Using R for Iterative and Incremental Processing

Shivaram Venkataraman, Indrajit Roy,
Alvin AuYoung, Robert Schreiber

UC Berkeley and HP Labs

amplab UC BERKELEY

Big Data, Complex Algorithms

PageRank
(Dominant eigenvector)

Recommendations
(Matrix factorization)

Anomaly detection
(Top-K eigenvalues)

User Importance
(Vertex Centrality)

Big Data, Complex Algorithms

PageRank
(Dominant eigenvector)

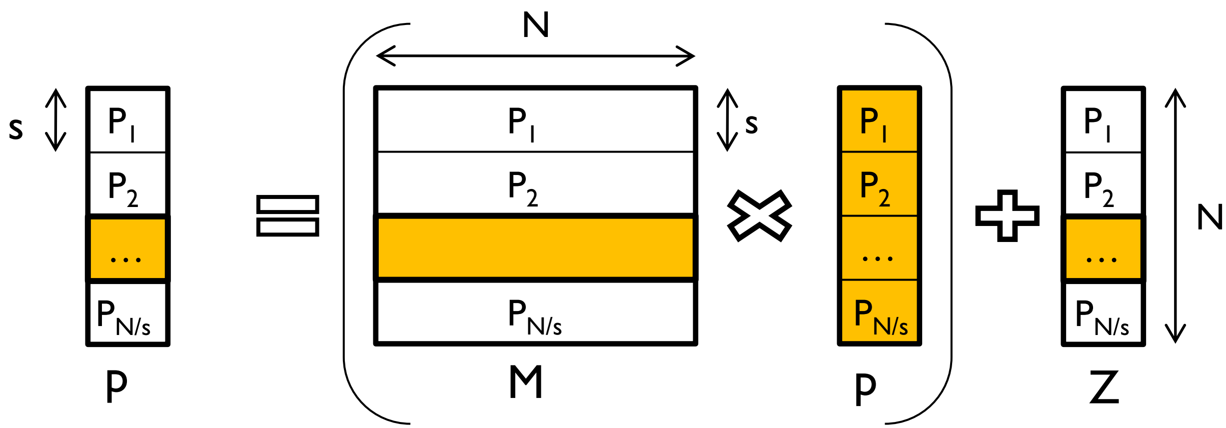
Machine learning + Graph algorithms

Iterative Linear Algebra Operations

Anomaly detection
(Top-K eigenvalues)

User Importance
(Vertex Centrality)

PageRank Using Matrices



Dominant eigenvector

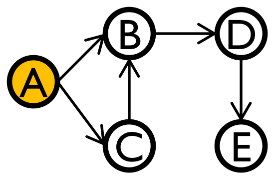
M = modified web graph matrix

p = PageRank vector

Simplified algorithm:

repeat { p = M*p + Z}

Breadth-first Search Using Matrices



X					
	1	0	0	0	0

	A	B	C	D	E
A	1	1	1	0	0
B	0	1	0	1	0
C	0	1	1	0	0
D	0	0	0	1	1
E	0	0	0	0	1

← G

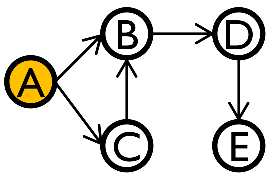
G = adjacency matrix

X = BFS vector

Simplified algorithm:

repeat { X = G*X }

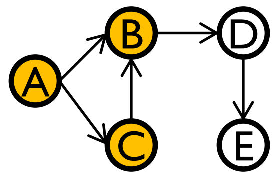
Breadth-first Search Using Matrices



X					
X	1	0	0	0	0

	A	B	C	D	E
A	1	1	1	0	0
B	0	1	0	1	0
C	0	1	1	0	0
D	0	0	0	1	1
E	0	0	0	0	1

← G



*	*	*	0	0
A	**B**	**C**	**D**	**E**

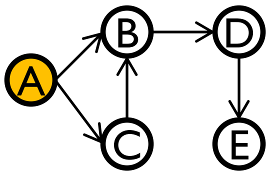
G = adjacency matrix

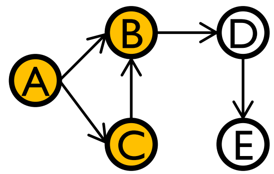
X = BFS vector

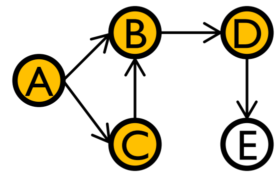
Simplified algorithm:

repeat { X = G*X }

Breadth-first Search Using Matrices







X	1	0	0	0	0

	A	B	C	D	E
A	1	1	1	0	0
B	0	1	0	1	0
C	0	1	1	0	0
D	0	0	0	1	1
E	0	0	0	0	1

← G

*	*	*	0	0
A	**B**	**C**	**D**	**E**

*	*	*	*	0
A	**B**	**C**	**D**	**E**

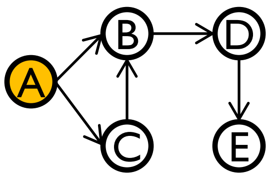
G = adjacency matrix

X = BFS vector

Simplified algorithm:

repeat { X = G*X }

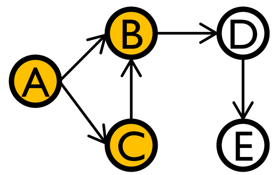
Breadth-first Search Using Matrices



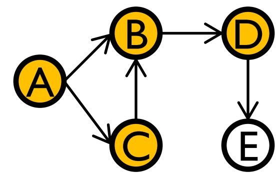
X	1	0	0	0	0

	A	B	C	D	E
A	1	1	1	0	0
B	0	1	0	1	0
C	0	1	1	0	0
D	0	0	0	1	1
E	0	0	0	0	1

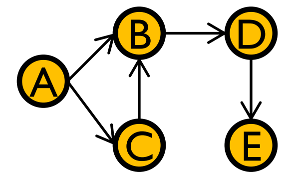
← G



*	*	*	0	0
A	**B**	**C**	**D**	**E**



*	*	*	*	0
A	**B**	**C**	**D**	**E**



*	*	*	*	*
A	**B**	**C**	**D**	**E**

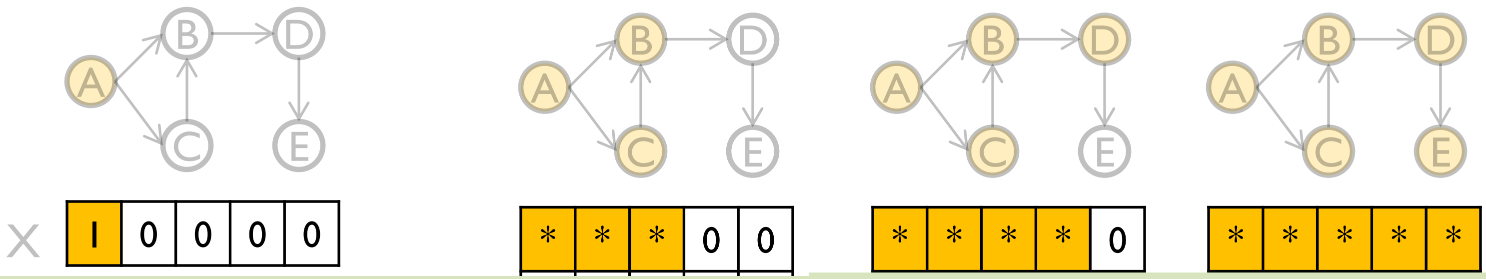
G = adjacency matrix

X = BFS vector

Simplified algorithm:

repeat { X = G*X }

Breadth-first Search Using Matrices



Matrix operations

Easy to express

Efficient to implement

E	0	0	0	0	1

G = adjacency matrix

X = BFS vector

Simplified algorithm:

repeat { X = G*X }

Linear Algebra on Existing Frameworks

Matrix Operations: *Structured*, coarse grained

Need global state

Linear Algebra on Existing Frameworks

Matrix Operations: *Structured*, coarse grained

Need global state

Data-parallel frameworks – MapReduce/Dryad

– Process each *record* in parallel

– Use case: Computing sufficient statistics

Linear Algebra on Existing Frameworks

Matrix Operations: *Structured*, coarse grained

Need global state

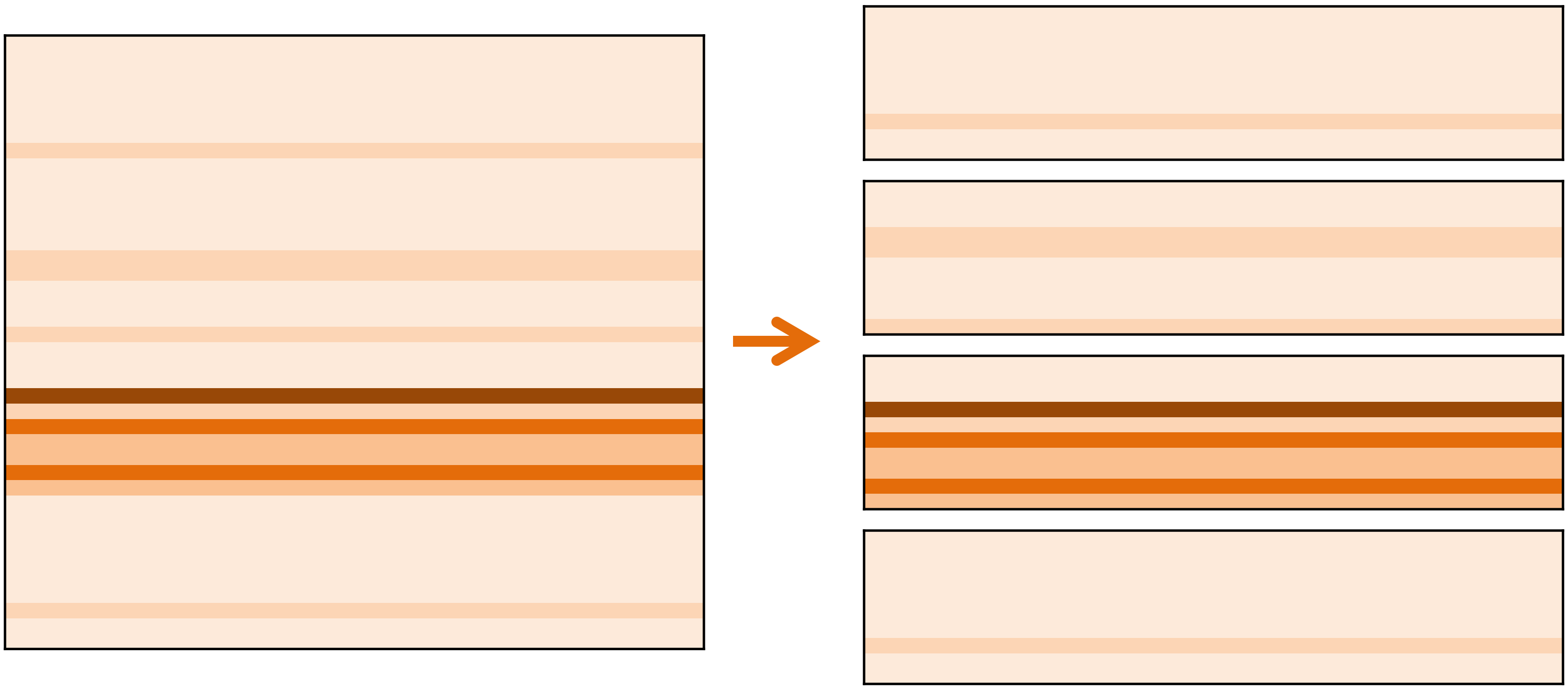
Data-parallel frameworks – MapReduce/Dryad

- – Process each *record* in parallel
- – Use case: Computing sufficient statistics

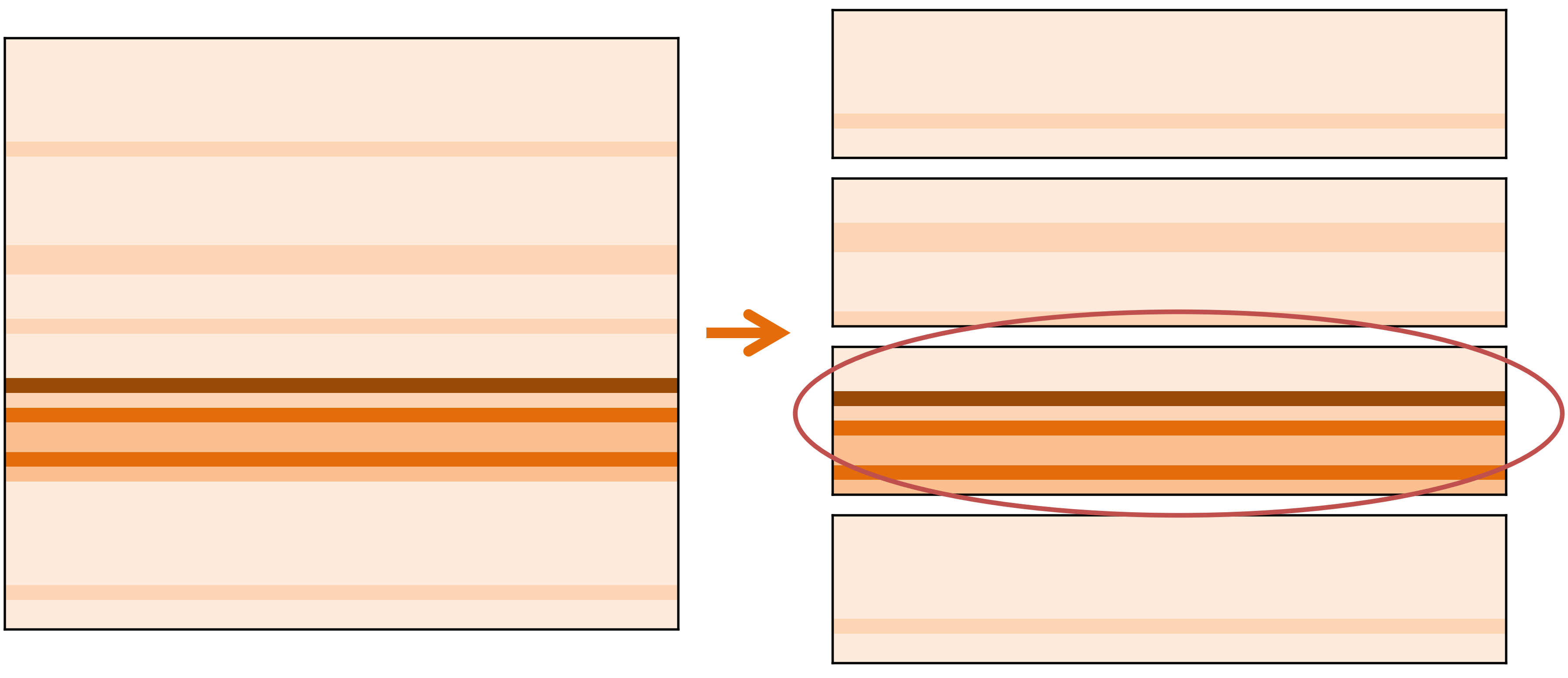
Graph-centric frameworks – Pregel/GraphLab

- – Process each *vertex* in parallel
- – Use case: Graph models

Challenge 1 – Sparse Matrices

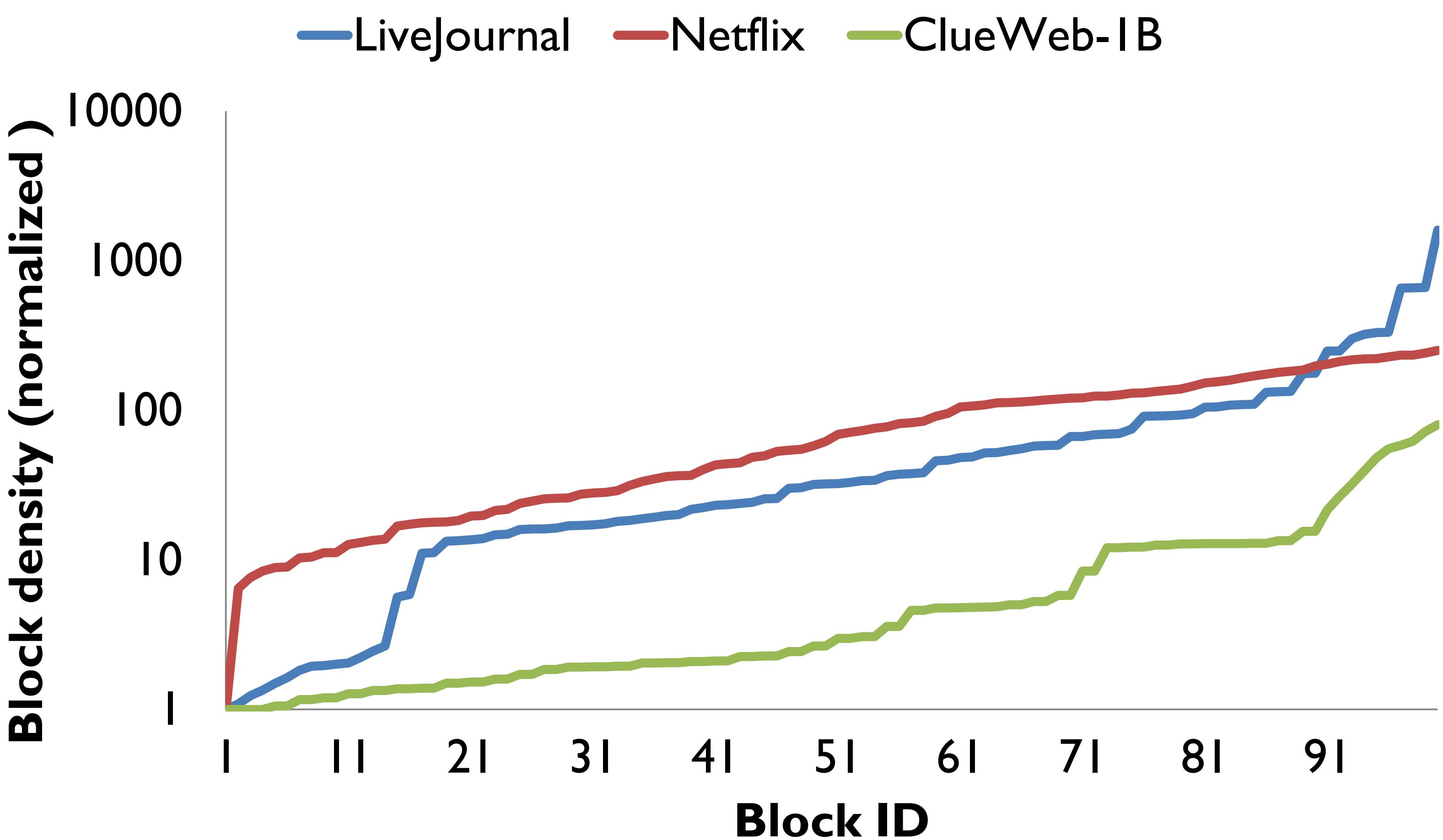


Challenge 1 – Sparse Matrices

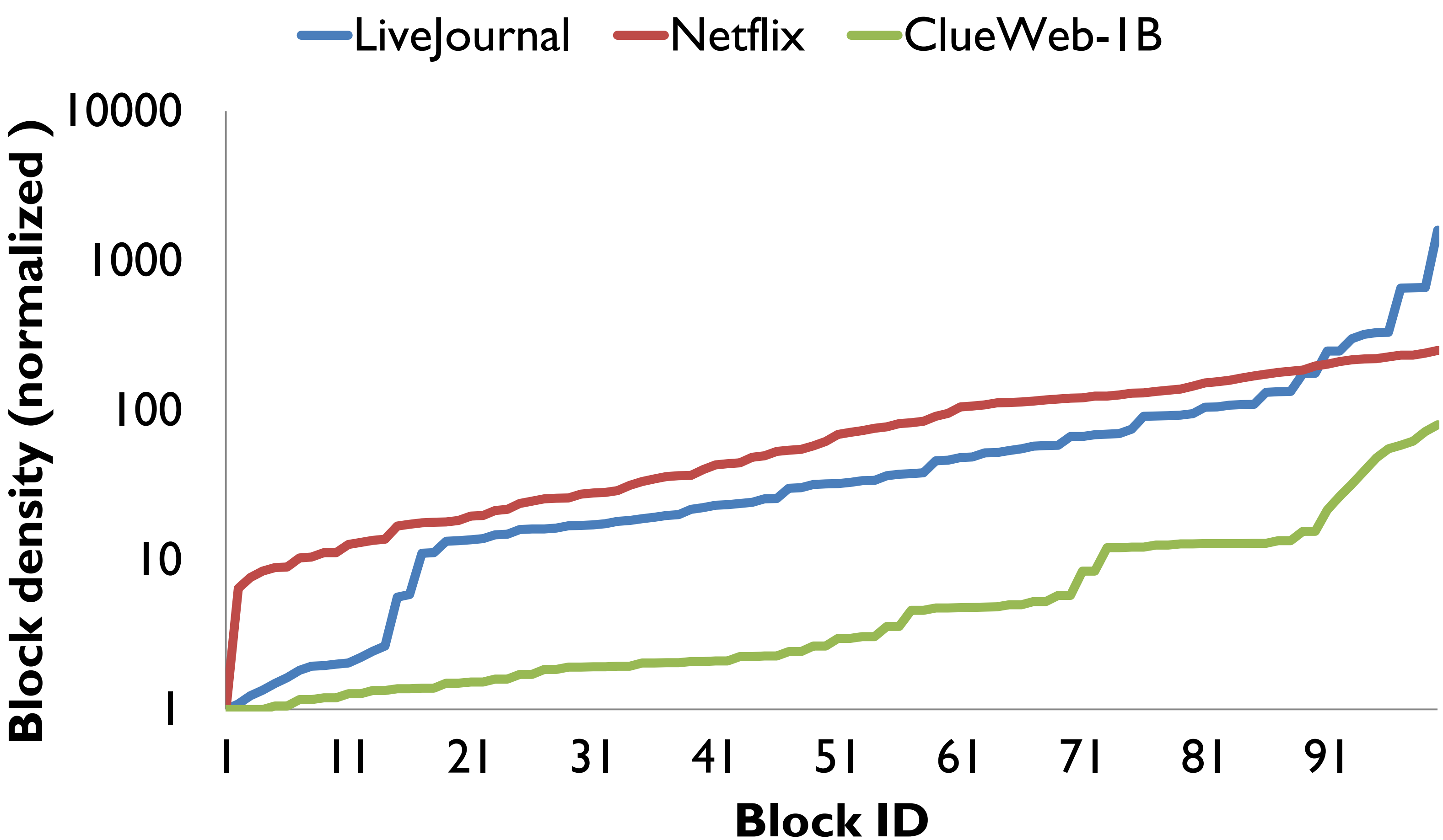


Challenge 1 – Sparse Matrices

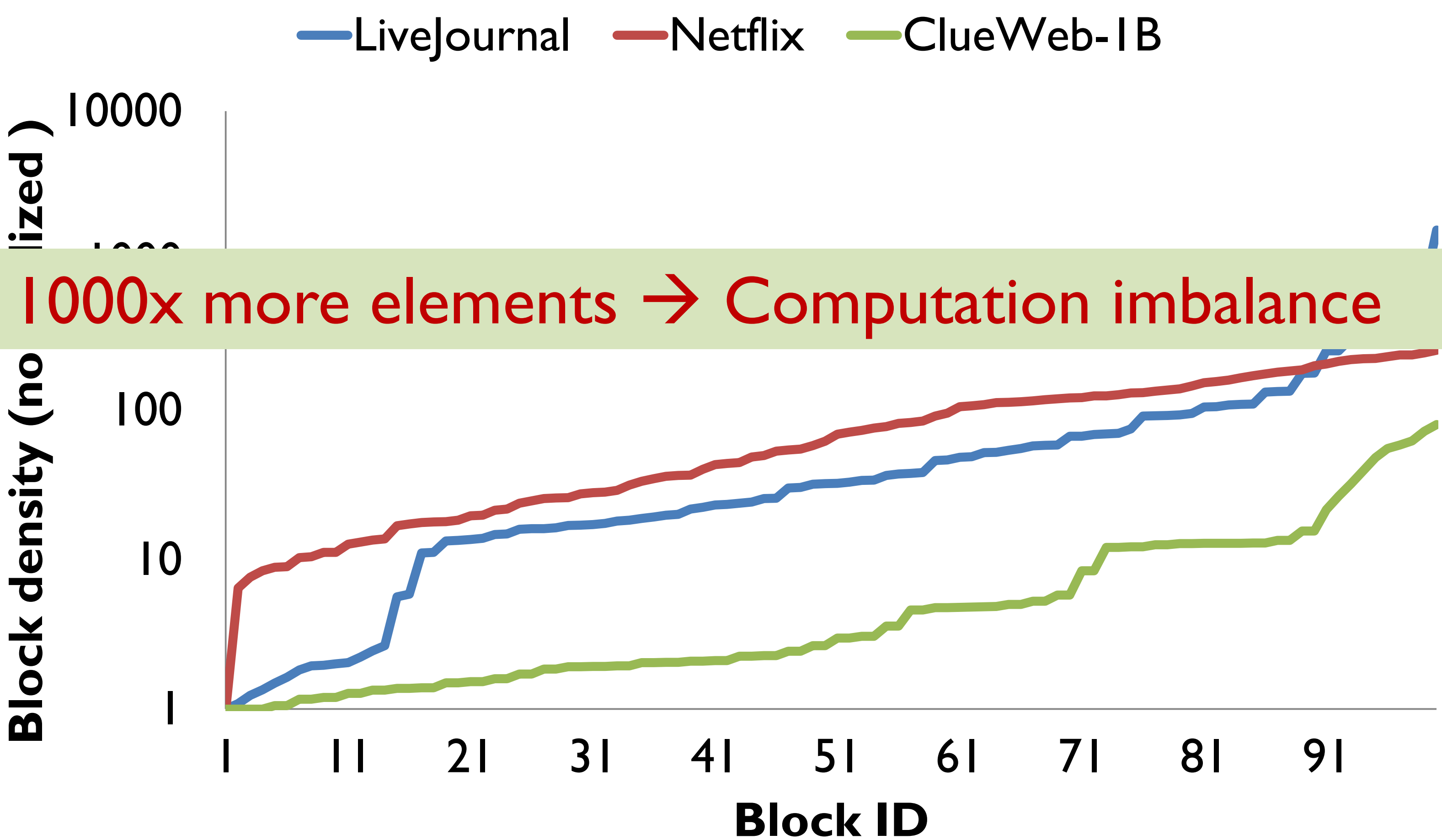
Challenge 1 – Sparse Matrices



Challenge 1 – Sparse Matrices

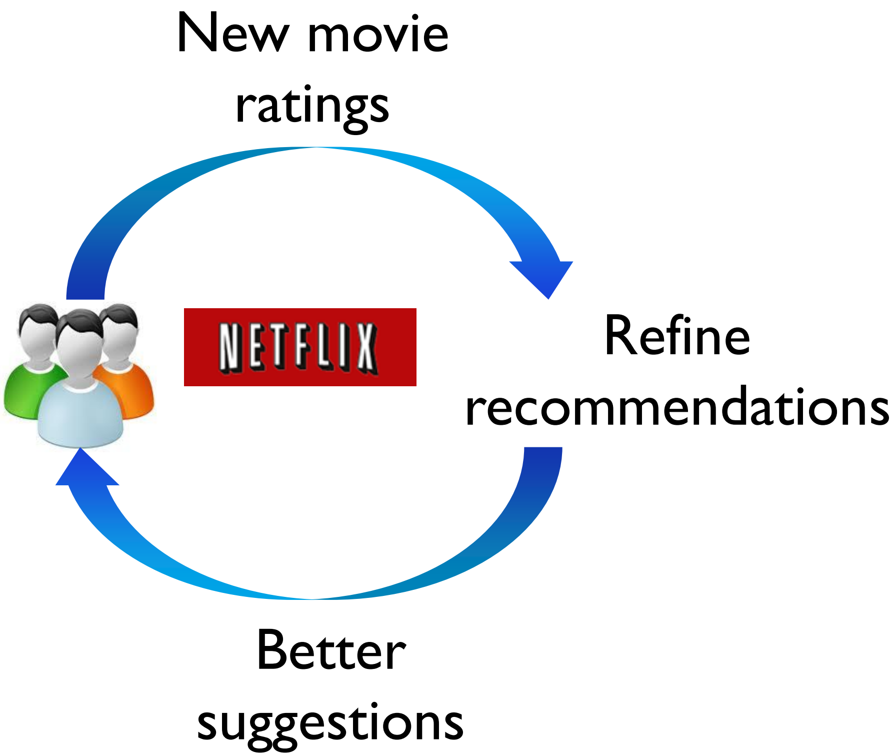


Challenge 1 – Sparse Matrices



1000x more elements → Computation imbalance

Challenge 2 – Incremental Updates



Incremental computation on consistent view of data

Presto

Framework for large-scale iterative linear algebra

Extend R for scalability and incremental updates

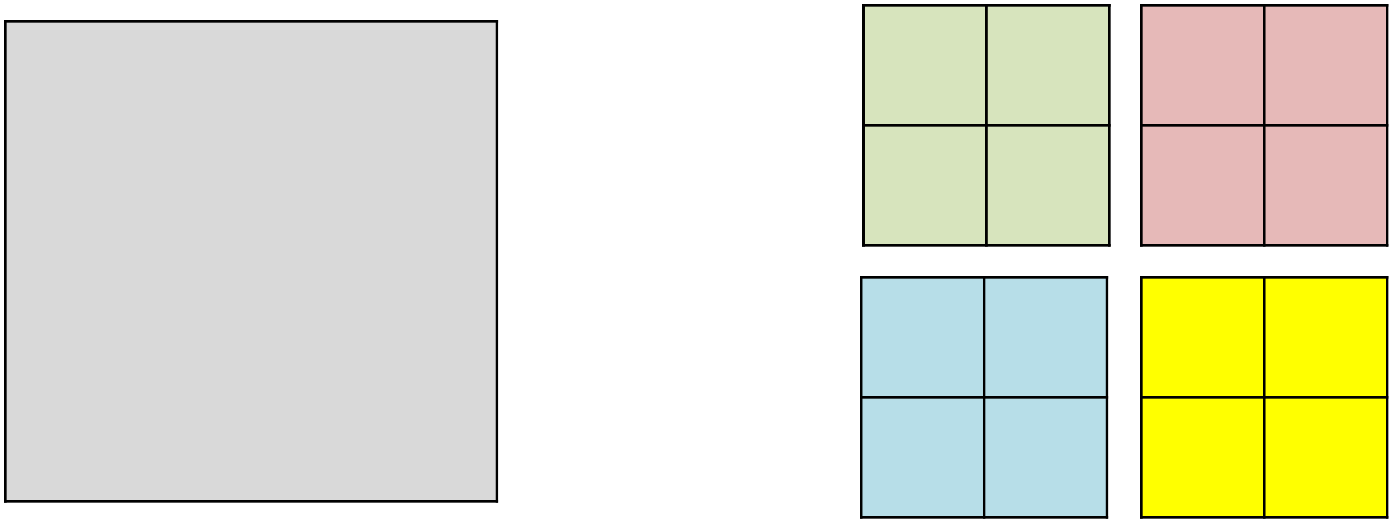
Outline

- Motivation
- Programming model
- Design
- Applications and Results

Programming Model

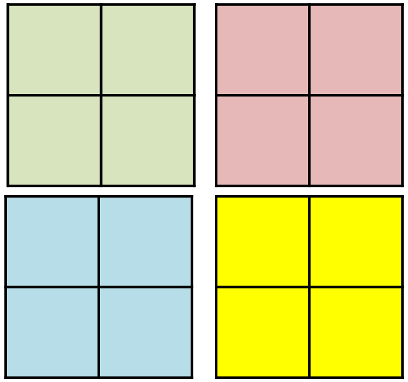
One data structure: Distributed Array

A ← *darray*(…)



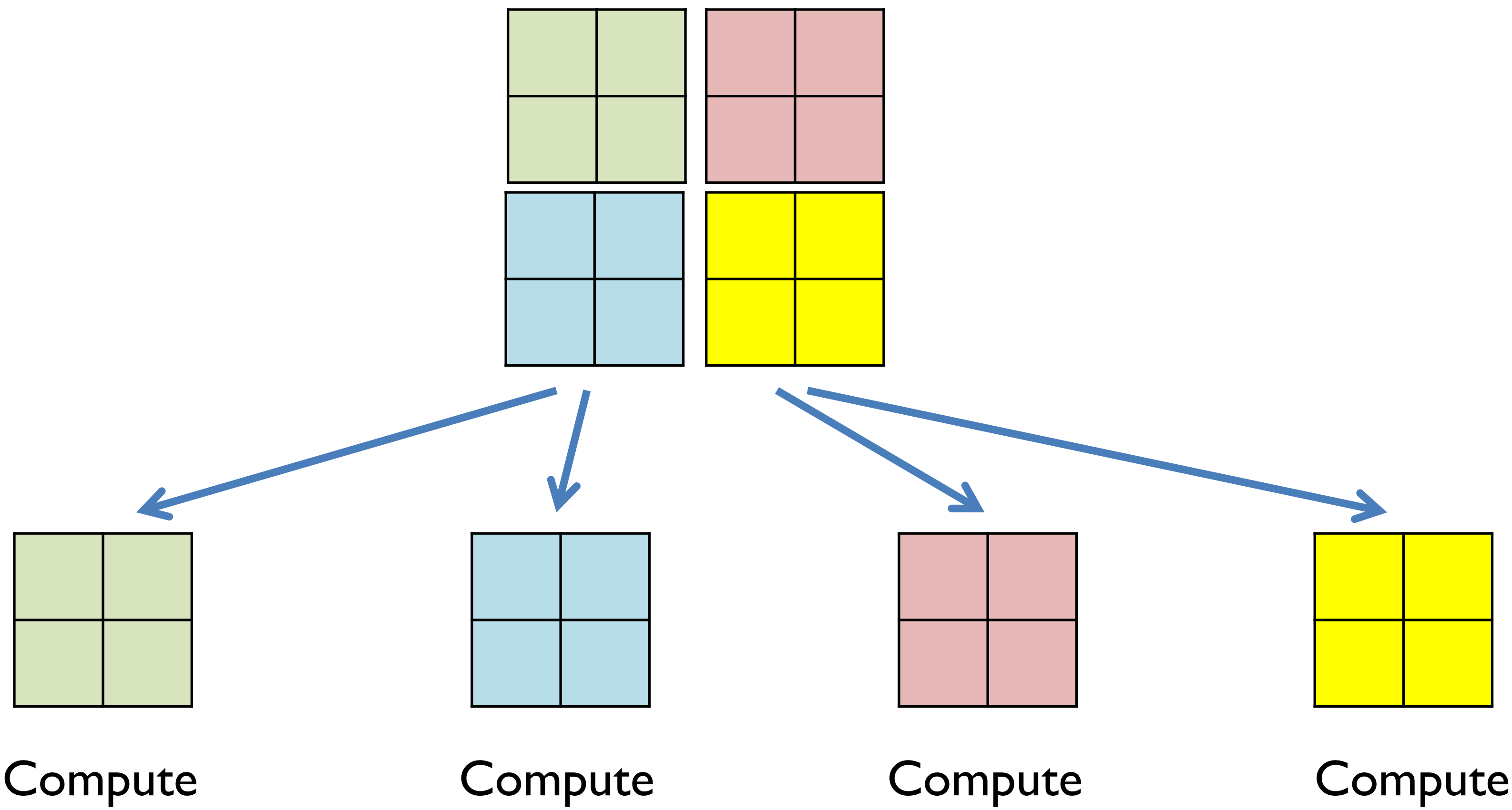
Programming Model

Iteration: foreach



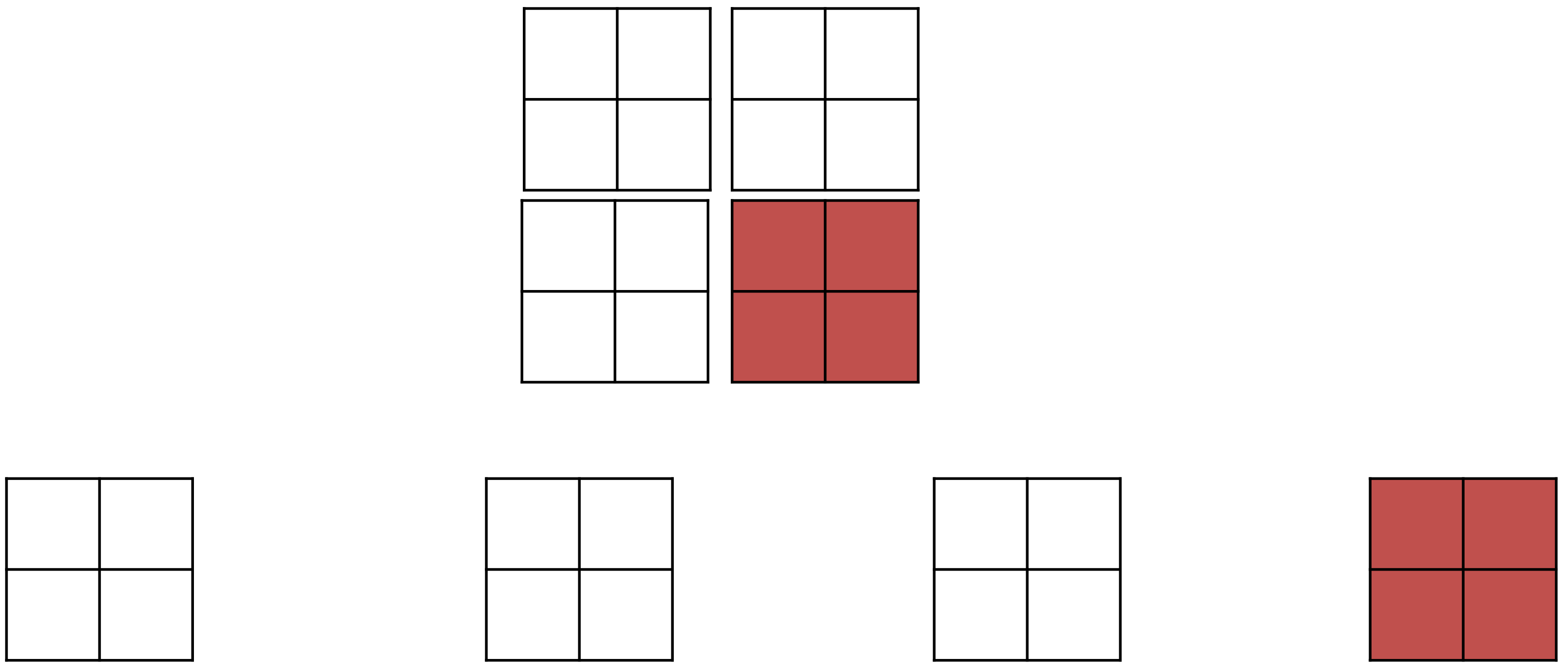
Programming Model

Iteration: foreach



Programming Model

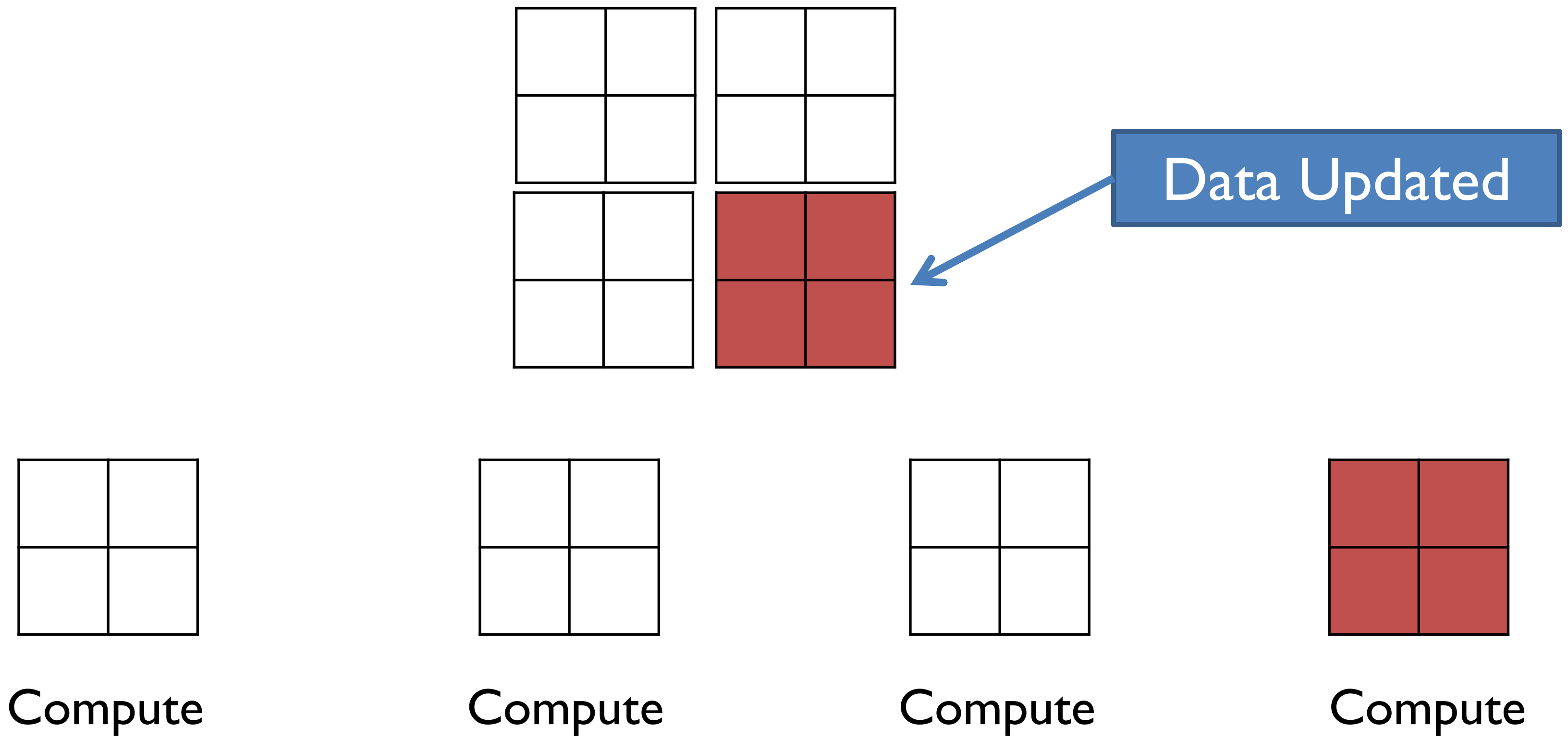
Incremental updates: onchange, update



Compute Compute Compute Compute

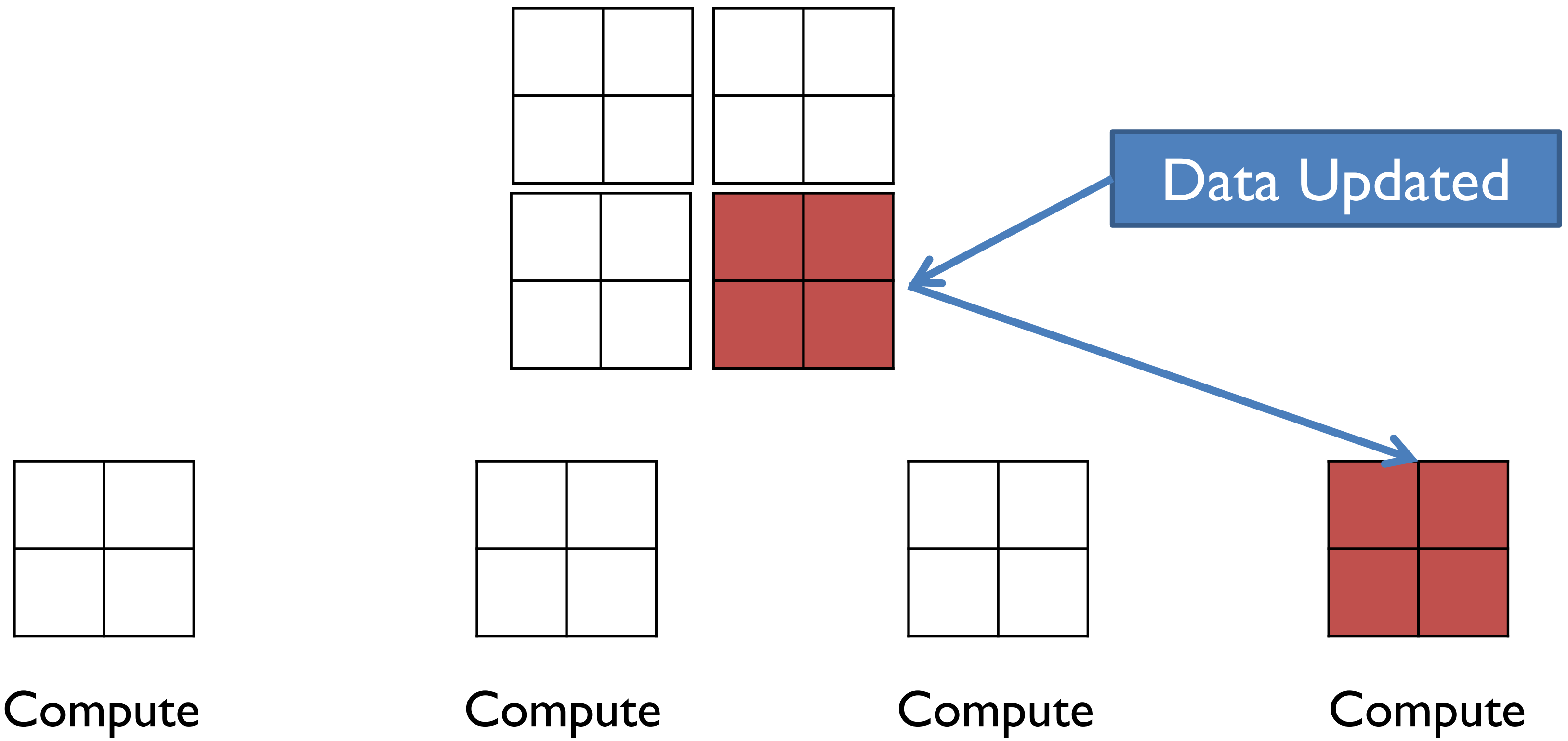
Programming Model

Incremental updates: onchange, update

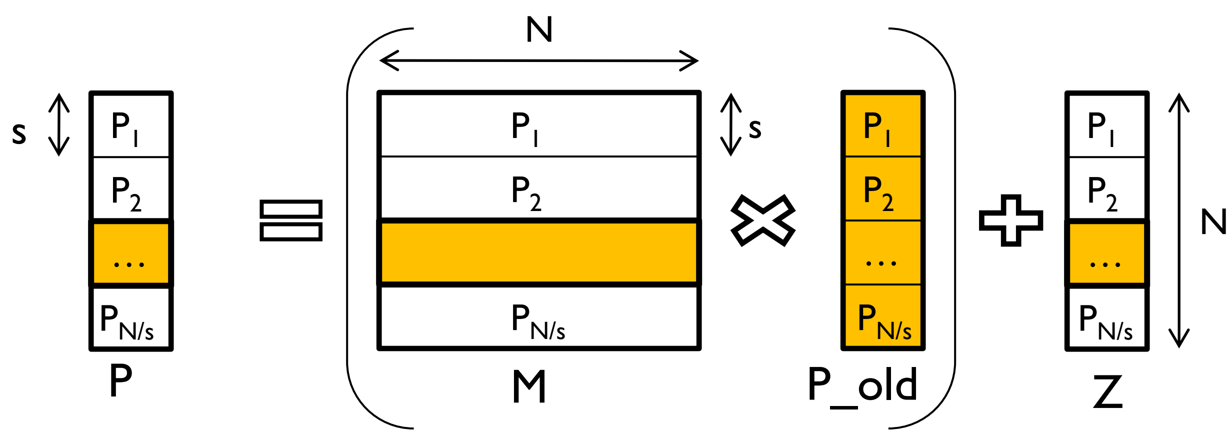


Programming Model

Incremental updates: onchange, update

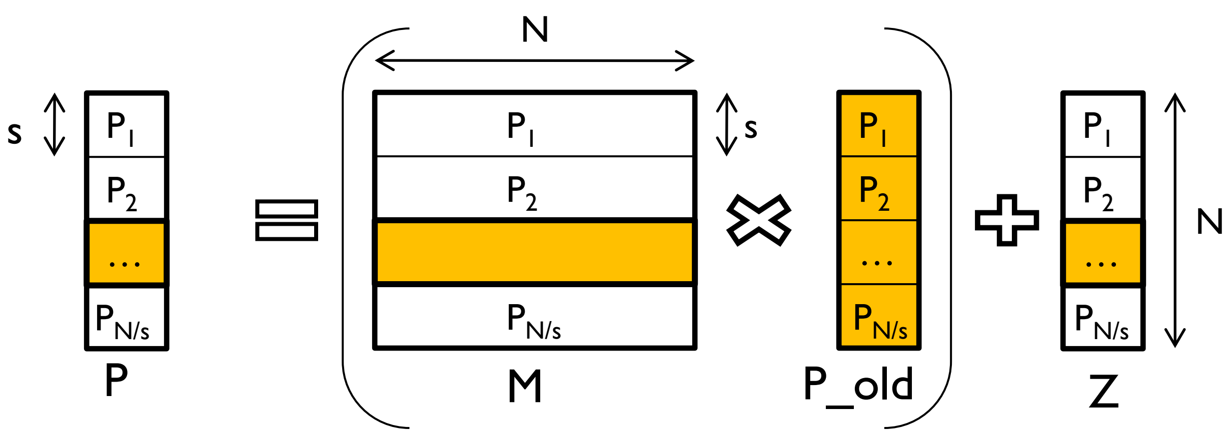


PageRank Using Presto




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s,1))
while(..){
  foreach(i,1:len,
     calculate(p=splits(P,i),m=splits(M,i),
               x=splits(P_old),z=splits(Z,i)) {
        p ← (m*x)+z
     }
  )
  P_old ← P
}
```

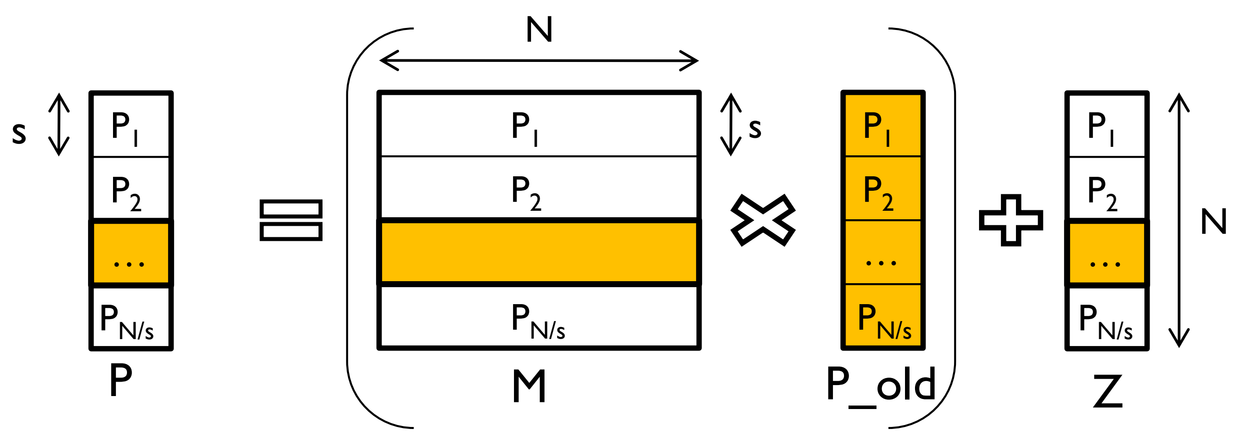
PageRank Using Presto




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s,1))
while(..){
  foreach(i,1:len,
     calculate(p=splits(P,i),m=splits(M,i),
               x=splits(P_old),z=splits(Z,i)) {
         p ← (m*x)+z
     }
  )
  P_old ← P
}
```

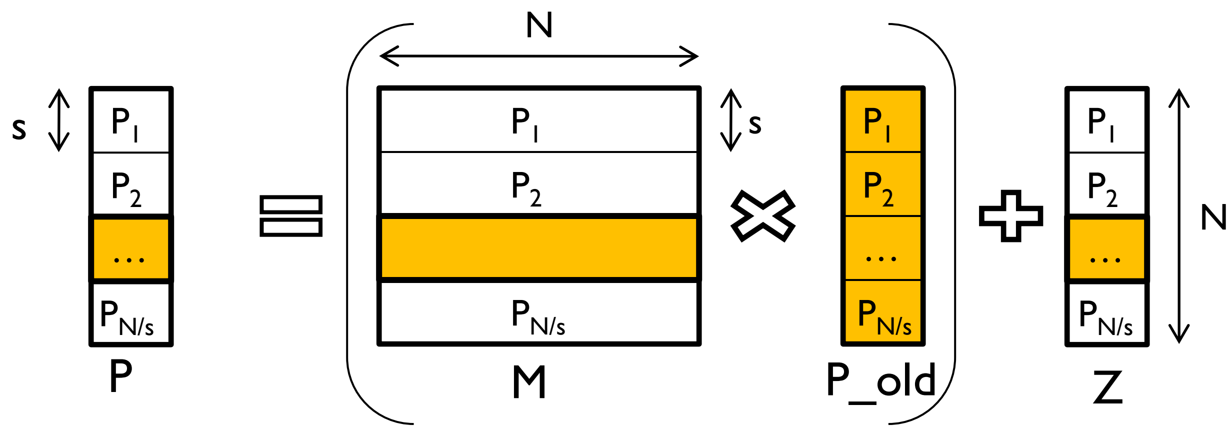
Create Distributed Array

PageRank Using Presto




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s,1))
while(..){
  foreach(i,1:len,
     calculate(p=splits(P,i), m=splits(M,i),
               x=splits(P_old), z=splits(Z,i)) {
         p ← (m*x)+z
     }
  )
  P_old ← P
}
```

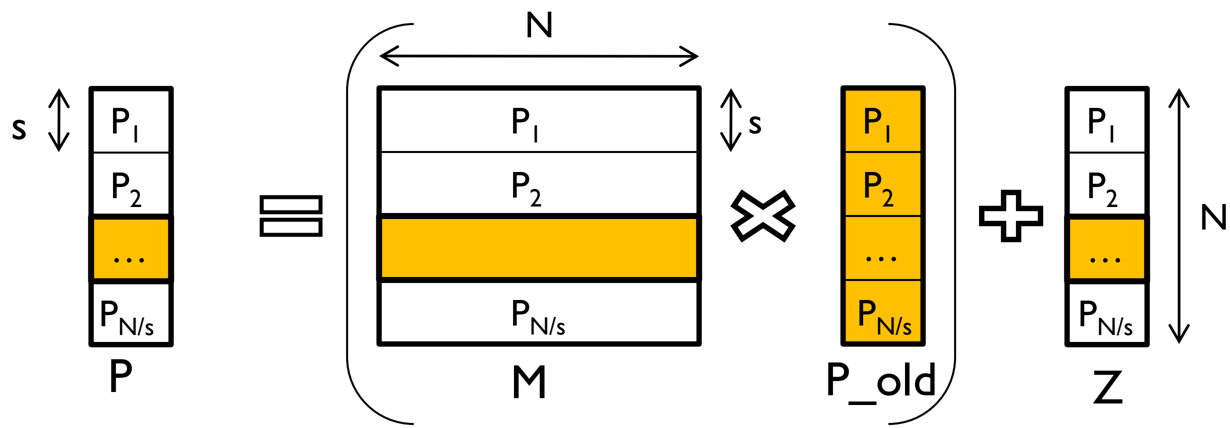
PageRank Using Presto




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s,1))
while(..){
  foreach(i,1:len,
     calculate(p=splits(P,i), m=splits(M,i),
               x=splits(P_old), z=splits(Z,i)) {
         p ← (m*x)+z
     }
  )
  P_old ← P
}
```

Execute function in a cluster

PageRank Using Presto

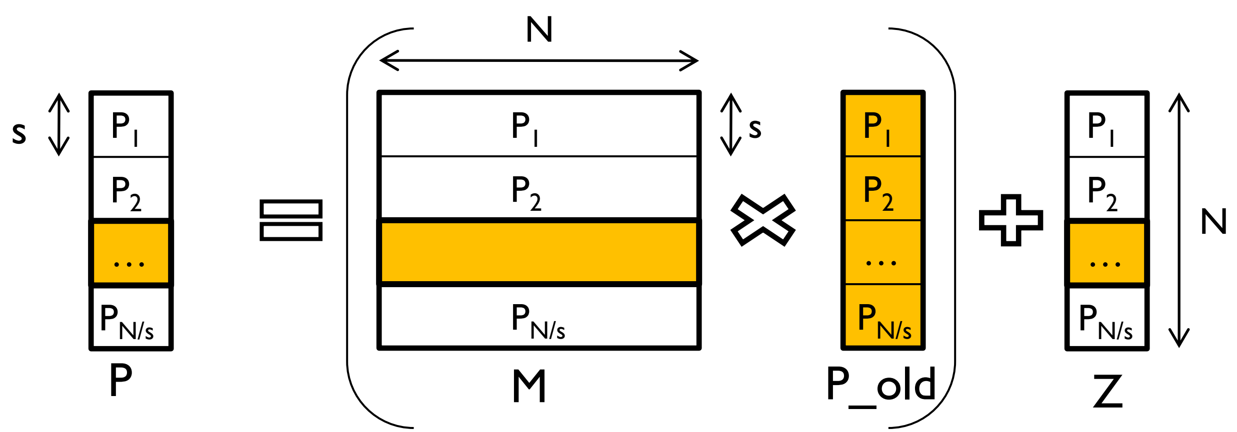



```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s,1))
while(..){
  foreach(i,1:len,
     calculate(p=splits(P,i), m=splits(M,i),
               x=splits(P_old), z=splits(Z,i)) {
         p ← (m*x)+z
       }
  )
  P_old ← P
}
```

Execute function in a cluster

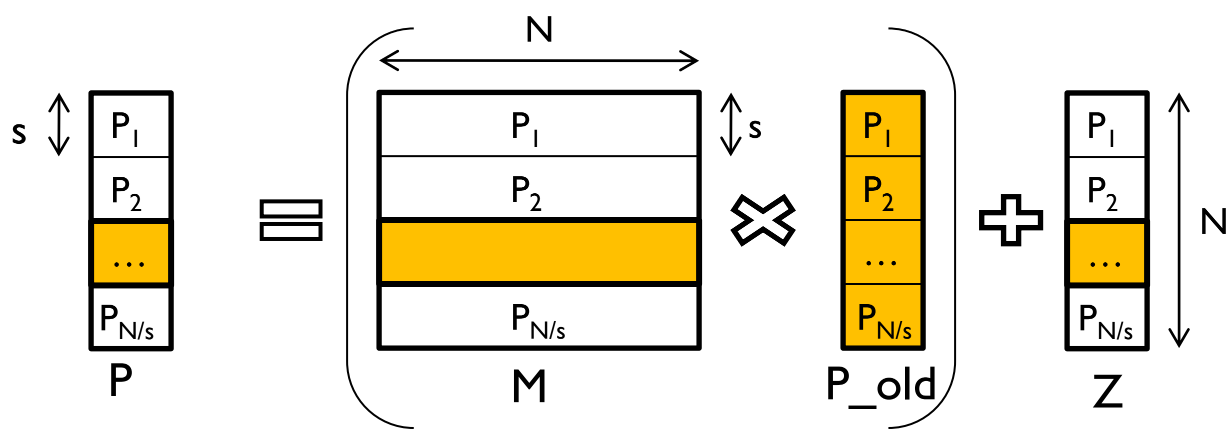
Pass array partitions

Incremental PageRank




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s1))
onchange(M) {
  while(..){
    foreach(i,1:len,
       calculate(p=splits(P,i), m=splits(M,i),
                 x=splits(P_old), z=splits(Z,i)) {
         p ← (m*x)+z
         update(p)
      })
    P_old ← P
  }}
```

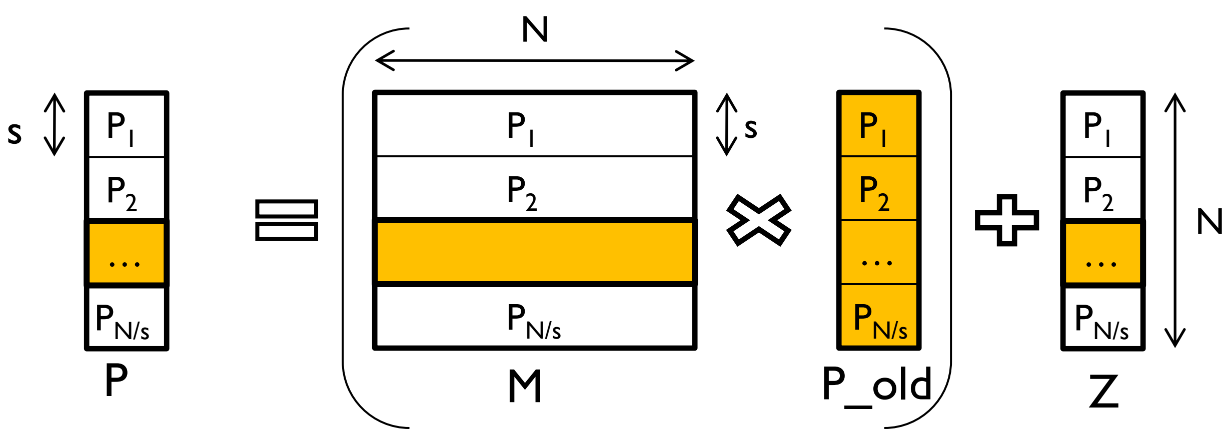
Incremental PageRank




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s1))
onchange(M) {
  while(..){
    foreach(i,1:len,
       calculate(p=splits(P,i), m=splits(M,i),
                 x=splits(P_old), z=splits(Z,i)) {
          p ← (m*x)+z
          update(p)
      })
    P_old ← P
}}
```

Execute when data changes

Incremental PageRank




```
M ← darray(dim=c(N,N),blocks=(s,N))
P ← darray(dim=c(N,1),blocks=(s1))
onchange(M) {
  while(..){
    foreach(i,1:len,
       calculate(p=splits(P,i), m=splits(M,i),
                 x=splits(P_old), z=splits(Z,i)) {
           p ← (m*x)+z
           update(p)
      })
    P_old ← P
  }}
```

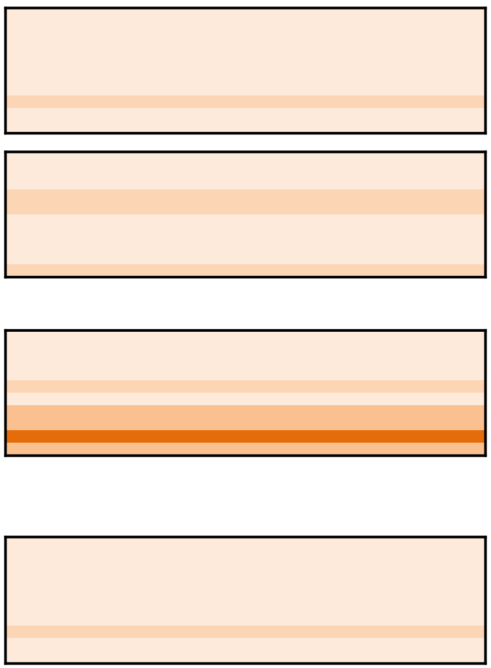
Execute when data changes

Update page rank vector

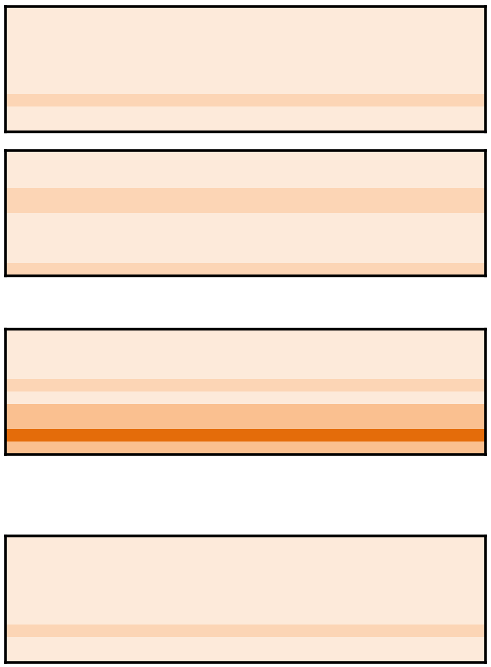
Outline

- Motivation
- Programming model
- Design
- Applications and Results

Dynamic Partitioning of Matrices

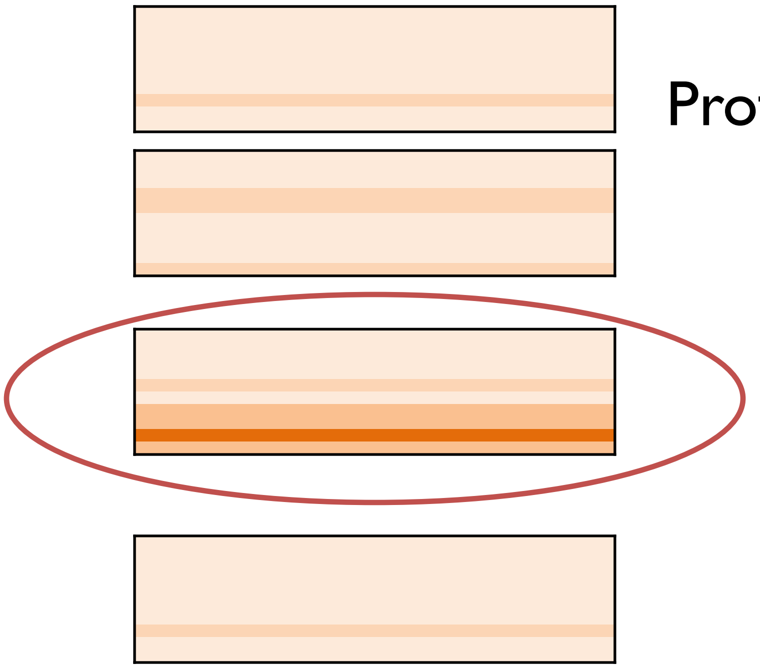


Dynamic Partitioning of Matrices



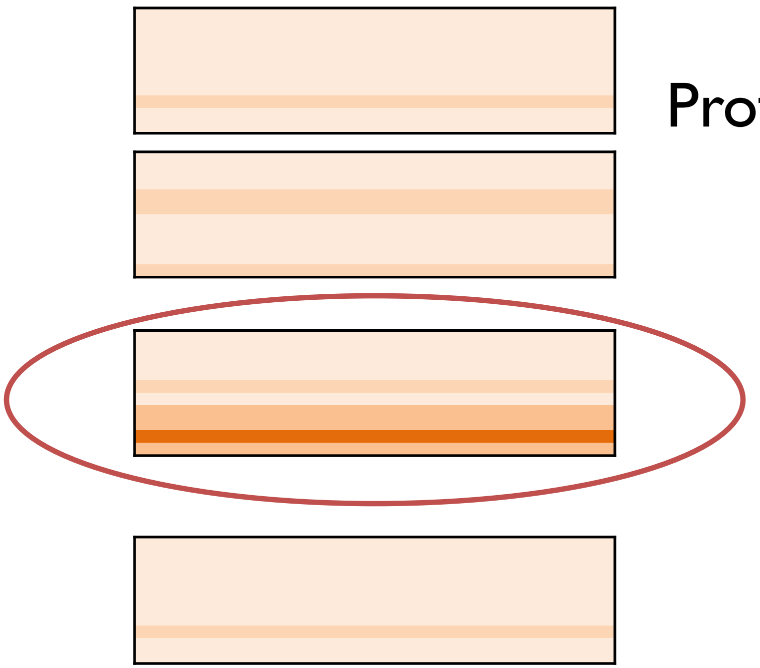
Profile execution

Dynamic Partitioning of Matrices



Profile execution

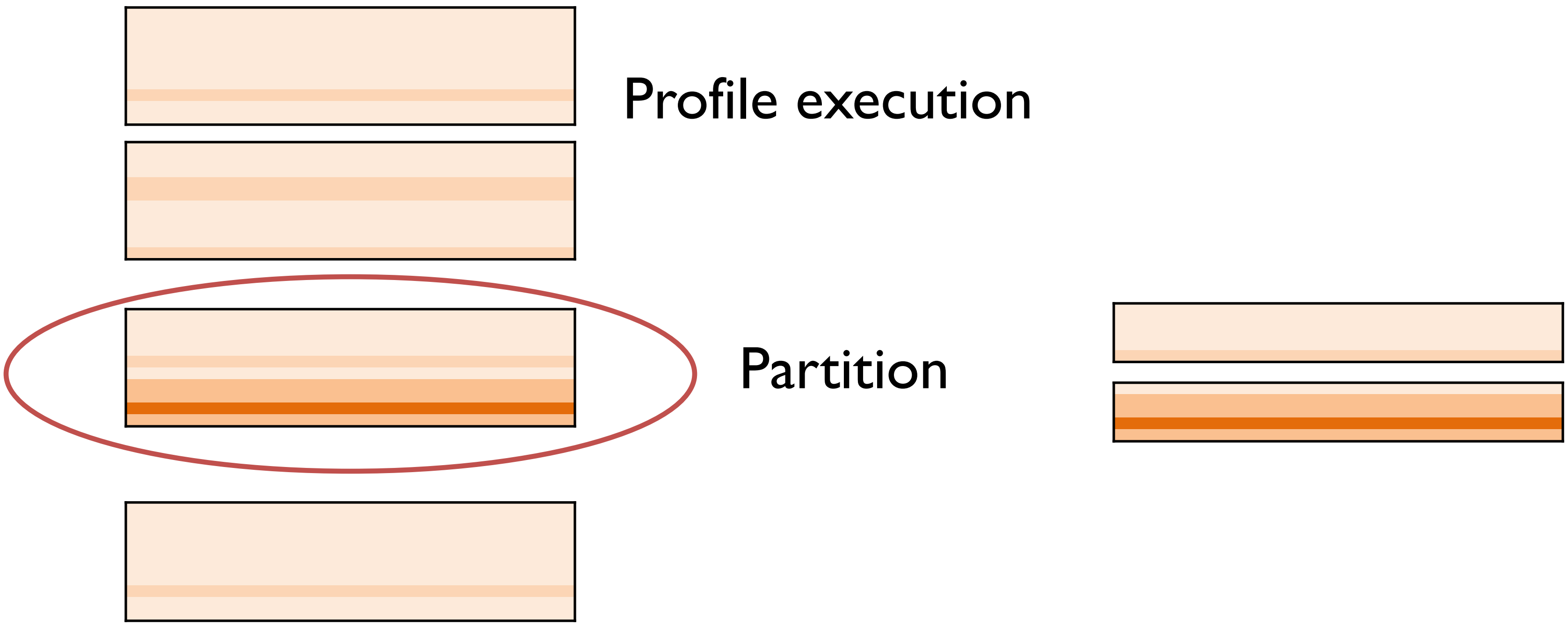
Dynamic Partitioning of Matrices



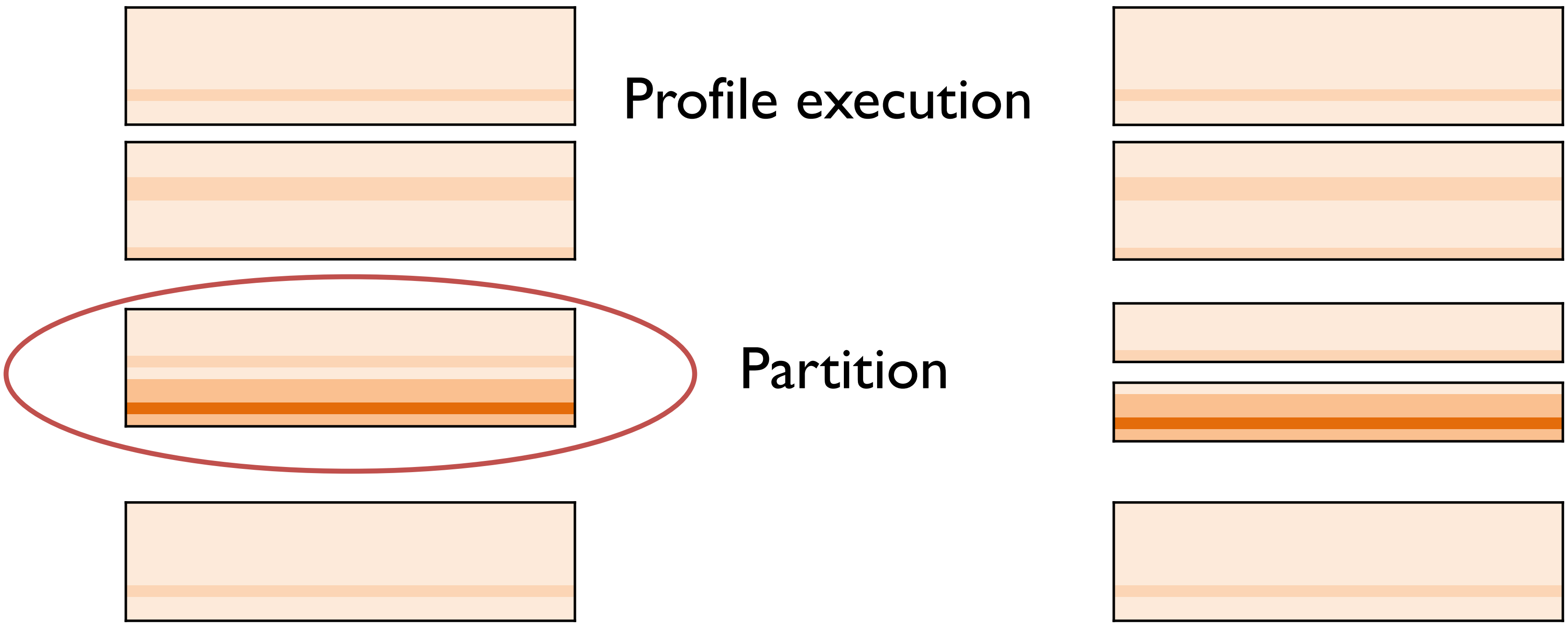
Profile execution

Partition

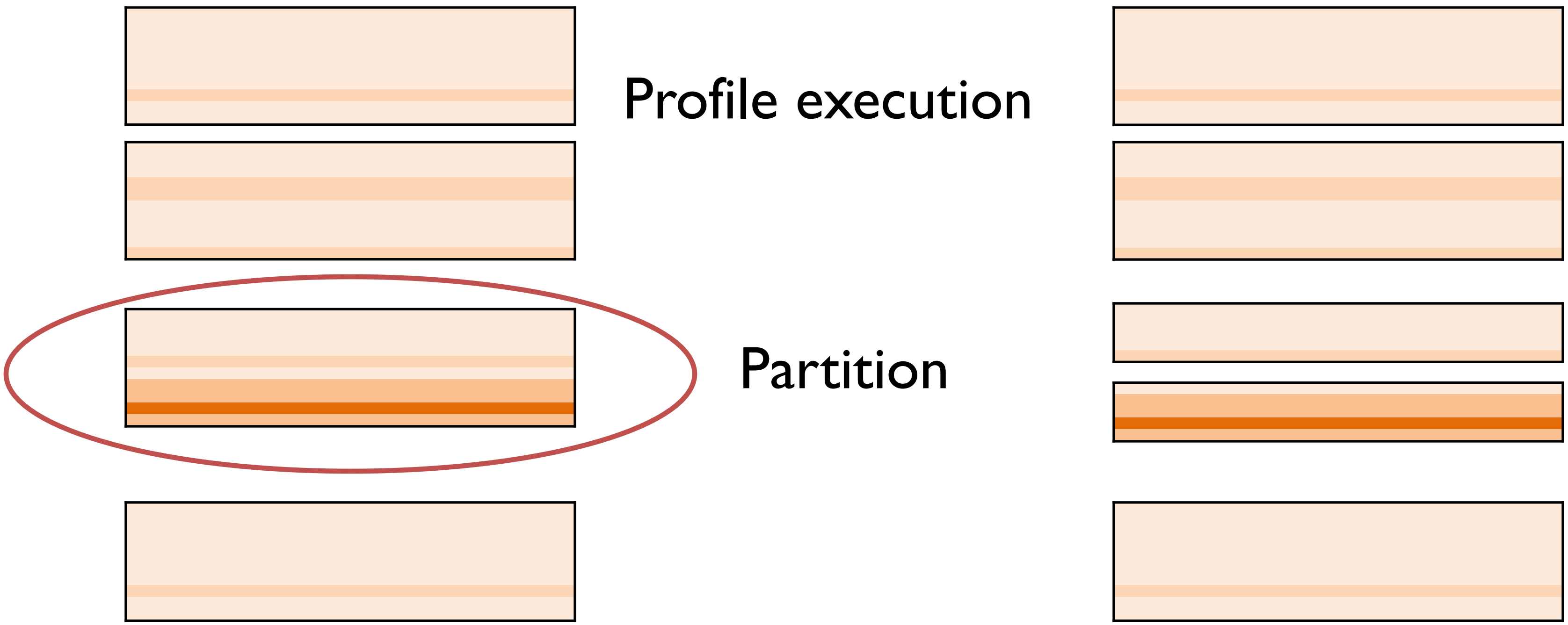
Dynamic Partitioning of Matrices



Dynamic Partitioning of Matrices

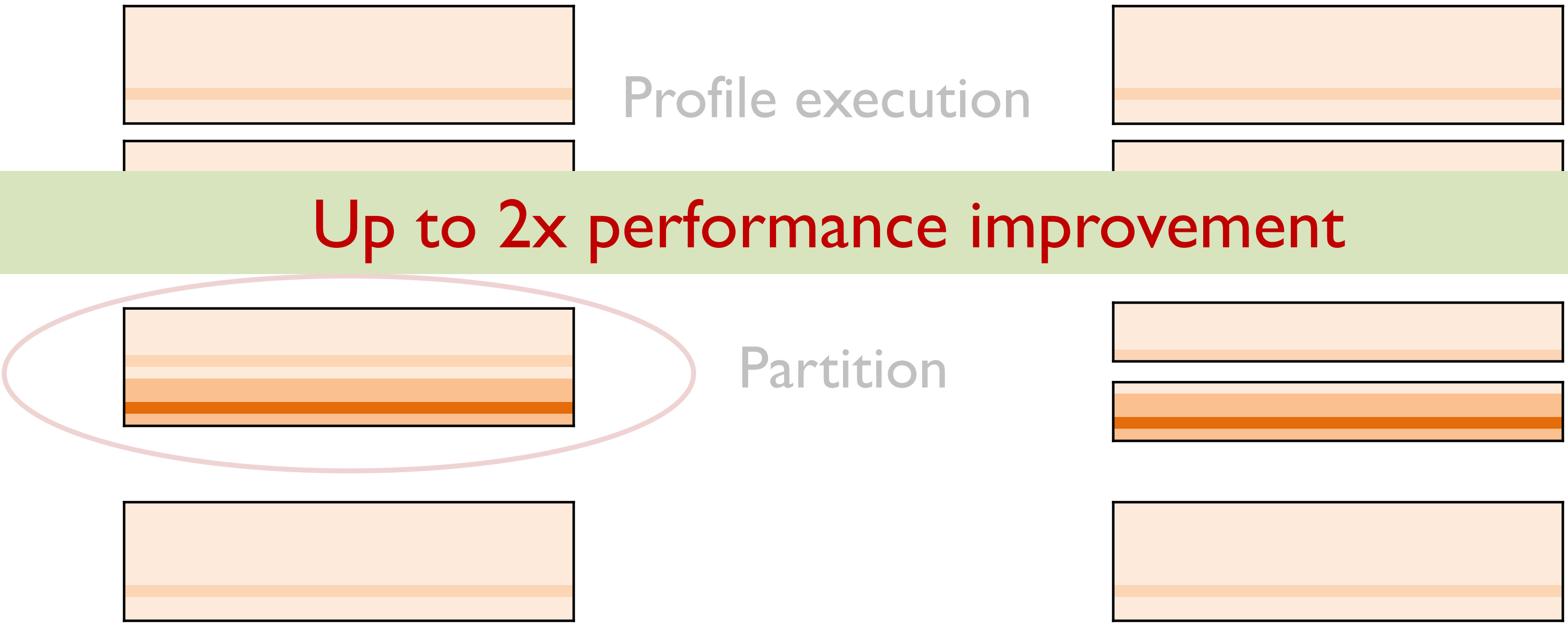


Dynamic Partitioning of Matrices



Programmers specify **size invariants**.

Dynamic Partitioning of Matrices

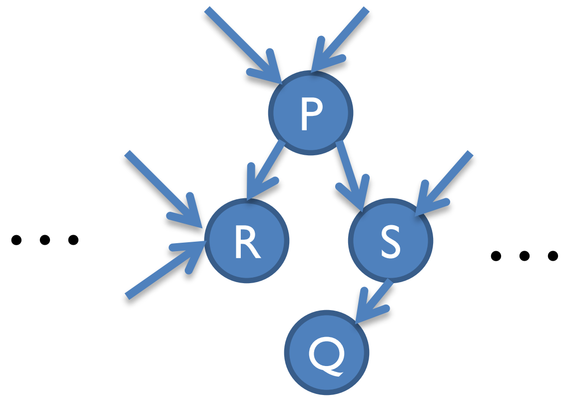


Up to 2x performance improvement

Programmers specify size invariants.

Incremental Updates Using Consistent Snapshots

Web Graph

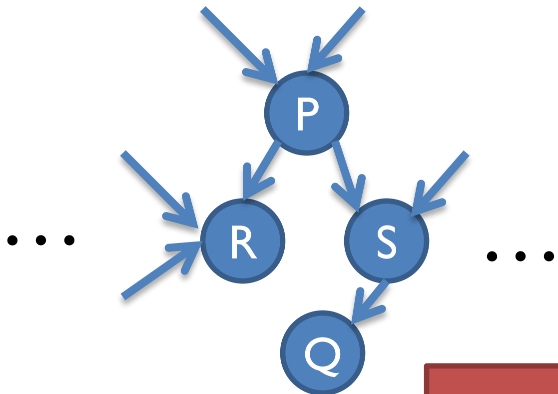


Adjacency Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ & & \cdots & \end{pmatrix}$$

Incremental Updates Using Consistent Snapshots

Web Graph



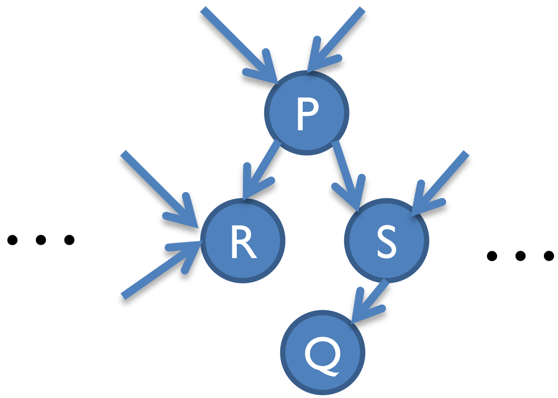
Adjacency Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ & \cdots & & \end{pmatrix}$$

onchange(M_1)

Incremental Updates Using Consistent Snapshots

Web Graph



Adjacency Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ & \multicolumn{2}{c}{\cdots} & \end{pmatrix}$$

Page Rank

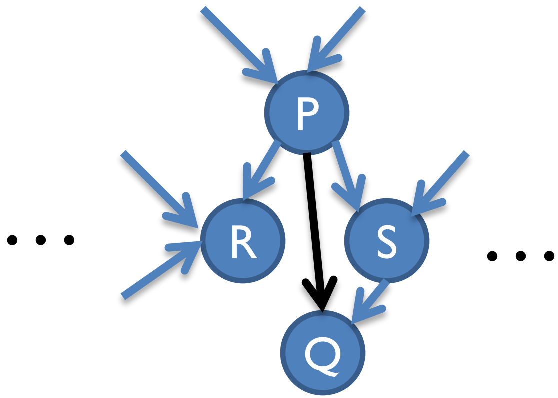
$$\begin{pmatrix} 0.035 \\ 0.006 \\ 0.008 \\ 0.032 \end{pmatrix}$$

…

update → P_1

Incremental Updates Using Consistent Snapshots

Web Graph



Adjacency Matrix

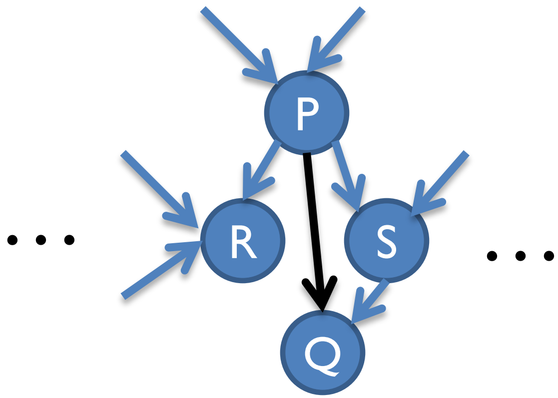
$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ & \cdots & & \end{pmatrix}$$

Page Rank

$$\begin{pmatrix} 0.035 \\ 0.006 \\ 0.008 \\ 0.032 \end{pmatrix}$$
$$\cdots$$

Incremental Updates Using Consistent Snapshots

Web Graph



Adjacency Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ & \cdots & & \end{pmatrix}$$

Page Rank

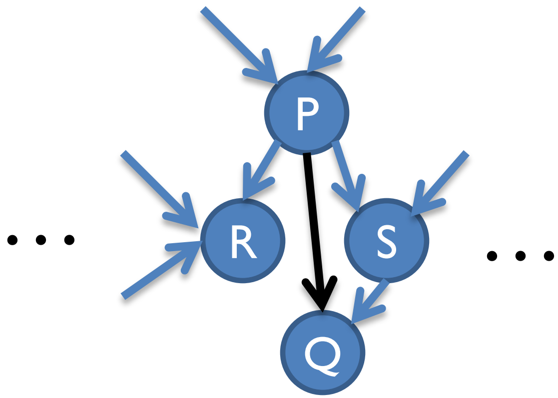
$$\begin{pmatrix} 0.035 \\ 0.006 \\ 0.008 \\ 0.032 \end{pmatrix} \cdots$$

onchange(M_2)

$$\begin{pmatrix} 0 & \color{red}{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Incremental Updates Using Consistent Snapshots

Web Graph



Adjacency Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & \\ & \cdots & & \end{pmatrix}$$

Page Rank

$$\begin{pmatrix} 0.035 \\ 0.006 \\ 0.008 \\ .032 \end{pmatrix} \cdots$$

update → P_2

$$\begin{pmatrix} 0 & \color{red}{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0.035 \\ \color{red}{0.028} \\ 0.008 \\ 0.032 \end{pmatrix}$$

Versioned Distributed Arrays

Mechanics of versioning

- *update*: Increment version number
- *onchange*: Bind a version number for the array before executing the handler

Outline

- Motivation
- Programming model
- Design
- Applications and Results

Applications Implemented in Presto

Application	Algorithm	Presto LOC
PageRank	Eigenvector calculation	41
Triangle counting	Top-K eigenvalues	121
Netflix recommendation	Matrix factorization	130
Centrality measure	Graph algorithm	132
k-path connectivity	Graph algorithm	30
k-means	Clustering	71
Sequence alignment	Smith-Waterman	64

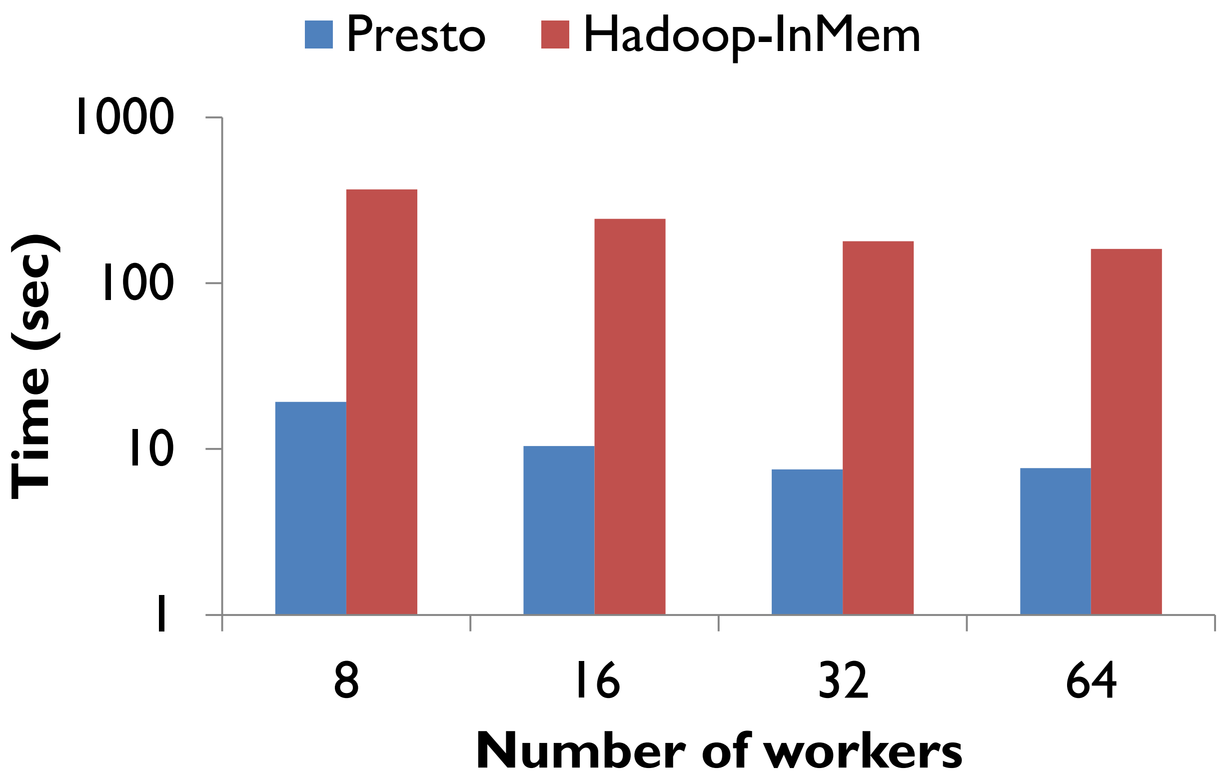
Applications Implemented in Presto

Application	Algorithm	Presto LOC
PageRank	Eigenvector calculation	41
Triangle counting	Top-K eigenvalues	121
Centrality measure	Graph algorithm	132
k-path connectivity	Graph algorithm	30
k-means	Clustering	71
Sequence alignment	Smith-Waterman	64

Fewer than 140 lines of code

Presto is Fast !

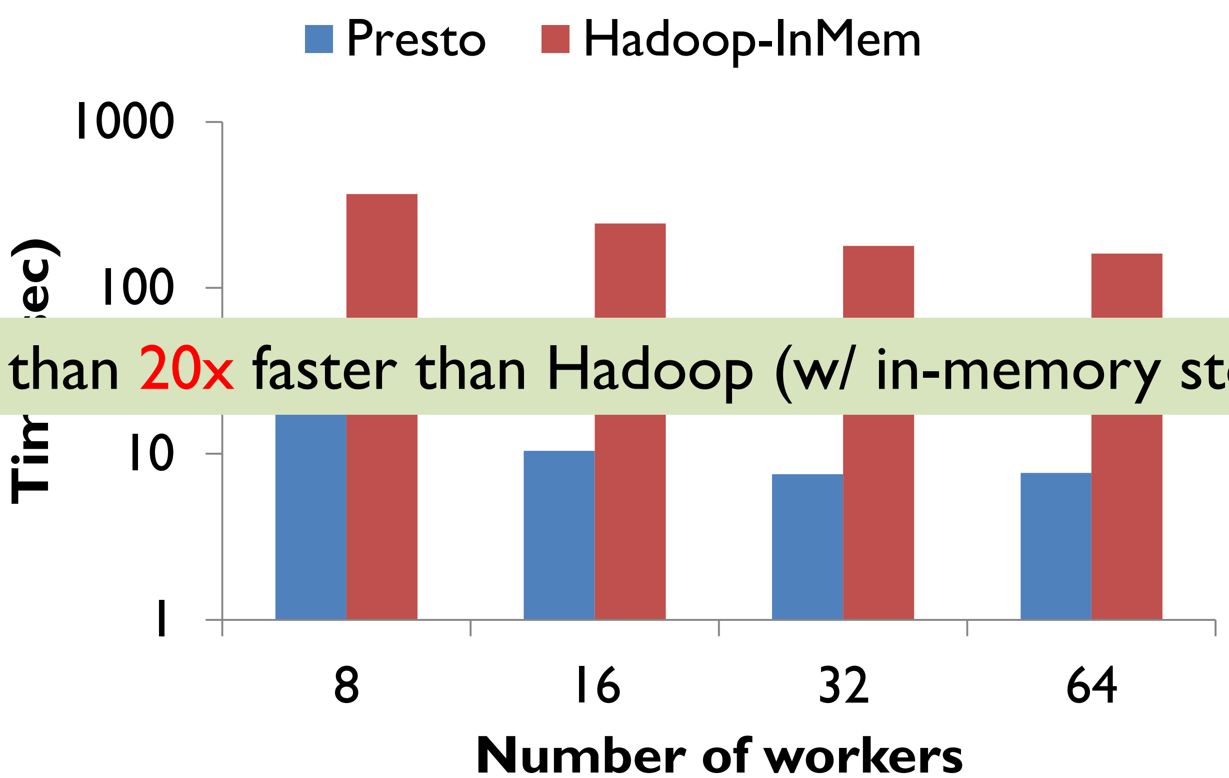
PageRank per-iteration execution time



Data: 100M nodes, 1.2B edges. Setup: 10G network. 12 cores, 96GB RAM.

Presto is Fast !

PageRank per-iteration execution time



More than 20x faster than Hadoop (w/ in-memory storage)

Data: 100M nodes, 1.2B edges. Setup: 10G network. 12 cores, 96GB RAM.

More in the Paper

- Memory management, caching of partitions
- Scheduling operations
- Storage driver interface to HBase
- Fault tolerance

Conclusion

Linear Algebra is a powerful abstraction

Easily express machine learning, graph algorithms

Challenges: Sparse matrices, Incremental data

Presto – prototype extends R

Open source version soon !