

# Asynchronous Algorithms in MapReduce



Karthik Kambatla

Naresh Rapolu

Prof. Suresh Jagannathan

Prof. Ananth Grama

# Agenda

- Background
  - Asynchronous Algorithms, MapReduce
- Motivation
- Relaxed semantics for MapReduce
- Evaluation
- Conclusions

# Asynchronous Algorithms

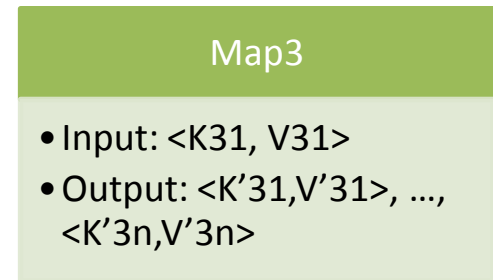
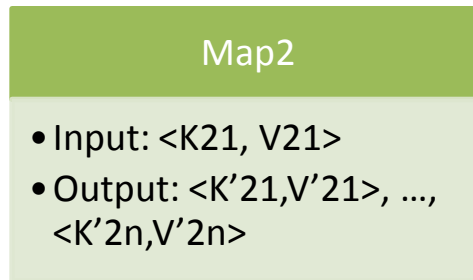
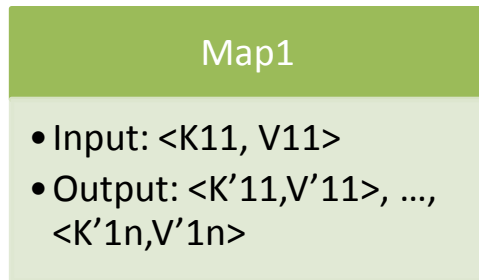
- Improve **performance** in **parallel** environments.
  - Infrequent **synchronization** reduces communication
  - Can increase **data locality**
  - Examples
    - Graph algorithms, Numerical methods, Classification, etc.
- More **pronounced gains** in **distributed** environments
  - Higher communication and data-movement costs

# MapReduce

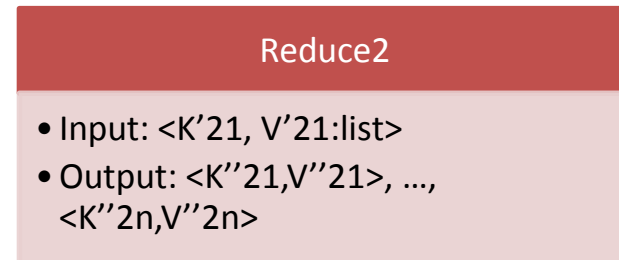
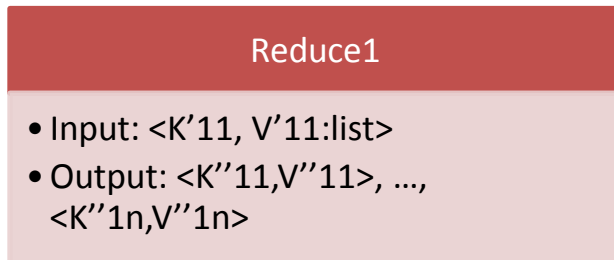
- Distributed execution engine: **easy-to-use**, highly **scalable**, **fault-tolerant**.
  - Processes big data on commodity hardware
  - Great for embarrassingly data-parallel applications
- Usage
  - Data analytics and other operations on web-scale data
  - Does it lend itself to other **kinds of parallelism** and **optimizations**?

# MapReduce Model

INPUT <key, value>:list

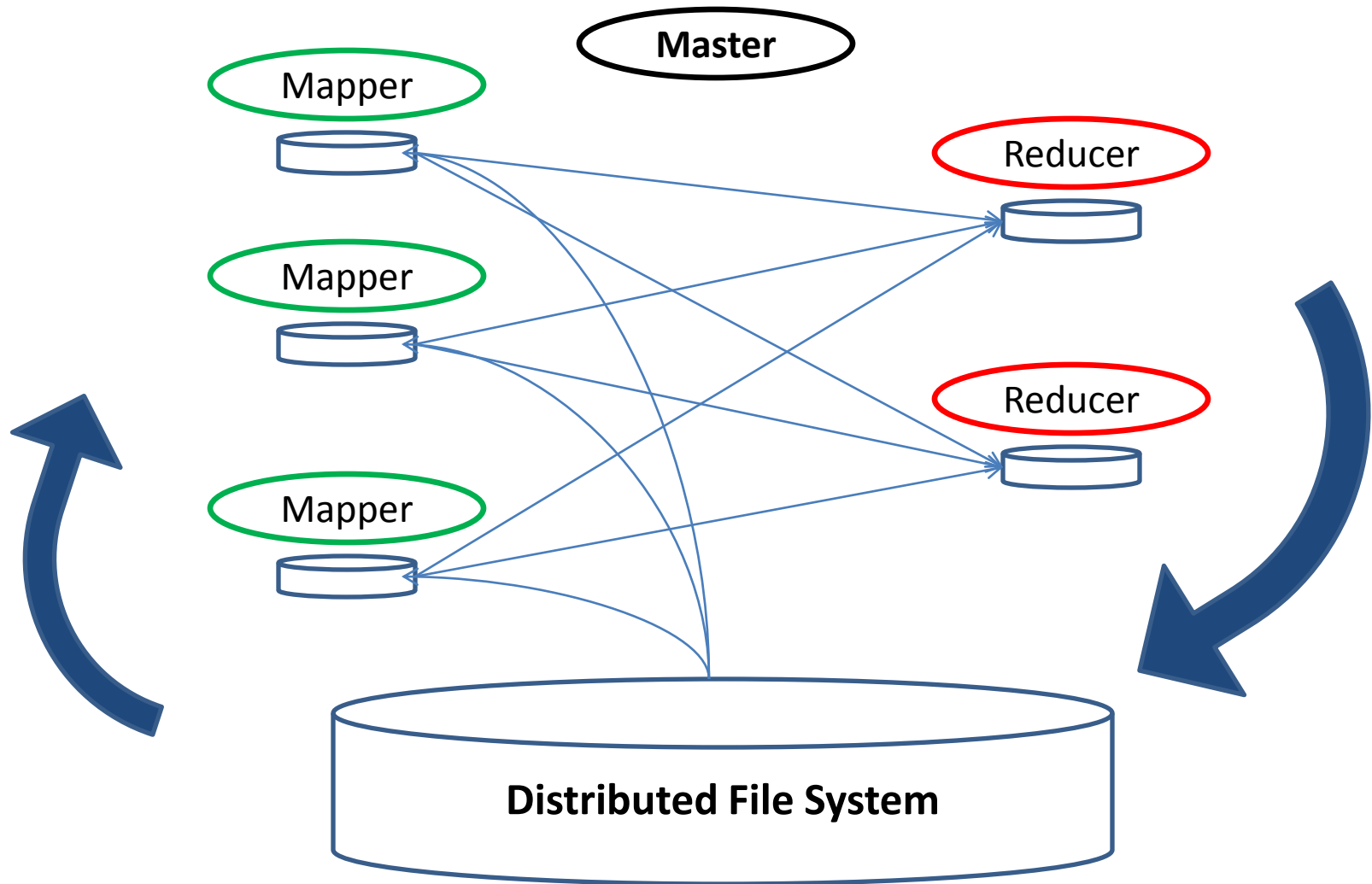


INTERMEDIATE <key, value:list>: list



OUTPUT value:list

# MapReduce: Data Flow



# Observations

- I/O takes **longer** than Computation
  - More pronounced in iterative algorithms
    - Especially when the input doesn't change much across iterations
- Strict **synchronization barrier**
  - Between map and reduce
  - Between iterations in iterative MapReduce
- Limited applicability of known optimizations
  - **Asynchronous algorithms**, speculative parallelism etc.

# Relaxed Synchronization

- Hierarchical MapReduce: **local** and **global**
  - Each iteration in a MapReduce job has multiple iterations of local MapReduce
  - **Partial synchronization**: after every local iteration, synchronize on subset of the data
  - **Global synchronization**: at the end of local iterations, synchronize on the whole data across all machines
  - Input data **partitioning**
    - Fewer dependencies across partitions

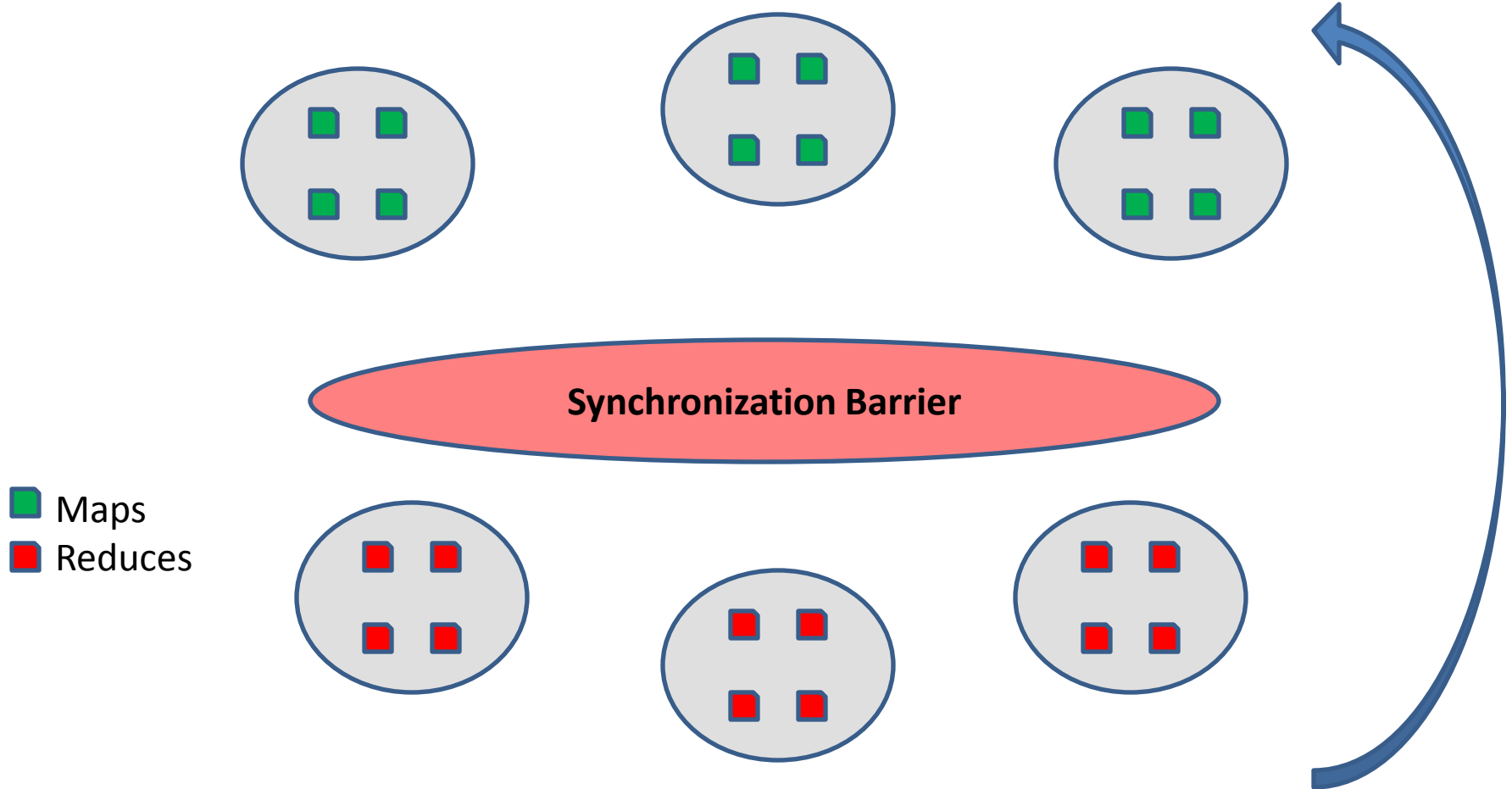
# Illustrative Example: PageRank

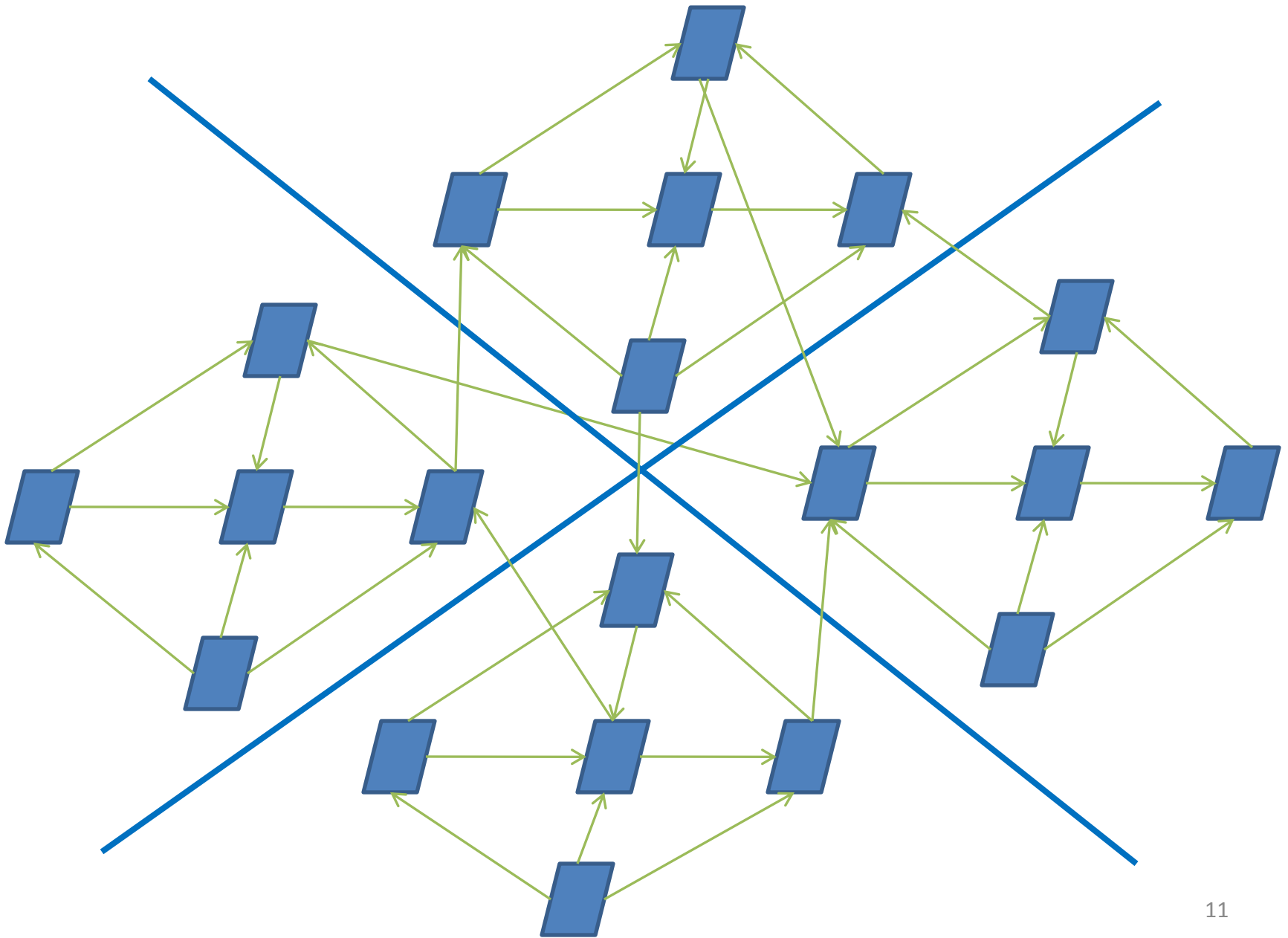
- Weighted in-links determine the rank of a node.

$$PR_d = (1 - \chi) + \chi * \sum_{(s,d) \in E} s.pagerank / s.outlinks$$

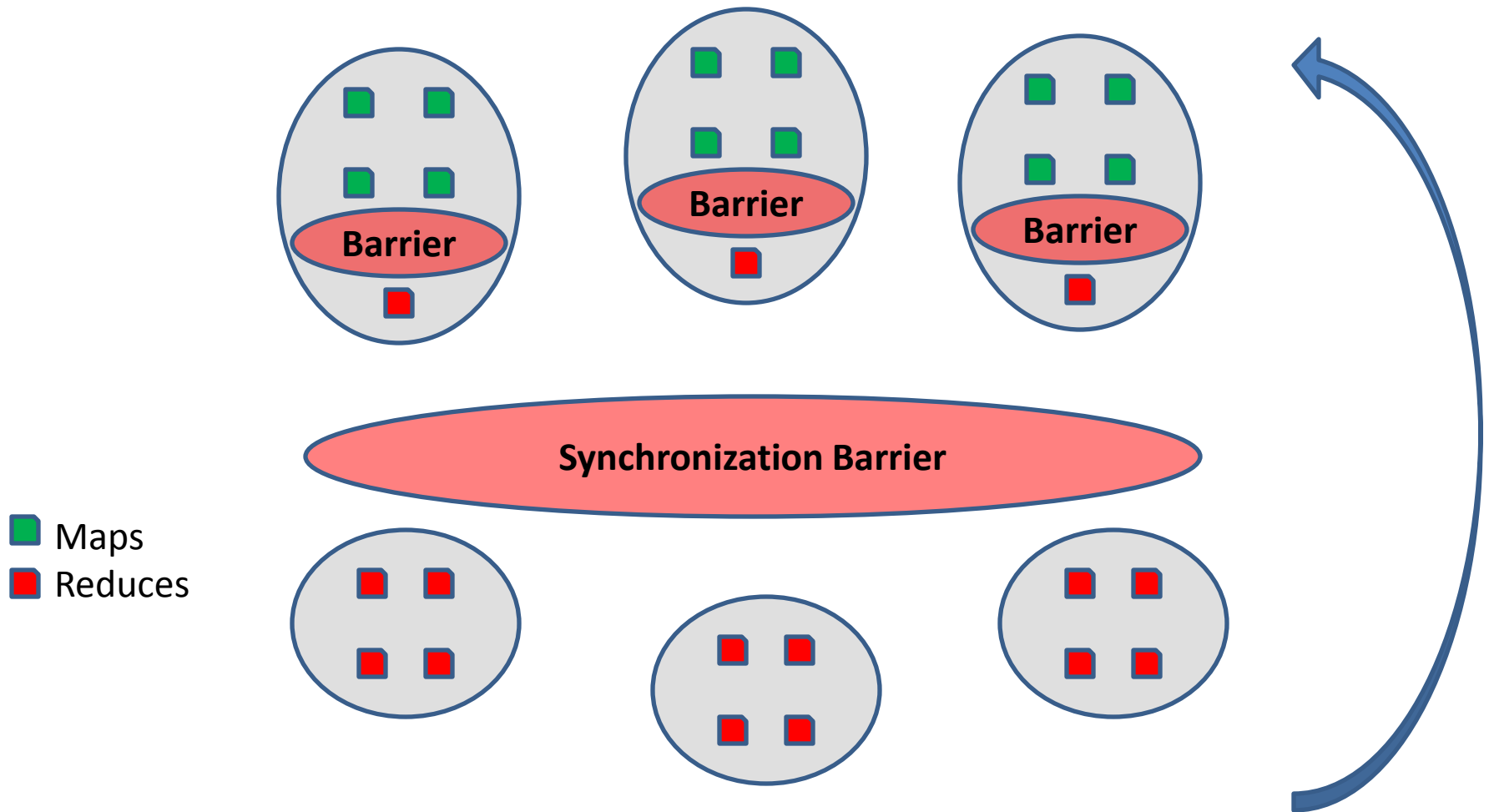
- Implementation: **Power method**
  - Each iteration executes a MapReduce
    - **map**: each node emits (dest, pagerank/#outlinks)
    - **reduce**: sums up all the in-link weights and computes pagerank
  - Input: power-law
    - Many strongly-connected components with few edges across components

# General PageRank





# Relaxed PageRank



# Semantics: Iterative MapReduce

$l, \sigma \Rightarrow \sigma(l)$  (LOCAL-LOOKUP)

$l, \lambda \Rightarrow \lambda(l)$  (GLOBAL-LOOKUP)

$Apply(\mathbf{I}, \langle e, f_m, f_r, l \rangle) \Rightarrow_g \mathbf{I} e f_m f_r l$  (APPLY-ITER)

$$\frac{\mathbf{while}(cond_g) \mathbf{G} cond_l f_m f_r l_g, \lambda \Rightarrow_g l'_g, \lambda'}{\mathbf{I} cond_g f_m f_r l_g, \lambda \Rightarrow_g l'_g, \lambda'}$$
 (ITER-MAPRED)

$$\frac{\mathbf{while} cond \mathbf{map} (\mathbf{L} f_m f_r) \bar{l}_l, \sigma \Rightarrow_l \bar{l}'_l, \sigma' \quad \mathbf{agg} \bar{l}'_l, \sigma, \lambda \equiv l'_g, \sigma, \lambda' \quad \mathbf{fold} f_r l'_g, \lambda \Rightarrow_g l''_g, \lambda'}{\mathbf{G} cond f_m f_r l_g, \lambda, \sigma \Rightarrow_g l''_g, \lambda', \sigma'}$$
 (MAPRED-GLOBAL)

$$\frac{\mathbf{map} f_m l_l, \sigma \Rightarrow_l l''_l, \sigma' \quad \mathbf{fold} f_r l''_l, \sigma \Rightarrow_l l'_l, \sigma'}{\mathbf{L} f_m f_r l_l, \sigma \Rightarrow_l l'_l, \sigma'}$$
 (MAPRED-LOCAL)

$ch l_g \equiv \bar{l}_l \equiv \{l_l \mid l_l \subset l_g \ \& \ \cap_{l_l \in \bar{l}_l} l_l = \phi\}; \forall l_i, \sigma_i [l_i \mapsto \lambda(l)]$  (CHUNKIFY)

$agg \bar{l}_l \equiv l_g \equiv \cup_{l_l \in \bar{l}_l} l_l; \forall l_i, \lambda [l \mapsto l_i]$  (AGGREGATE)

# Realizing the semantics

- Code *gmap*, *greduce*, *lmap*, *lreduce*
  - *lmap*, *lreduce* use `EmitLocalIntermediate()` and `Emit Local()`
  - Synchronized hashtables for local storage

```
gmap(xs : X list) {  
  
  while(no-local-convergence-intimated) {  
  
    for each element x in xs {  
      lmap(x); // emits lkey, lval  
    }  
  
    lreduce(); // operates on the output of lmap functions  
  }  
  
  for each value in lreduce-output{  
    EmitIntermediate(key, value);  
  }  
}
```

# Evaluation

- 3 applications
  - PageRank (**mat-vec**: eigen value and linear system solvers)
  - Single Source Shortest Path (MST, transitive closure, etc.)
  - K-Means (**clustering**)
- Test bed
  - 8 Amazon EC2 Large Instances
    - 64-bit compute units with 15 GB RAM, 4x 420 GB storage
    - Hadoop 0.20.1; 4 GB heap space per slave

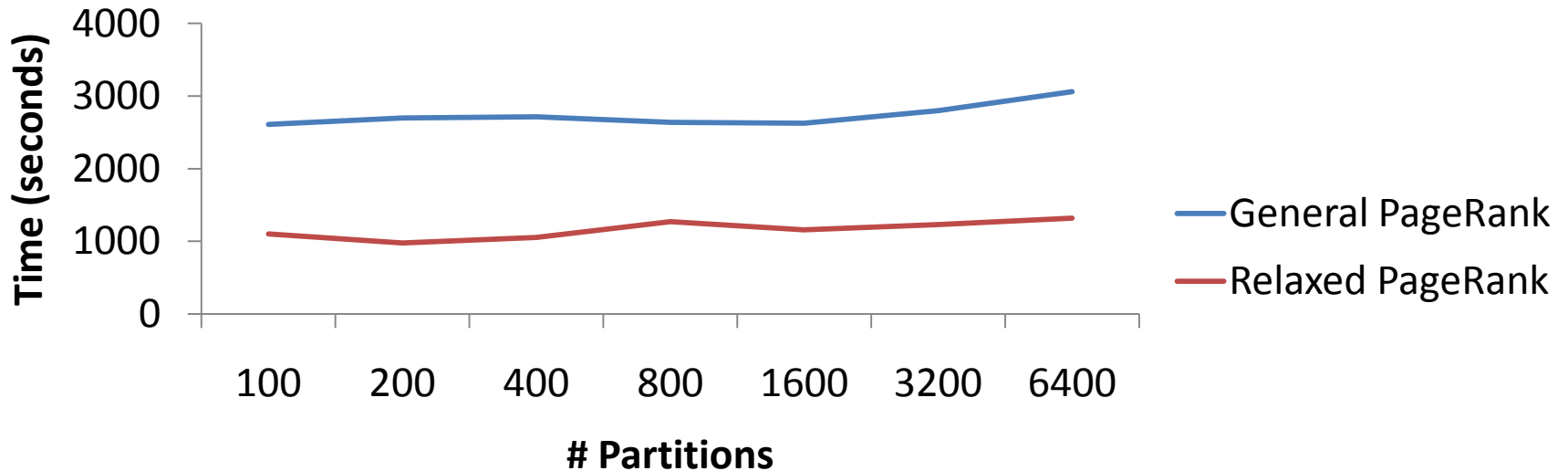
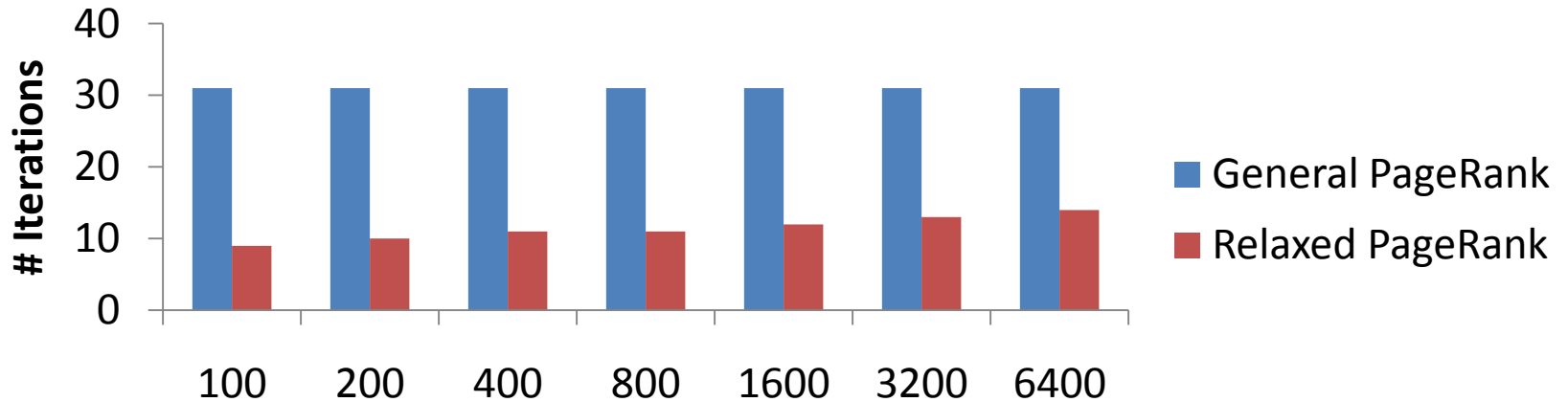
# PageRank

- Input: Partitioned using METIS ( < 10 seconds)

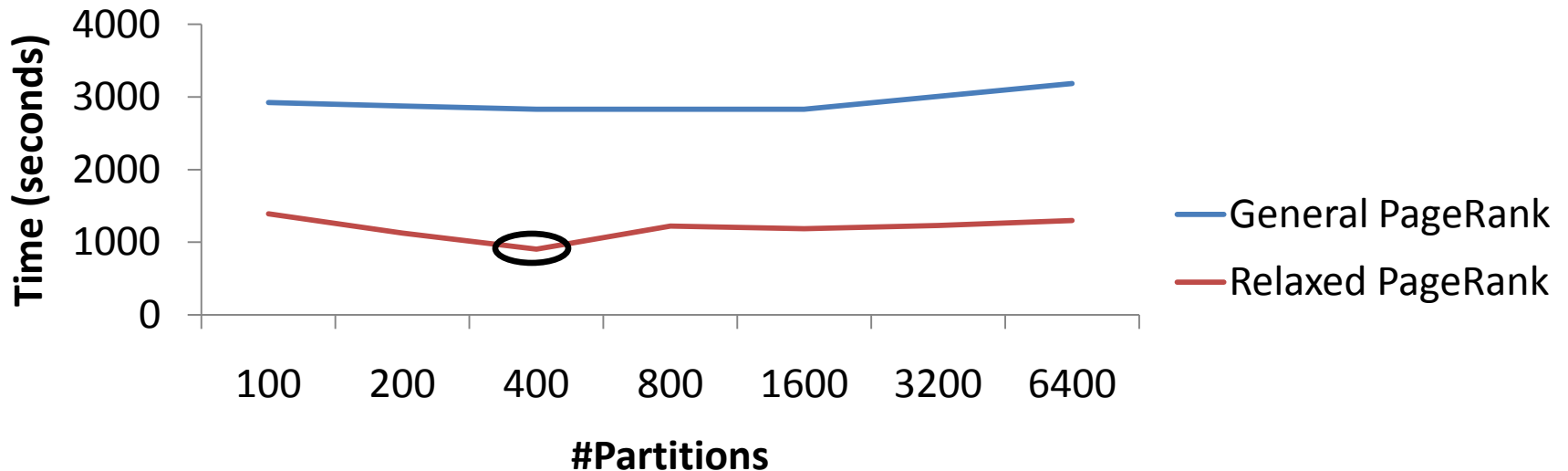
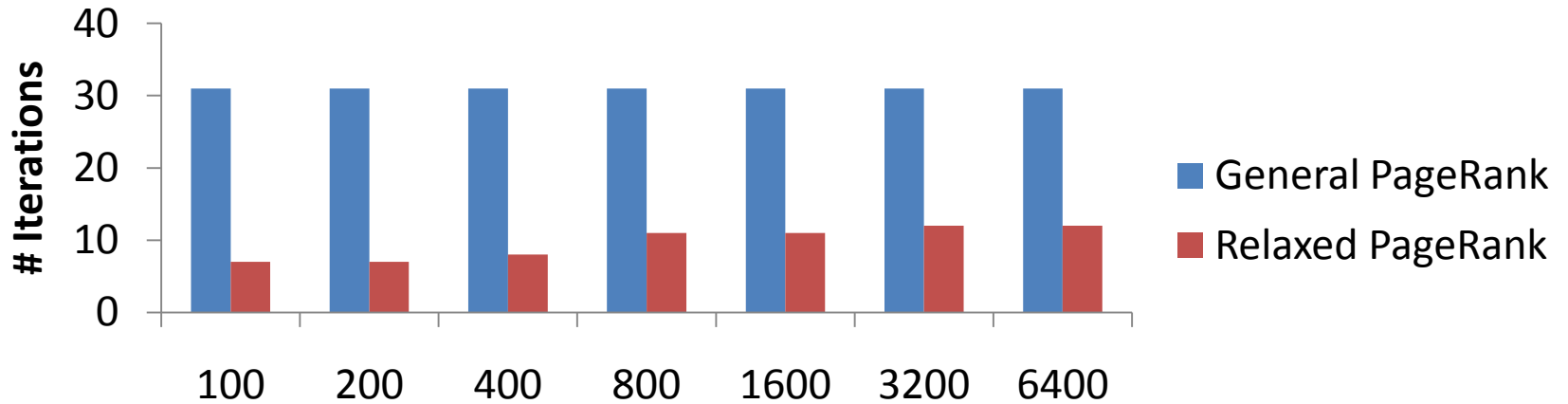
	Graph A	Graph B
Nodes	280,000	100,000
Edges	3 million	3 million

- Damping factor = 0.85

# Performance: Graph A

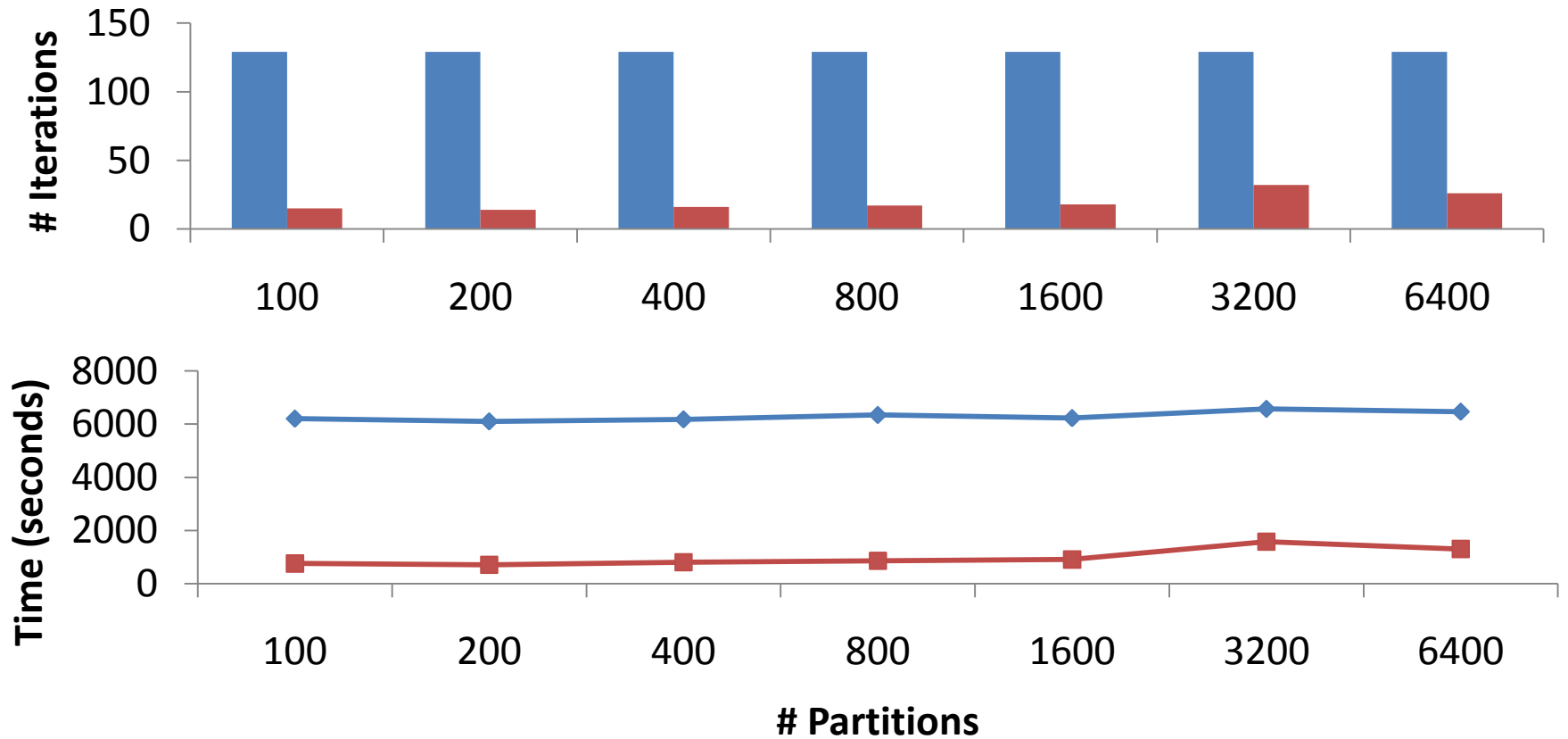


# Performance: Graph B



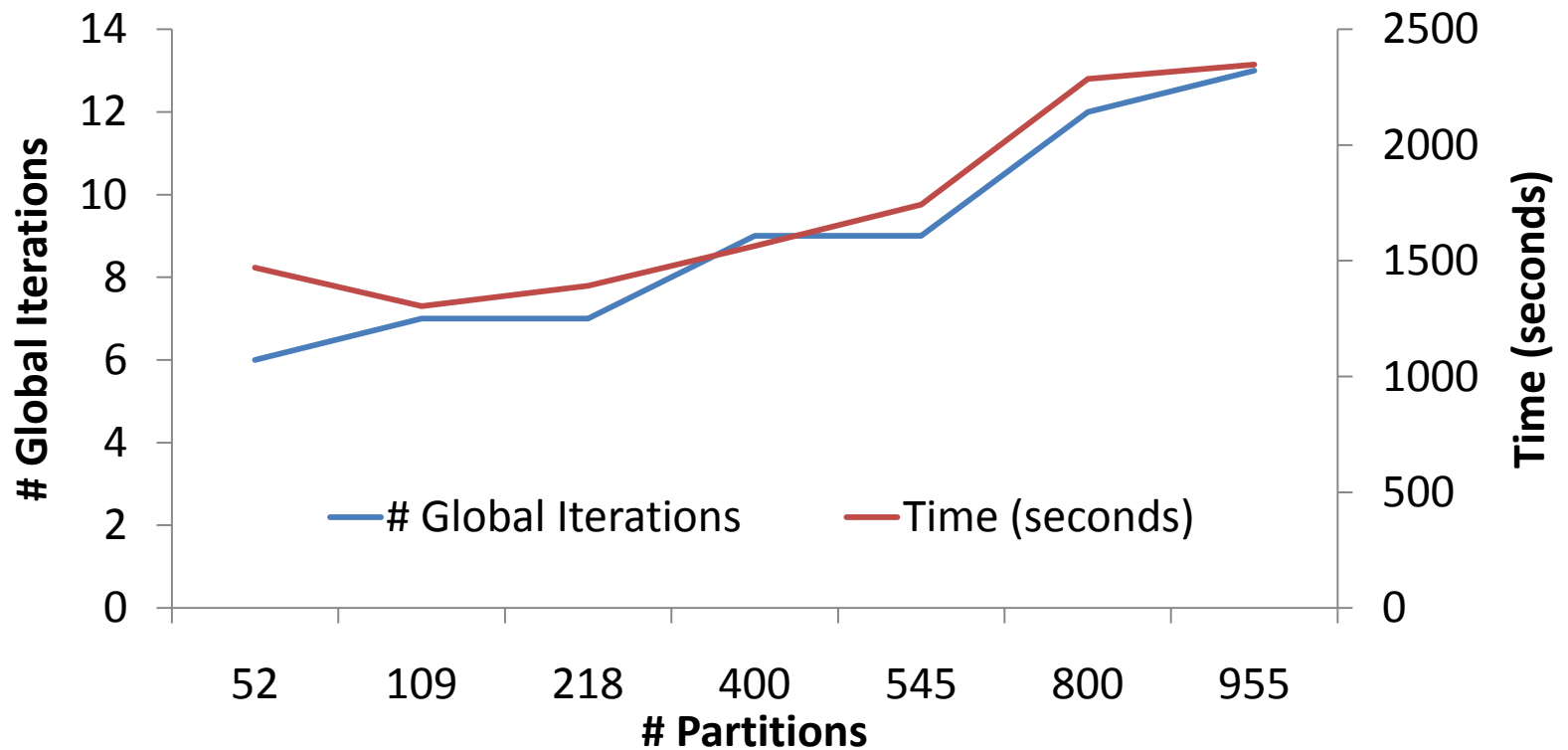
# Single Source Shortest Path

■ General SSSP ■ Relaxed SSSP

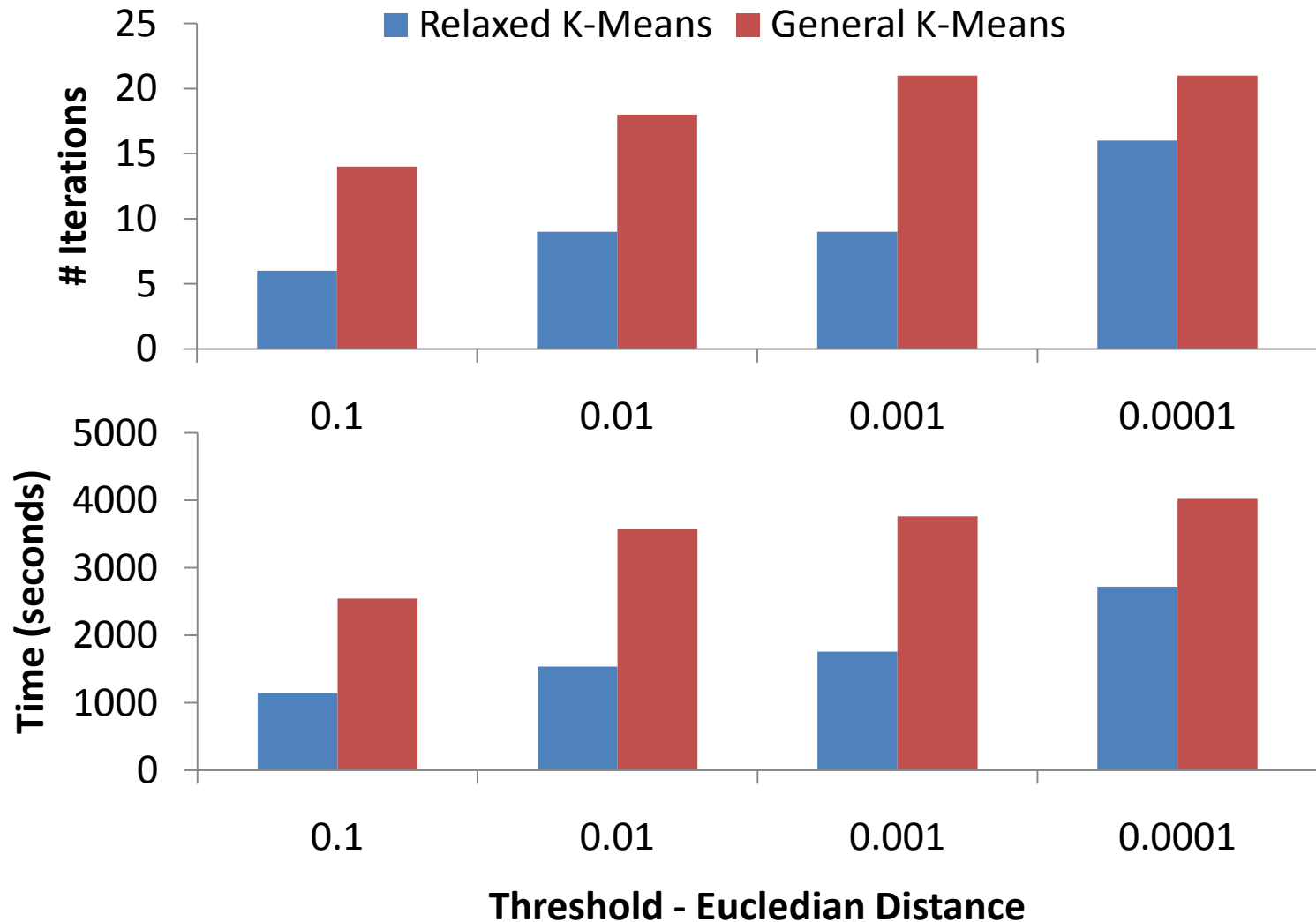


# K-Means

- On US Census data
- General K-Means: 18 iterations
- Threshold: 0.01 for convergence



# K-Means: Threshold



# Ongoing and Future Work

- Continuation
  - Implement **truly hierarchical** relaxed synchronization
  - **Automatically generate** local map and reduce functions from the original MapReduce program
- Other optimizations
  - Speculative Parallelism

# Conclusions

- MapReduce has limited applicability outside data-parallel applications
- We propose relaxed synchronization semantics for iterative MapReduce to allow asynchrony
- Semantics evaluation
  - API implementation using hashtables for local storage
  - PageRank, Single Source Shortest Path, and K-Means
- Upto 8x speed-ups achieved

# Thank You!