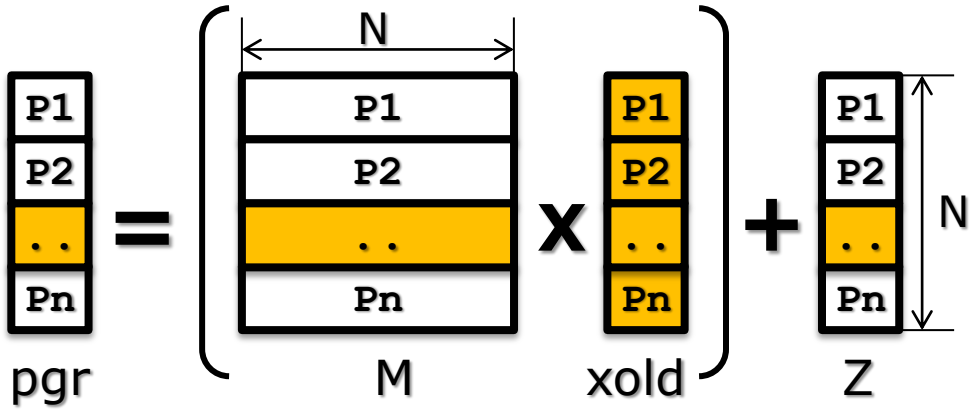
```
> idx←array(1:3,dim=c(3,2))
> idx
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
```

#Index vector

```
> A%*%idx
     [,1] [,2]
[1,]   84   84
[2,]   90   90
[3,]   96   96
```

#Matrix multiply

Example: PageRank



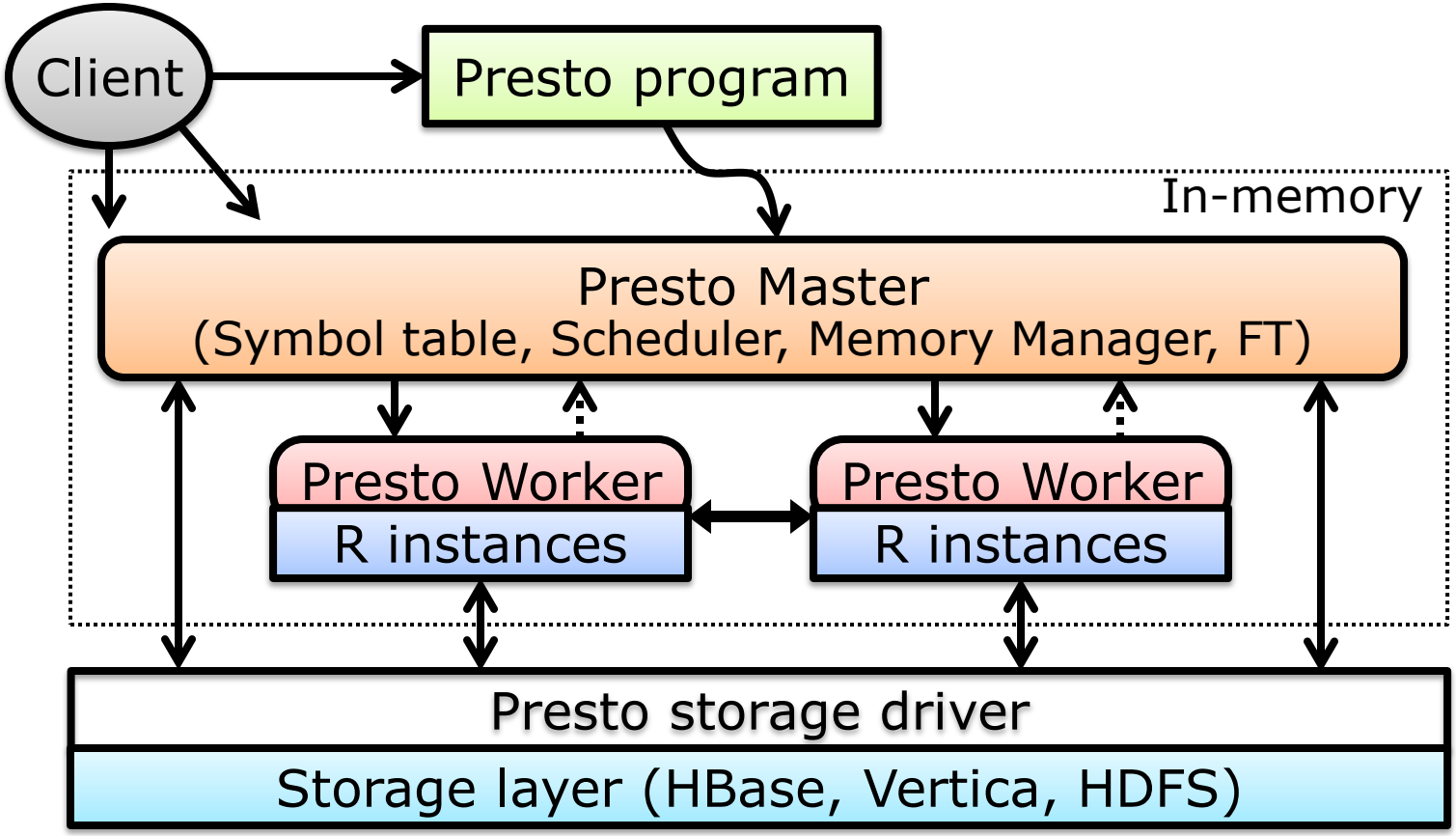
$$R[i] = \alpha + (1-\alpha) \sum_{(j,i)\in E} \frac{1}{L[j]} R[j]$$

M = *web graph matrix*
pgr = *PageRank vector*

```
#Load data from adjacency matrix in HBase
: M←array(dim=c(N,N),sparse=T)
: load(M,table='web-graph',drv='HBase')
: pgr←array(dim=c(ncol(M),1),sparse=F)
: ...
#Calculate PageRank(pgr)
: repeat{
:     pgr←(M%*%xold)+Z
:     if (norm(pgr-xold)<1e-9) break
:     xold←pgr
: }
```

Programming Model

Presto = *R* + (***darry, foreach, spilts, update***)



Programming Model: darray

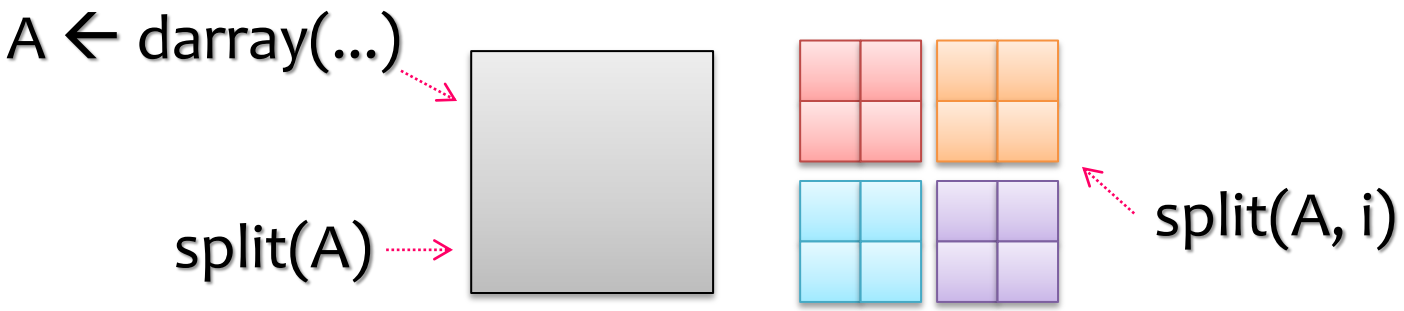
Distributed array (**darray**) provides a shared, in-memory view of multi-dimensional data stored across multiple servers

- Partition: *rows*, *columns* or *blocks*

splits function (*e.g.*, **splits(A)**, **split(A, i)**)

- Automatically fetch remote partitions and combine partitions to form a local array

---



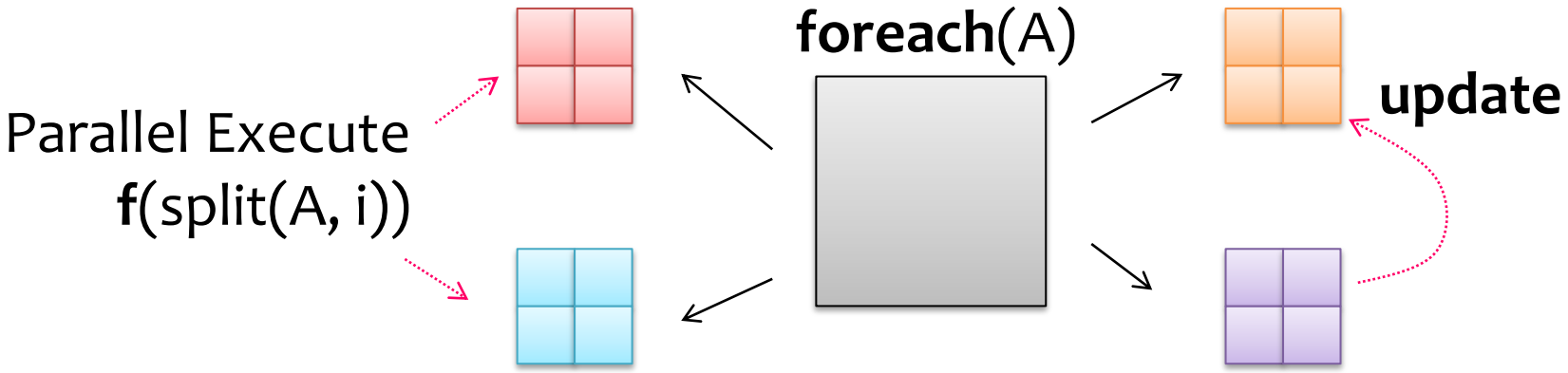
Programming Model: foreach

Runtime assigns tasks to workers for parallel execution of the loop body

- A barrier at the end of the loop

foreach function (*e.g.*, **foreach(v, A, f())**)

- Execute deterministic function **f** in parallel
- Call **update** inside function to publish changes

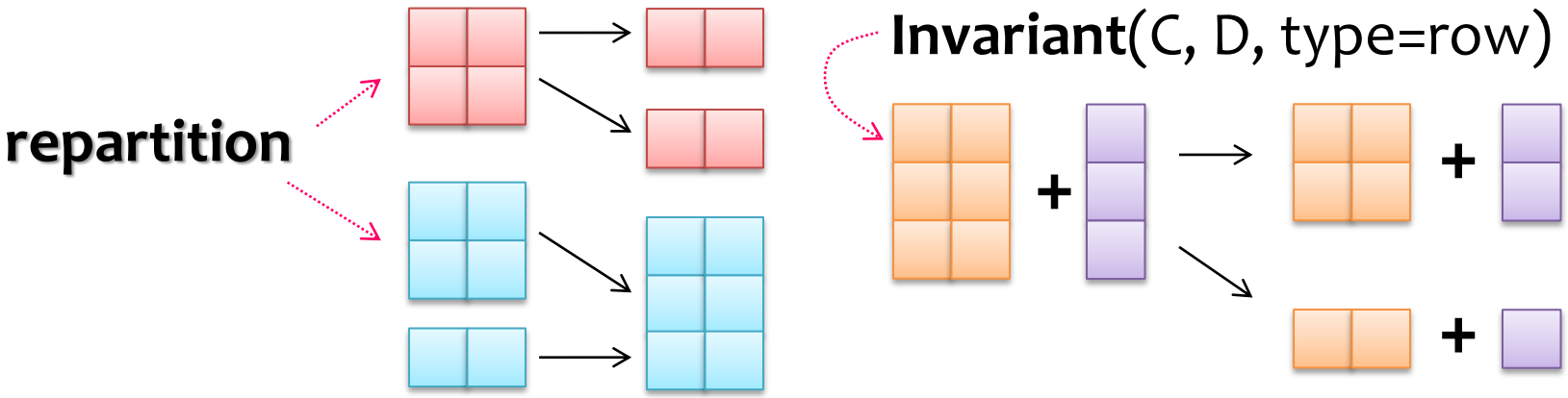


Programming Model: repartition

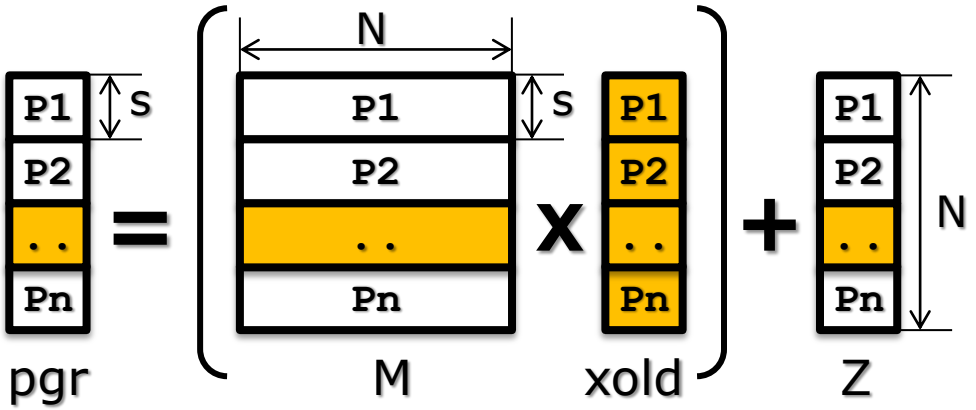
Dynamically repartition **darray** to reduce load imbalance and mitigate stragglers

repartition function (*e.g.*, **repartition(A, …)**)

- Subdivide and combine partitions
- Use **invariant** to automatically detect and reduce imbalance



Example: PageRank



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

M = *web graph matrix*
pgr = *PageRank vector*

```
#Load data from adjacency matrix in HBase
: M←darray(dim=c(N,N),blocks=c(s,N),sparse=T)
: load(M, table='web-graph', drv='HBase')
: pgr←darray(dim=c(N,1),blocks=c(s,N)sparse=F)
: ...
: invariant(pgr,M,xold,Z,type=ROW)
#Calculate PageRank(pgr)
: repeat{
:    foreach(i, i:numspilts(M),
:          prFunc(p=splits(pgr,i),m=splits(M,i)),
:                 x=splits(xold,i),z=splits(Z,i)){
:       p←(m%*%x)+z
:       update(p)
:    })
:    if (norm(pgr-xold)<1e-9) break
:    xold←pgr
: }
```

Partition darrays by row

Fix # of rows in repartition

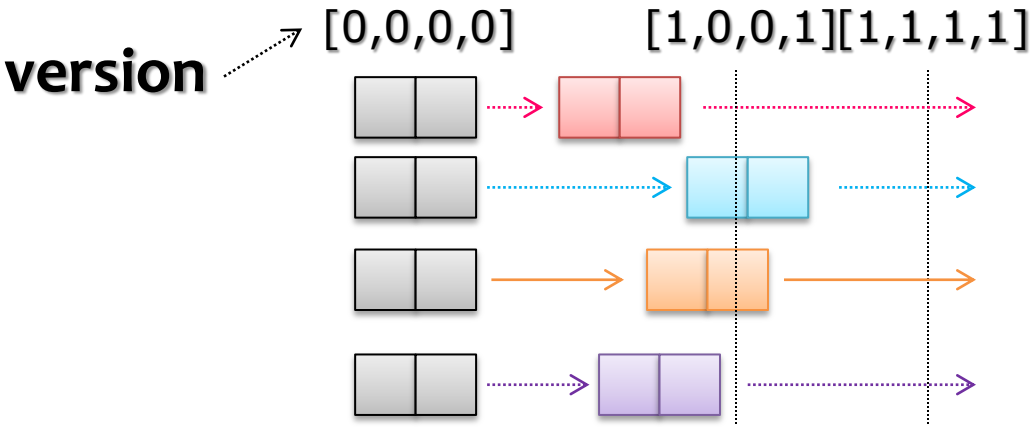
Parallelize computation

Publish the change before next iteration

Versioning Arrays

The **version** of **darray** is a concatenation of the versions of its partitions

- In spirit to vector clock
- Avoid read-write conflicts
- Garbage collection
  - Workers periodically inform the master about what partitions are still using



Efficient Multi-core Support

R is single threaded and not thread safe

Three drawbacks of *multi-worker* processes

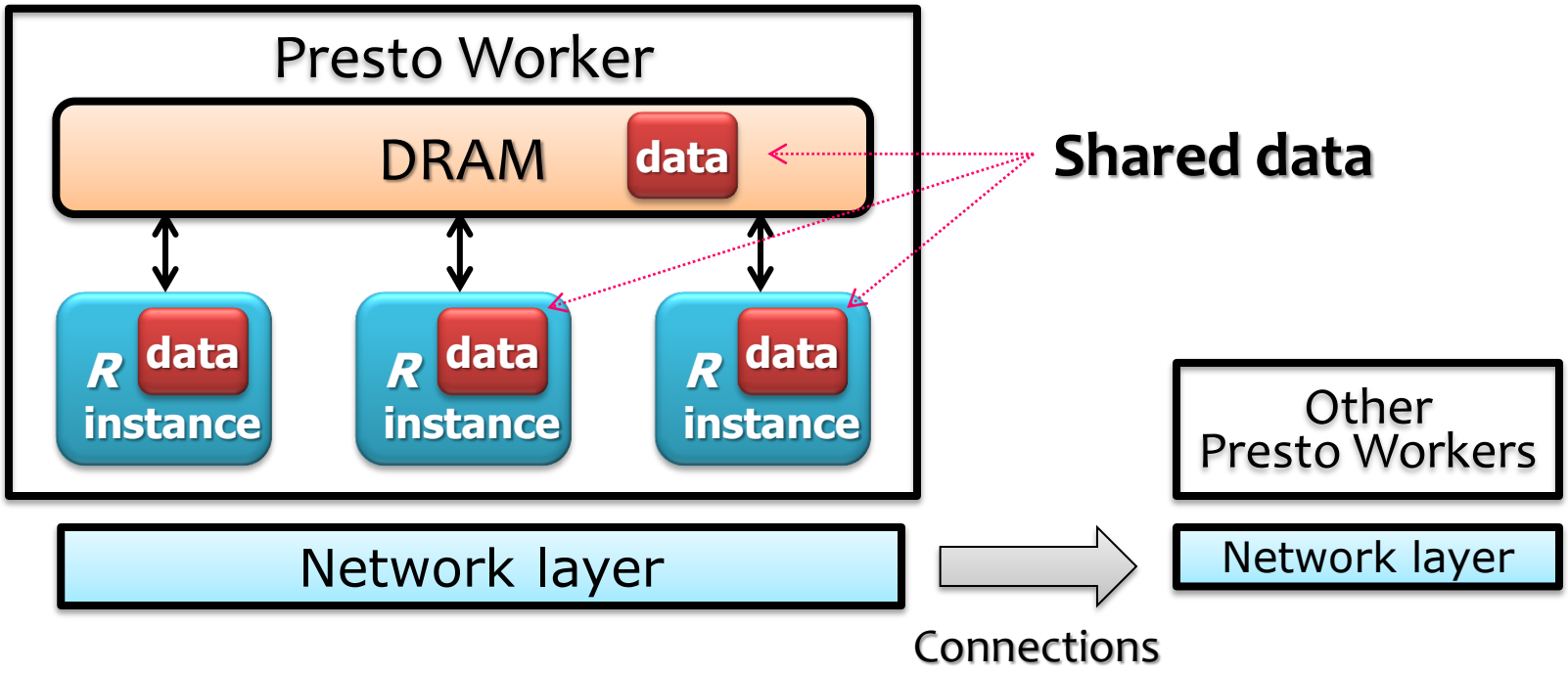
- Multiple copies of the same array (~~scalability~~)
- Copy data across processes (pipe, network)
- The overhead of communication is scale with #cores instead of #nodes

Encapsulate multiple R processes that can communicate through shared memory with zero copying

Efficient Multi-core Support

Efficiently initialize *R* objects by mapping data using **mmap** or shared memory constructs

---

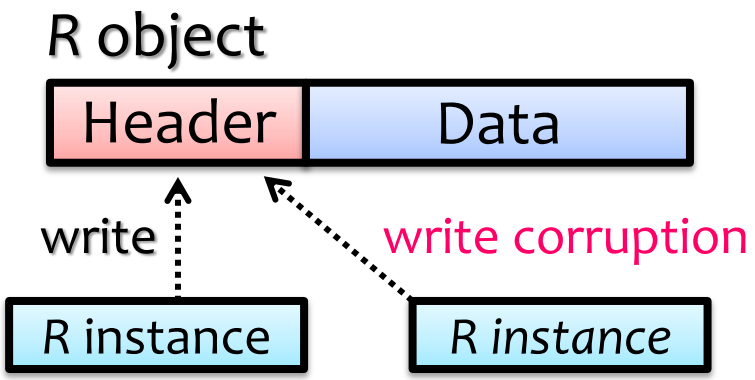


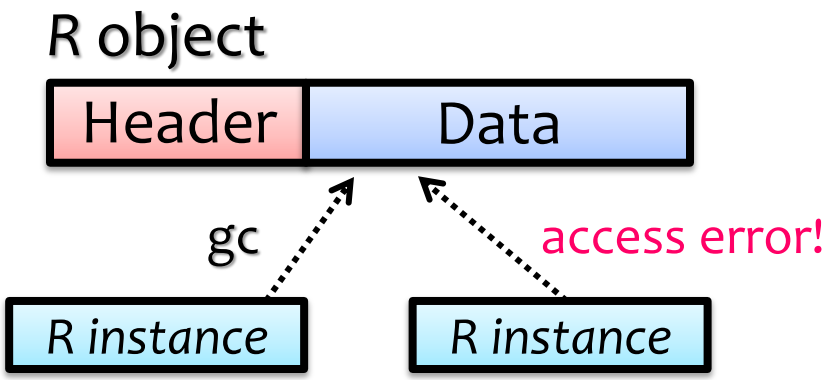
Issues with Data Sharing

R object: fixed-size header + an array of data

Simply sharing *R* objects across different *R* instances

- Header corruption
- Dangling pointer

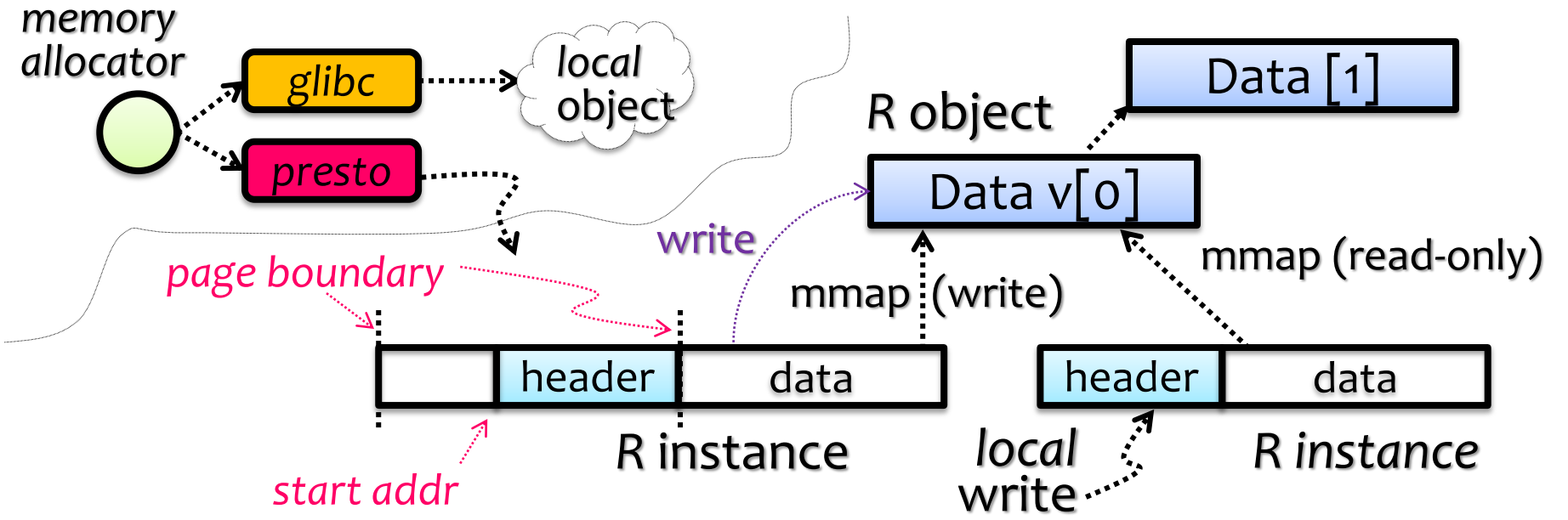




Safe Data Sharing

Sharing read-only data among *R* processes

- Only one process may write to a partition during a loop iteration
- Write always create a new version of a partition
- Page alignment (***mmap***)



Dynamic Partitioning

Partitioning a sparse matrix

- Lead to uneven distribution
- Cause a skew in execution time

Solution: dynamic partitioning

- Mitigate load imbalance
- Increase and decrease parallelism

Two observations

- Target algorithms are iterative
- The invariant provides correctness of repartition

Dynamic Partitioning

Presto runtime tracks

- # of elements in a partition (e_i) and median (e_m)
- Execution time of the tasks (t_i) and median (t_m)

The aim of dynamic partitioning

- Keep the partition size and the execution time of each task close to the median

After each iteration, runtime checks

- A partition has more(less) elements than the median by a given constant ($e_i/e_m \geq \delta$)

  → Subdivide(merge) partitions

Communication

Presto master uses three mechanisms to reduce communication

- Locality based scheduling
  - Use symbol table to minimize data movement
- Partition co-location
  - Reuse invariant information
- Automatically caching
  - Reuse versions info to maintain coherence

Fault Tolerance

Master failure: primary-backup replication

- Meta-data (*symbol table, execution status, worker information*) is replicated

Worker failure: (hypothesize *deterministic*)

- Detect: heartbeat message
- Recovery: re-execution
- *Spark*-like mechanism
  - Just store coarse-grained transformations
  - Recursively recreate corresponding version info (input version → output version)

Evaluation

Application	Algorithm Characteristic	R	Presto
PageRank	Eigenvector Calculation	20	41
Vertex Centrality	Graph Algorithm	40	128
Edge Centrality	Graph Algorithm	48	132
SSSP	Graph Algorithm	30	62
Netflix recommender	Matrix Decomposition	78	130
Triangle Count	Top-K eigenvalues	65	121
k-Means Clustering	Dense Linear Algebra	35	71

Platform

Opponents: Spark and Hadoop-mem-0.20 (Mahout)

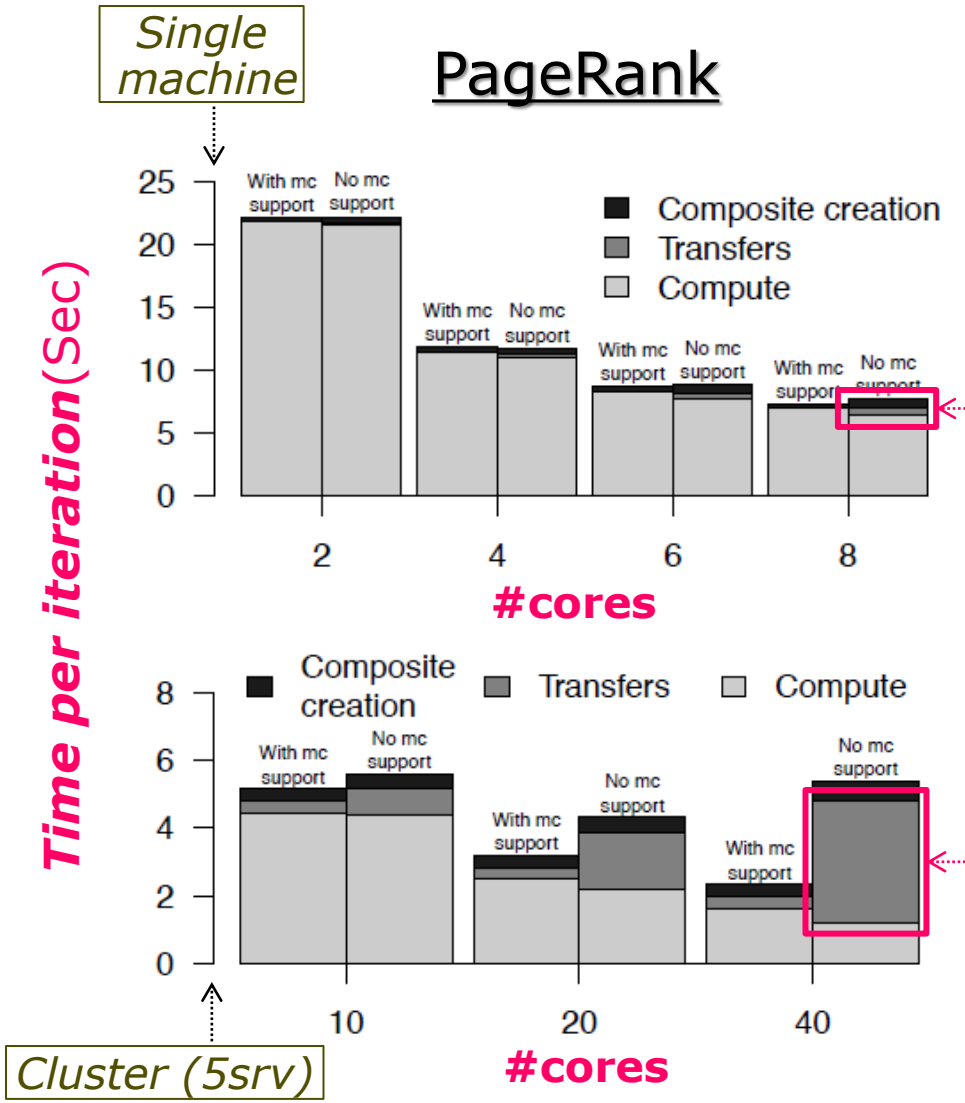
Hardware: 50 HP SL390
(2*12 Intel Xeon, 96G RAM, 10Gbps NIC)

Evaluation

Our evaluation shows that

- **Presto** is the first R extension to efficiently leverage multicores by reducing memory and network overheads
- **Presto** can handle load imbalance due to sparsity by dynamic partitioning
- **Presto** is much faster than current systems (*PageRank*: 40x Hadoop, 15x Spark)

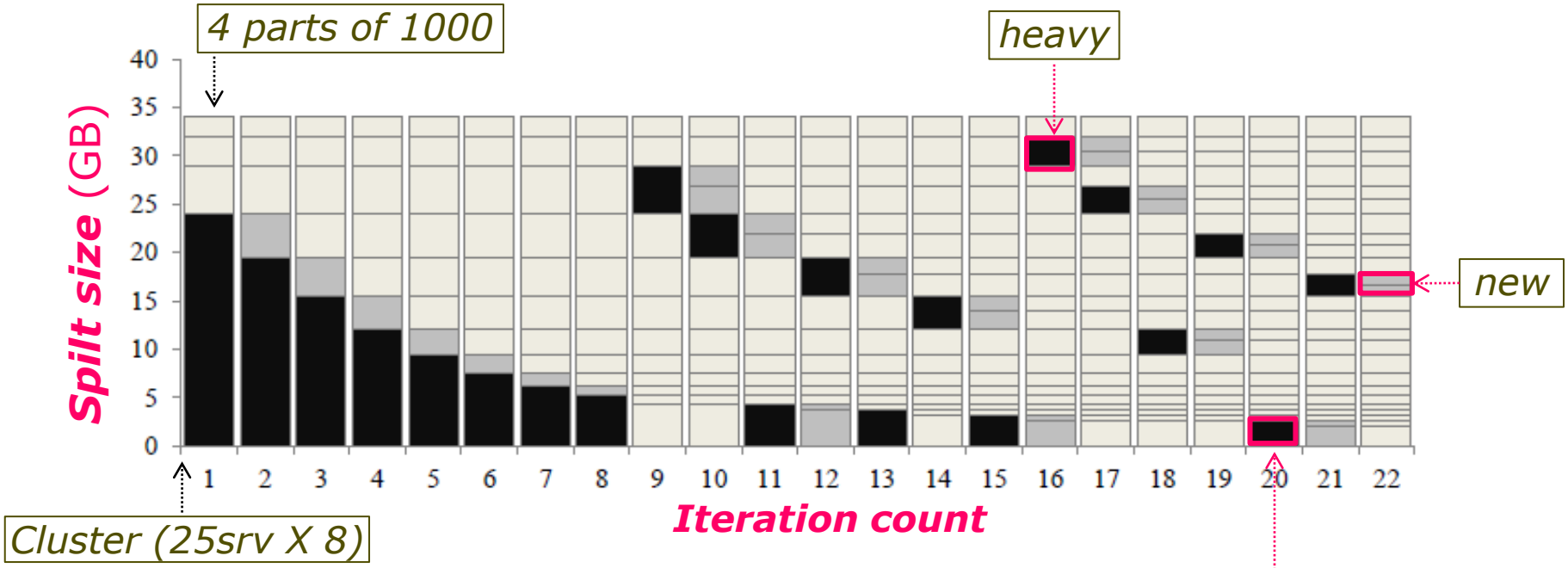
Multicore Support



NoMC spends 7% of time in data transfers and takes 5% longer to complete

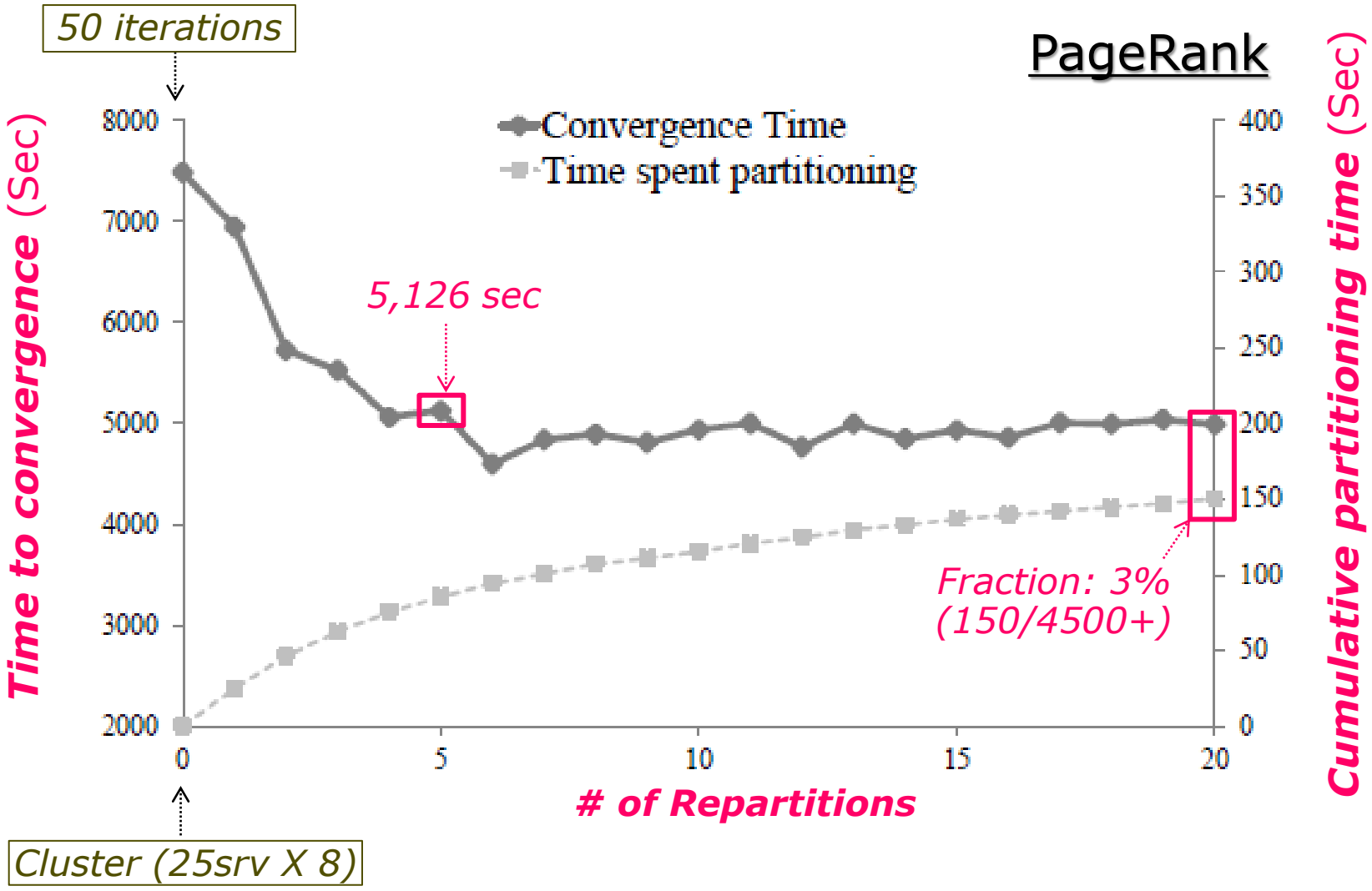
NoMC spends 9.7x net transfer overhead and does not scale

Dynamic Partitioning



The **first** block is continuously repartitioned at iteration **1-8**, **11**, **13**,**15**, and **20**. repartitioning reduce the size from **23GB** to **2.2GB**

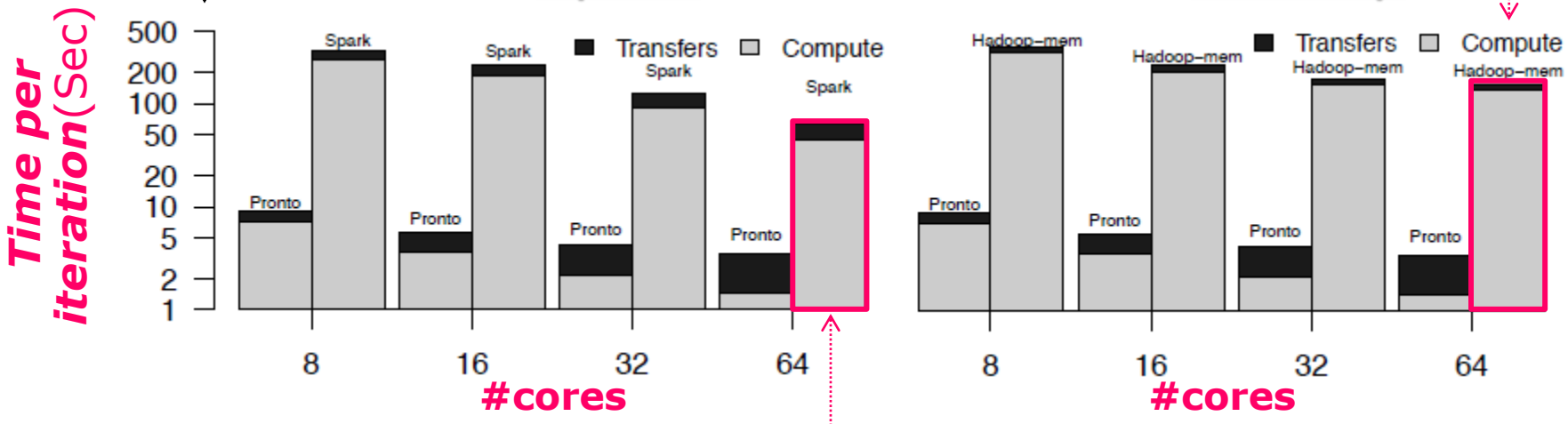
Dynamic Partitioning



Comparison with Opponents

*Hadoop takes **161 sec** per-iteration*

Cluster (8srv X 8)



*Spark takes **64.2 sec** per-iteration*
*(**44.3** in Map, and **19.8** in Reduce)*

Summary

Presto advocates the use of *sparse* matrix operations to simplify the implementation of machine learning and graph algorithms in a cluster.

-- Shivaram Venkataraman

Presto uses distributed arrays for structured processing, efficiently uses multicore, and dynamically partitions data to reduce load imbalance

Presto is a flexible computation model and substantially faster than Hadoop/Spark

Next Lecture

Topic: Distributed File System

Paper

Mike Burrows, "***The Chubby Lock Service for Loosely-Coupled Distributed Systems***". Symposium on Operating Systems Design and Implementation (OSDI), **2006**

THANKS