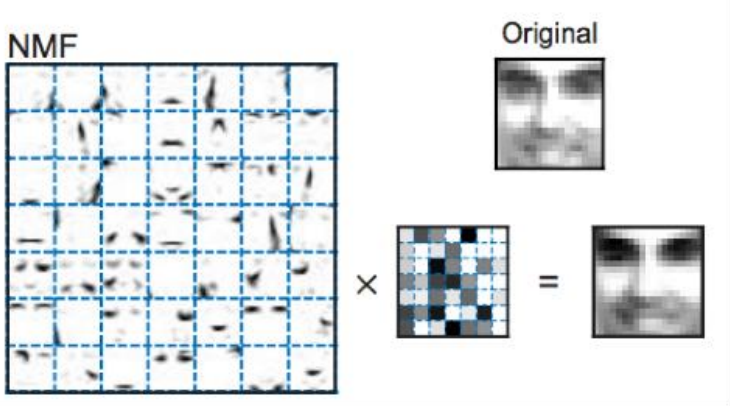
Scalable Nonnegative Matrix Factorization with Block-wise Updates

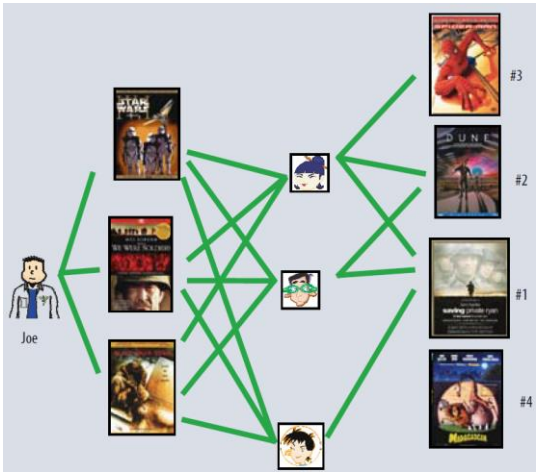
Jiangtao Yin, Lixin Gao, Zhongfei (Mark) Zhang

University of Massachusetts Amherst
Binghamton University

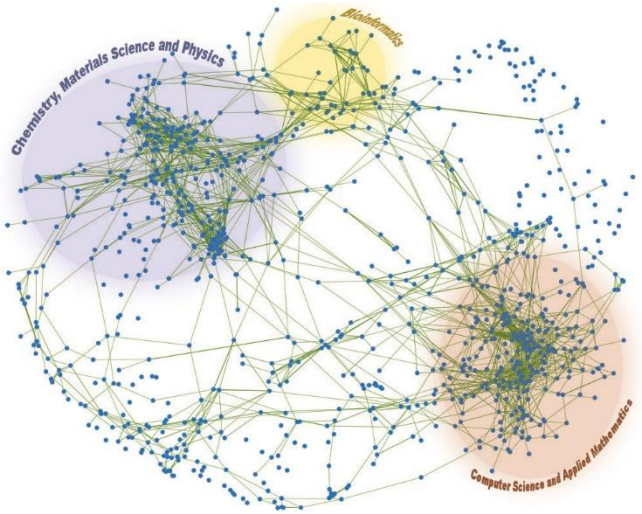
Applications of NMF



Computer Vision



Recommendation System

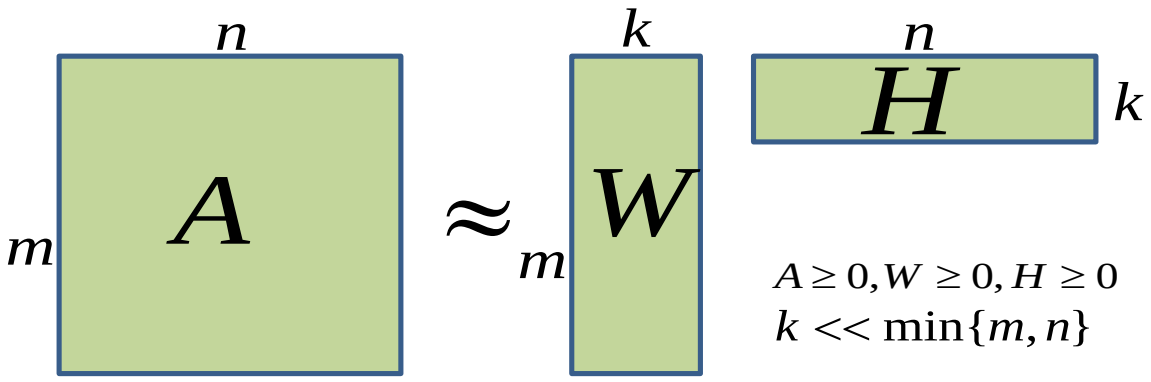


Document Clustering

Link Prediction

Nonnegative Matrix Factorization (NMF)

- Effective approach to uncover latent relationships in nonnegative matrices
  - Matrix A is factorized into two matrices W and H
  - Minimize a loss function: e.g., $L(W,H) = \|A - WH\|_F^2$



How to scale NMF to million-by-million matrices?

Outline

- Motivation
- Distributed NMF
- Implementation
- Evaluation

NMF Algorithm

- Update W and H alternatively
  - Initialize W, H with nonnegative W^0, H^0, t←0
  - Repeat until a convergence criterion is satisfied:
    - Find H^{t+1}: $L(A, W^t H^{t+1}) \leq L(A, W^t H^t)$
    - Find W^{t+1}: $L(A, W^{t+1} H^{t+1}) \leq L(A, W^t H^{t+1})$

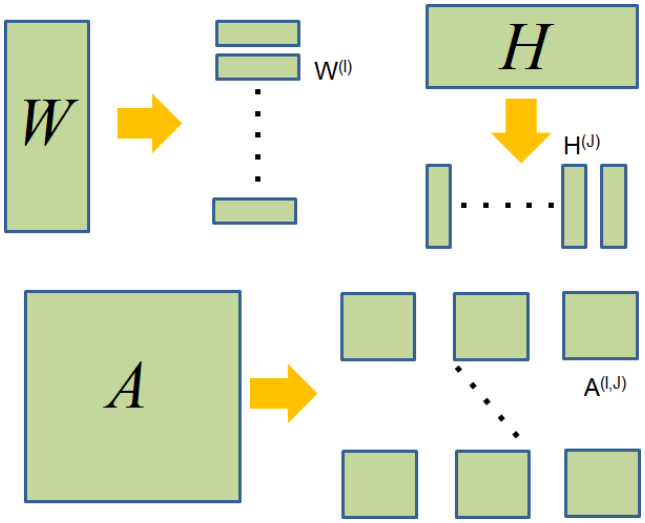
- Multiplicative update approach [Lee & Seung, 2000]

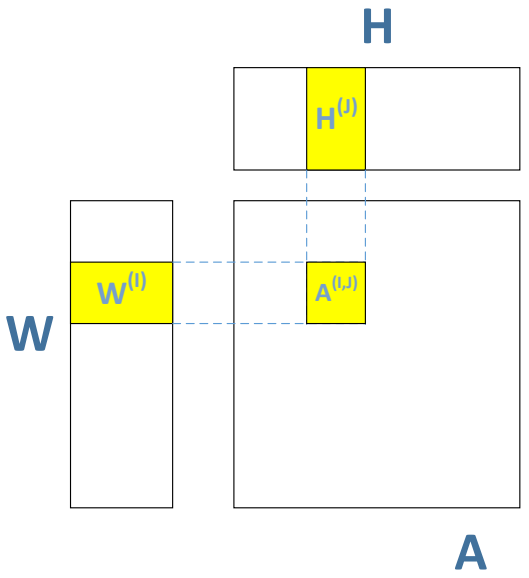
$$H = H * \frac{W^T A}{W^T W H} \qquad W = W * \frac{A H^T}{W H H^T}$$

Challenging to implement in distributed environment

Distributed NMF

- How to efficiently scale NMF in a distributed environment?
- Partition data across machines
  - Split W and H into blocks along short dimension to maximize parallelism
  - Partition *A* into corresponding blocks





Can we update on blocks (instead of the entire matrix)?

Decomposable Loss Functions

- The loss function typically is the sum of losses for each element in the matrix
  - For example, square of Euclidean distance,

$$L(A, WH) = \sum_{(i,j)} (A_{ij} - [WH]_{ij})^2$$

- Represent the loss function in terms of blocks

$$L(A, WH) = \sum_{I} \sum_{J} L(A^{(I,J)}, W^{(I)} H^{(J)})$$

Rewrite Loss Function

- Rewrite as the sum of local loss functions

$$L(A, WH) = \sum_I F_I = \sum_J G_J$$

$$F_I = \sum\nolimits_J L(A^{(I,J)}, W^{(I)} H^{(J)})$$

$$G_J = \sum\nolimits_I L(A^{(I,J)}, W^{(I)} H^{(J)})$$

- Update $W^{(I)}$ via minimizing F_I (by fixing H)

Flexible: we can update one block at a time instead of the whole factor matrix

Block-wise Updates

- Blocks of one factor matrix can be updated independently (by fixing the other factor matrix)

$$H^{(J)} = H^{(J)} * \frac{[\sum_I (W^{(I)})^T A^{(I,J)}]}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]}$$

$$W^{(I)} = W^{(I)} * \frac{[\sum_J A^{(I,J)} (H^{(J)})^T]}{[\sum_J W^{(I)} H^{(J)} (H^{(J)})^T]}$$

- New form of update functions
- Each block can be treated as one update unit
- Better suitable for distributed implementation

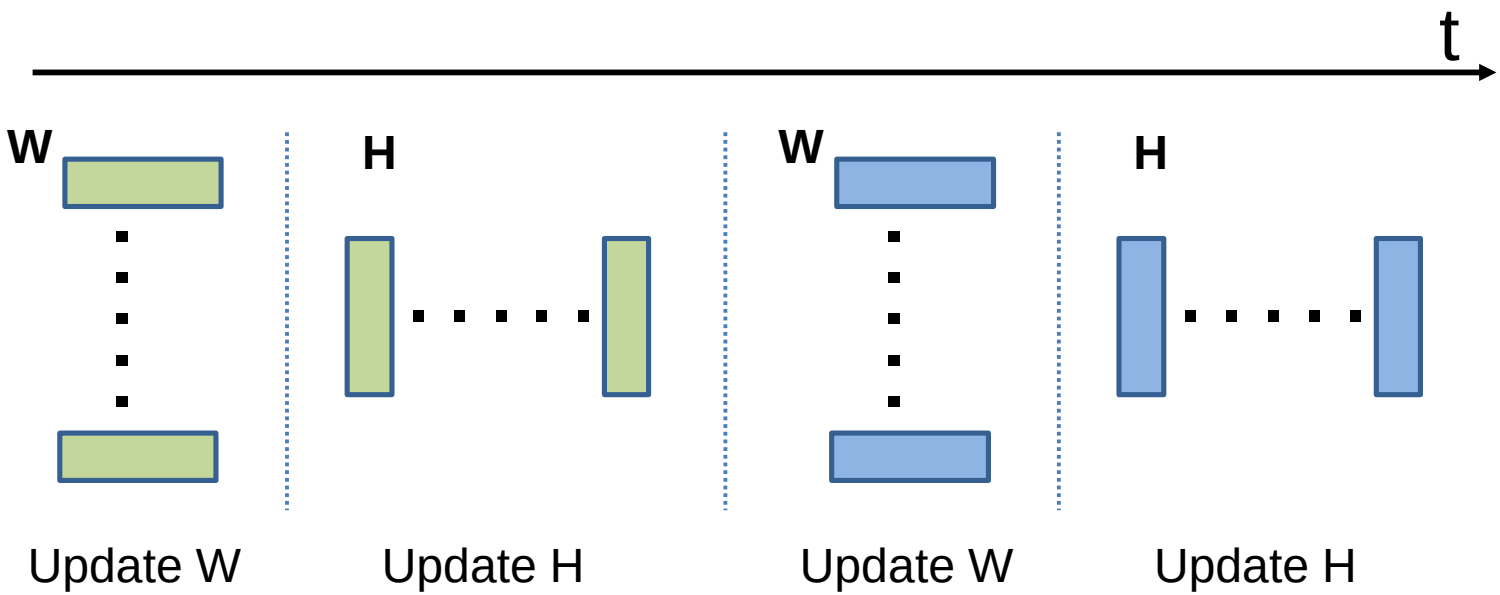
$$H = H * \frac{W^T A}{W^T W H}$$

$$W = W * \frac{A H^T}{W H H^T}$$

- Traditional form of update functions
- The entire matrix is an update unit

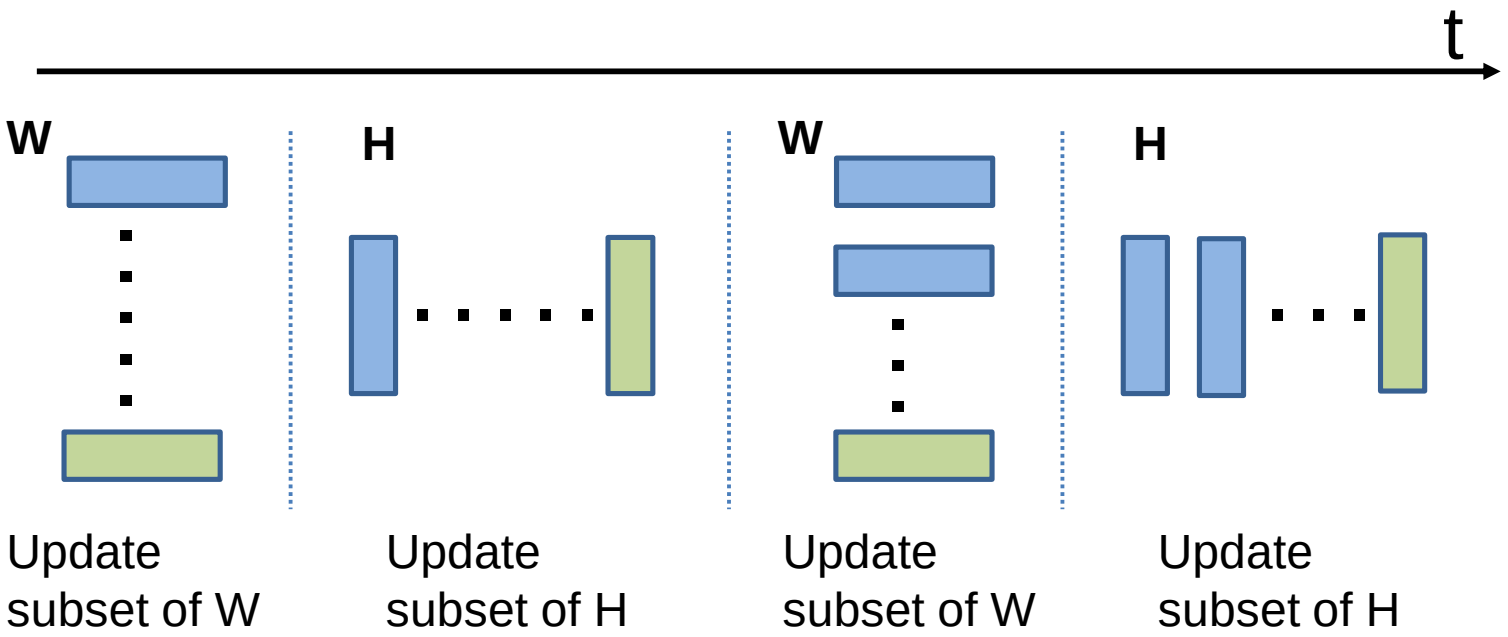
Concurrent Block-wise Updates

- Update all blocks of *W* and then update all blocks of *H*
  - Equivalent to traditional multiplicative update approach



Frequent Block-wise Updates

- Update a subset of W blocks and then update a subset of H blocks
  - Usually need incremental computation



Utilize the most recently updated data whenever possible

Incremental Computation for Block-wise Updates

- Reuse unchanged intermediate result
- Introduce a few auxiliary matrices:

$$X^{(J)} = \sum_I (W^{(I)})^T A^{(I,J)} \qquad X^{(I,J)} = (W^{(I)})^T A^{(I,J)}$$
$$S = \sum_I (W^{(I)})^T W^{(I)} \qquad S^{(I)} = (W^{(I)})^T W^{(I)}$$

$$\Rightarrow \quad H^{(J)} = H^{(J)} * \frac{X^{(J)}}{[SH^{(J)}]}$$

Incremental:

$$X^{(J)} = X^{(J)} + \sum_{I \in C} [(W^{(I)new})^T A^{(I,J)} - X^{(I,J)}]$$
$$S = S + \sum_{I \in C} [(W^{(I)new})^T W^{(I)new} - S^{(I)}]$$

C: subset of updated blocks

Cost grows linearly with the number of blocks updated

Outline

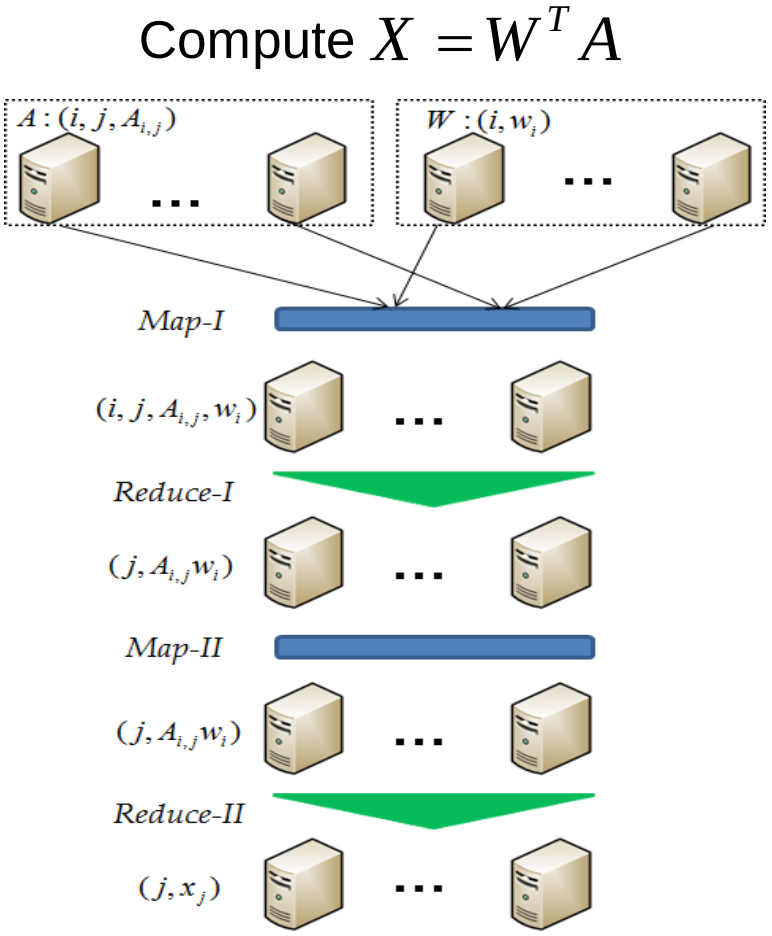
- Motivation
- Distributed NMF
- Implementation
- Evaluation

Existing MapReduce Implementation of Multiplicative Update Approach

- Existing well-known implementation for traditional updates [Liu et al., 2010]
  - Decompose update functions into basic matrix operations, e.g., matrix multiplication
  - Use MapReduce to implement those basic matrix operations
  - Shuffle related rows or columns to the same place (a huge amount of data)
  - One operation may need 2 jobs

$$H = H * \frac{W^T A}{W^T W H}$$

$$H = H * \frac{X}{Y}$$



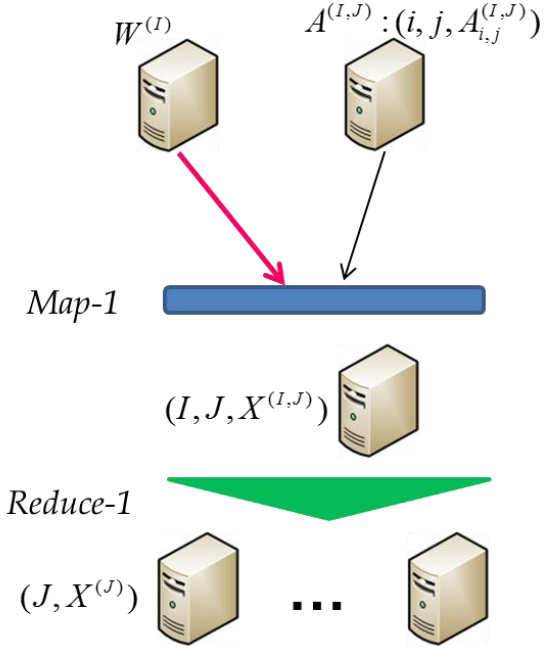
Our MapReduce Implementation of Block-wise Updates

- Computation at block level enables efficient implementation:
  - Decompose update functions into matrix operations that only involve blocks
  - Assign blocks of factor matrices to workers (e.g., mappers), one block to one worker
  - Complete a matrix operation in a worker's memory (blocks of A are not in memory)

$$H^{(J)} = H^{(J)} * \frac{[\sum_I (W^{(I)})^T A^{(I,J)}]}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]}$$

$$H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$$

$$X^{(J)} = \sum_I X^{(I,J)} = \sum_I (W^{(I)})^T A^{(I,J)}$$



Map-1: $X^{(I,J)} = (W^{(I)})^T A^{(I,J)}$

Reduce-1: $X^{(J)} = \sum_I X^{(I,J)}$

Much less data shuffled

Outline

- Motivation
- Distributed NMF
- Implementation
- Evaluation

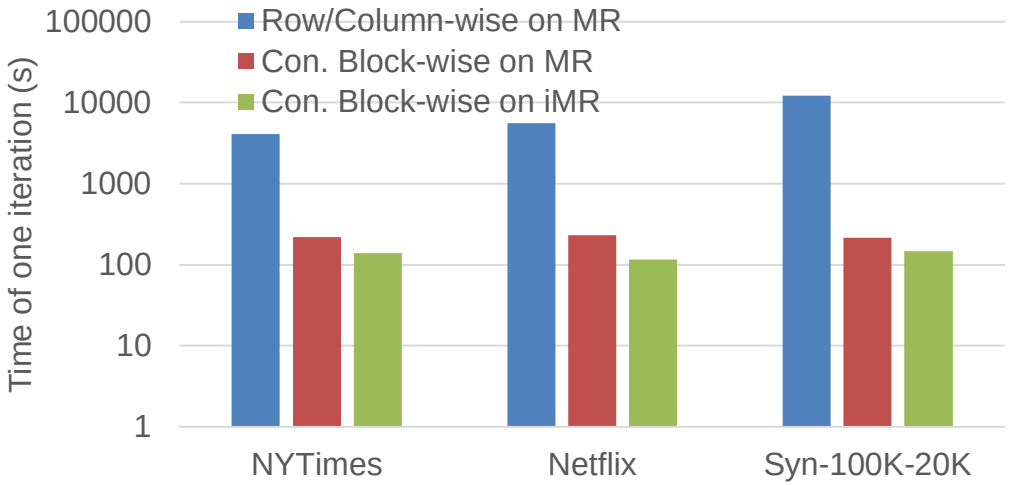
Experiment Setup

- Cluster
  - Local cluster: 4 machines, E8200 dual-core 2.66GHz CPU, 4GB of RAM
  - Amazon EC2 cloud: 100 medium instances

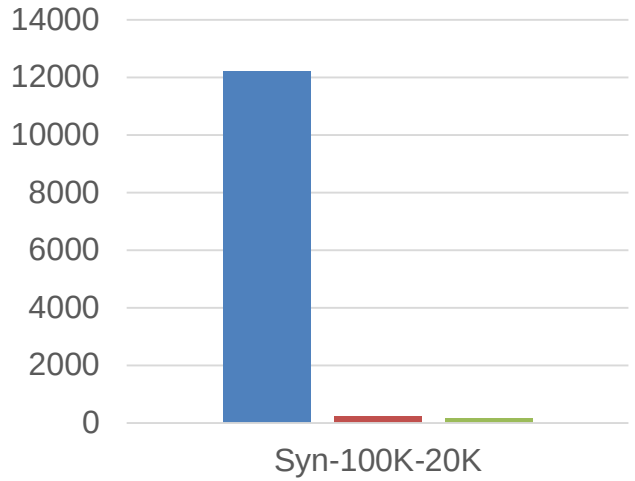
- Data sets

Dataset	# of rows	# of columns	# of nonzero elements
Netflix	480,189	17,770	100M
NYTimes	300,000	102,660	70M
Syn-m-n	m	n	0.1*m*n

Performance of Concurrent Block-wise Updates



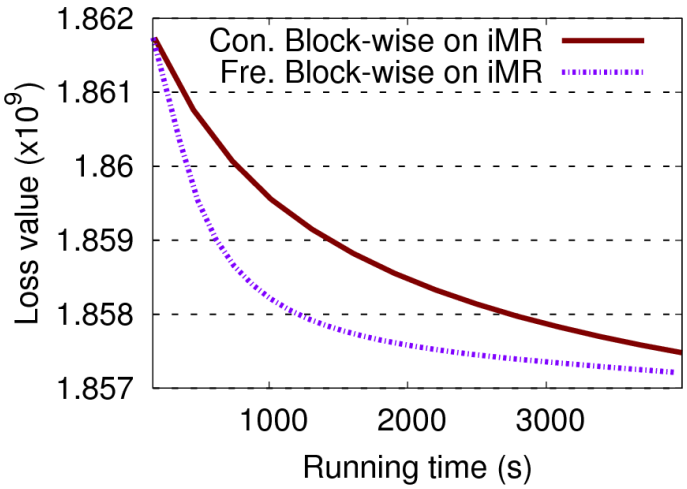
Y-axis in log-scale



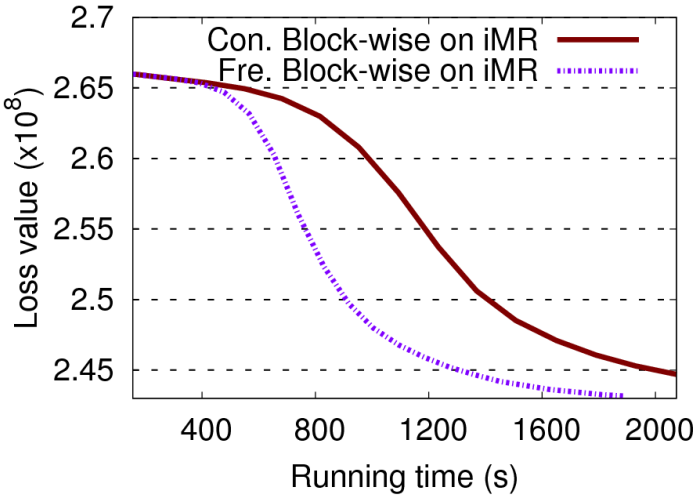
Y-axis in linear-scale

- "Row/Column-wise" denotes the implementation for the traditional updates ([Liu et al., 2010])
- Our implementation (of concurrent block-wise updates) on MapReduce is 19x - 57x faster
- Our implementation on iMapReduce (iterative version of MapReduce) is even faster

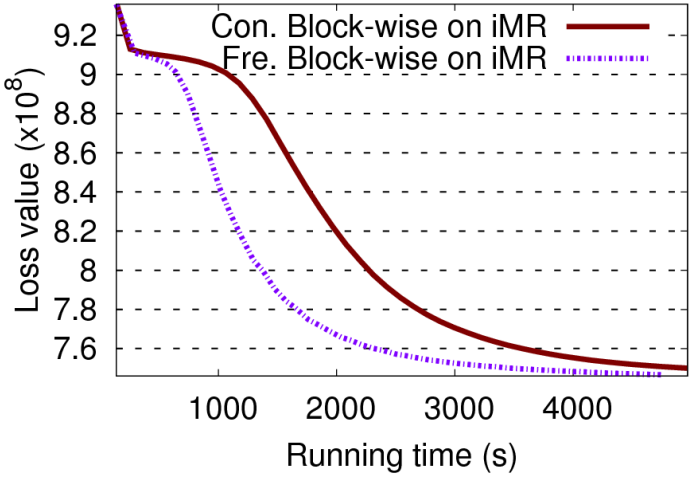
Effect of Frequent Updates



Syn-100k-20k



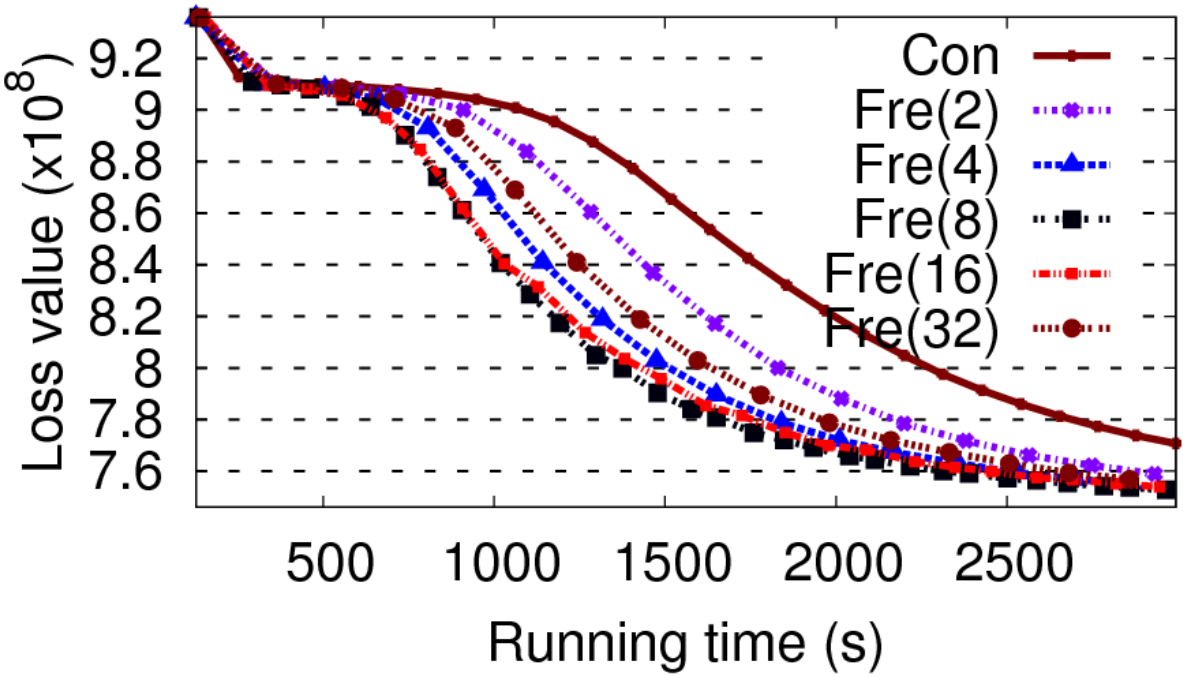
NYTimes



Netflix

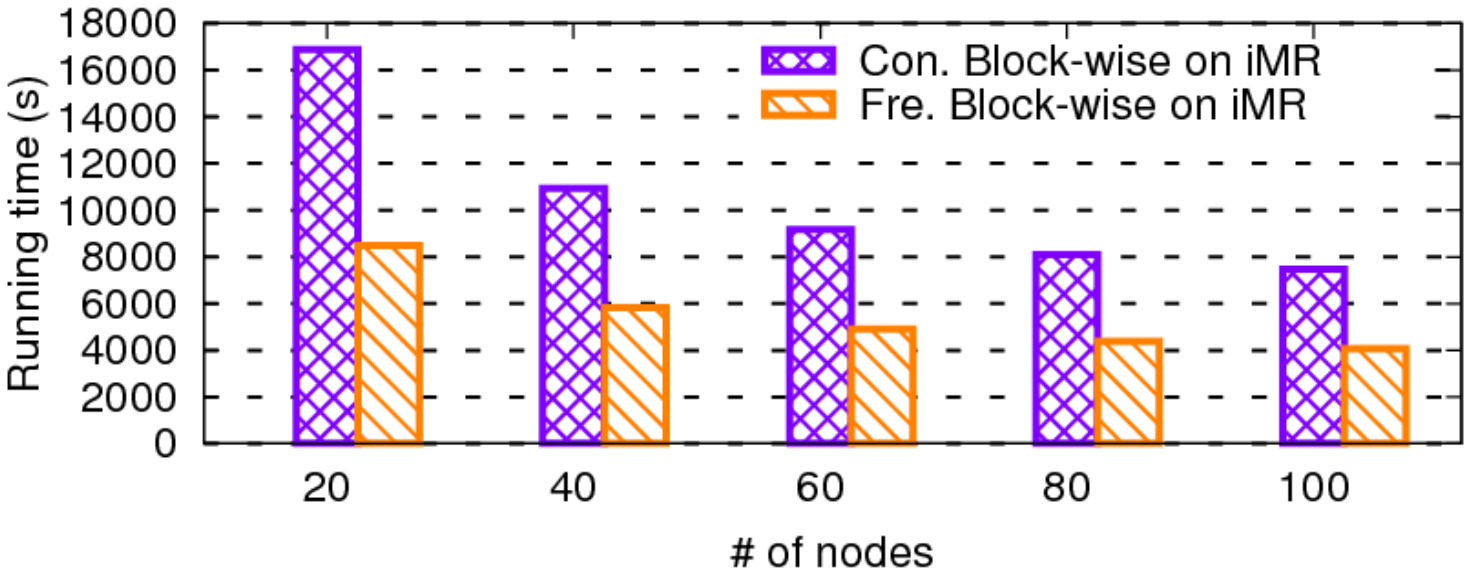
- Frequent block-wise updates always converge faster than concurrent block-wise updates

Determining Update Frequency



- Typically, set the size of the subset as 1/16 to 1/4 of the entire set
- Another way: test several settings, each in a few iterations, and select the best one
  - If a setting is better during the first few iterations, it will continue to be better

Scaling Performance



- Both approaches exhibit good speedups
- Frequent block-wise updates always converge much faster than concurrent block-wise updates

Conclusion

- Propose a new form of updates, block-wise updates, for NMF
  - Enable efficient distributed implementation
  - Support flexible update schemes, e.g., frequent updates
- Present efficient MapReduce-based implementation
- The implementation of block-wise updates achieves up to two orders of magnitude speedup

Questions?

Thank you!