November 2016

Accelerating Iterative Computations for Large-Scale Data Processing

Jiangtao Yin
University of Massachusetts Amherst

ACCELERATING ITERATIVE COMPUTATIONS FOR LARGE-SCALE DATA PROCESSING

A Dissertation Presented

by

JIANGTAO YIN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2016

Electrical and Computer Engineering

ACCELERATING ITERATIVE COMPUTATIONS FOR LARGE-SCALE DATA PROCESSING

A Dissertation Presented

by

JIANGTAO YIN

Approved as to style and content by:

Lixin Gao, Chair

Marco Duarte, Member

David Irwin, Member

Deepak Ganesan, Member

C. V. Hollot, Department Chair
Electrical and Computer Engineering

DEDICATION

To my family

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor, Prof. Lixin Gao, who guided me tirelessly throughout my PhD. Her insight, encouragement, and enthusiasm in research always inspire me. I am also grateful to the rest of my committee members, Prof. Marco Duarte, Prof. David Irwin, and Prof. Deepak Ganesan, for their valuable comments and advices.

I would also like to thank past and current members of our research group, Yong Liao, Yang Song, Samamon Khemmarat, Dong Yin, Yanfeng Zhang, Renjie Zhou, Guoyi Zhao, Xiang Cheng, Jiahui Jin, Meng Shen, Jianzhou Chen, Xiaozhe Shao, Fubao Wu, Tian Zhou, Hao Zhang, and others. They have kindly helped me in all sorts of things and have greatly contributed to my personal growth and the growth of my work.

Last but not least, I offer my deepest gratitude to my family. Their love and support have been and will continue to remain my backbone.

ABSTRACT

ACCELERATING ITERATIVE COMPUTATIONS FOR LARGE-SCALE DATA PROCESSING

SEPTEMBER 2016

JIANGTAO YIN

B.Eng., BEIJING INSTITUTE OF TECHNOLOGY

M.Eng., BEIJING UNIVERSITY OF POSTS AND TELECOMMUNICATIONS

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lixin Gao

Recent advances in sensing, storage, and networking technologies are creating massive amounts of data at an unprecedented scale and pace. Large-scale data processing is commonly leveraged to make sense of these data, which will enable companies, governments, and organizations, to make better decisions and bring convenience to our daily life. However, the massive amount of data involved makes it challenging to perform data processing in a timely manner. On the one hand, huge volumes of data might not even fit into the disk of a single machine. On the other hand, data mining and machine learning algorithms, which are usually involved in large-scale data processing, typically require time-consuming iterative computations. Therefore, it is imperative to efficiently perform iterative computations on large computer clusters or cloud using highly-parallel and shared-nothing distributed systems.

This research aims to explore new forms of iterative computations that reduce unnecessary computations so as to accelerate large-scale data processing in a distributed

environment. We propose the iterative computation transformation for well-known data mining and machine learning algorithms, such as expectation-maximization, nonnegative matrix factorization, belief propagation, and graph algorithms (e.g., PageRank). These algorithms have been used in a wide range of application domains. First, we show how to accelerate expectation-maximization algorithms with frequent updates in a distributed environment. Then, we illustrate the way of efficiently scaling distributed nonnegative matrix factorization with block-wise updates. Next, our approach of scaling distributed belief propagation with prioritized block updates is presented. Last, we illustrate how to efficiently perform distributed incremental computation on evolving graphs.

We will elaborate how to implement these transformed iterative computations on existing distributed programming models such as the MapReduce-based model, as well as develop new scalable and efficient distributed programming models and frameworks when necessary. The goal of these supporting distributed frameworks is to lift the burden of the programmers in specifying transformation of iterative computations and communication mechanisms, and automatically optimize the execution of the computation. Our techniques are evaluated extensively to demonstrate their efficiency. While the techniques we propose are in the context of specific algorithms, they address the challenges commonly faced in many other algorithms.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Recent advances in sensing, storage, and networking technologies, such as smart devices, cloud storage, and mobile networks, have led to massive amounts of data being generated and collected at an unprecedented scale and pace. For example, companies like Google, Facebook, and Amazon, maintain and process petabytes of data, including user interactions, product sales, system logs, and other types of information. The ability to perform timely analytical processing on these data (i.e., large-scale data processing) will enable companies, governments, and organizations, to make better decisions and bring convenience to our daily life. Large-scale data processing typically involves data mining and machine learning algorithms. Despite the advances in data mining and machine learning algorithms, the massive amount of data involved makes it challenging to perform large-scale data processing in a timely manner. On the one hand, huge volumes of data are typically stored in distributed file systems, since they might not even fit into the disk of a single machine. On the other hand, data mining and machine learning algorithms usually require time-consuming iterative computations to achieve the final results. As the volume of data grows and the speed with which new data is generated increases, it is imperative to perform iterative computations on large computer clusters or cloud using highly-parallel and shared-nothing distributed systems.

MapReduce [22] has been proposed for processing large amounts of data in a cluster of machines or the cloud environment. Since its introduction, MapReduce and its open source implementation, Hadoop [2], have become extremely popular. It pro-

vides a simple programming model, distributed execution, distributed data storage, and fault tolerance. This enables programmers with no experience on distributed systems to exploit a large cluster of commodity machines (a shared-nothing architecture, with each machine having its own storage, CPU, and memory) to perform large-scale data processing. However, while MapReduce is highly effective in hiding the complexity of distributed processing and fault tolerance of the system, it is mainly designed for "embarrassingly parallel tasks".

In this work, we challenge the conventional wisdom that distributed iterative computations have to be performed with traditional update functions. We aim to identify new forms of update functions that reduce unnecessary computations so as to accelerate iterative computations in a distributed environment. We propose the update function transformation for well-known data mining and machine learning algorithms such as expectation-maximization, nonnegative matrix factorization, belief propagation, and graph algorithms (e.g., PageRank). First, we show how to accelerate expectation-maximization algorithms with frequent updates in a distributed environment [96]. Then, we illustrate the way of efficiently scaling distributed nonnegative matrix factorization with block-wise updates [95]. Next, our approach of scaling distributed belief propagation with prioritized block updates is presented [93]. Last, we illustrate how to efficiently perform distributed incremental computation on evolving graphs [94]. We will show how to implement these transformed update functions on existing programming models such as the MapReduce-based model, as well as develop new scalable and efficient distributed programming models and frameworks when necessary. The goal of these supporting distributed frameworks is to lift the burden of the programmers in specifying transformation of update functions and communication mechanisms, and automatically optimize the execution of the computation. In the following, we provide backgrounds and discuss challenges in transforming up-

date functions of iterative computations and in implementing them in a distributed environment.

1.1 Frequent Updates for Expectation-Maximization

Expectation-maximization (EM) [23] is one of the most popular approaches in discovering knowledge from a large collection of datasets, and has many applications such as image understanding, document classification, and genome data analysis. It is an iterative approach that alternates between performing an expectation step (E-step) and a maximization step (M-step). For instance, a data clustering algorithm (e.g., k-means) can be seen as an example of the EM approach. Such an algorithm groups similar data points into the same cluster. In the E-step, the assignment of points to clusters is performed according to the current information of the clusters (e.g., the current centroid). In the M-step, the resulted assignment is used to further update the information of the clusters. Such an iterative refinement process continues with many iterations until the clustering algorithm converges.

Due to its popularity, many methods for accelerating EM algorithms have been proposed. Some of them [70, 85] transform original update functions by performing a partial E-step. Such a partial E-step selects only a subset of data points for computing the distribution. The advantage of the partial E-step is that it allows the M-step to be performed more frequently, so that the algorithm can leverage more up-to-date parameters to process data points and potentially accelerates convergence. Despite the fact that the EM algorithm with frequent updates has the potential to speedup convergence, parallelizing it can be challenging. Although computing the distribution can be performed concurrently, parameters such as centroids of clusters are global parameters. Updating these global parameters has to be performed in a centralized location and all workers have to be synchronized. Synchronization in a distributed

environment may result in considerable overhead. Therefore, we have to control the frequency of parameter update to obtain a good performance.

In this research, we propose two approaches to parallelize the EM algorithm with frequent updates in a distributed environment: partial concurrent and subrange concurrent. In the partial concurrent approach, each E-step processes only a block of data points. The size of a block controls the frequency of parameter update. In the subrange concurrent approach, each E-step computes the distribution in a subrange instead of the whole range. The subrange size can determine the frequency of parameter update. We control the parameter update frequency by setting the block/subrange size, and provide strategies to determine the optimal values. Additionally, both approaches can scale to any number of workers/processors.

We design and implement a distributed framework, FreEM, for implementing the EM algorithm with frequent updates based on the two proposed approaches. FreEM eases the process of programming EM algorithms in a distributed environment. Programmers only need to specify the E-step and the M-step. The detailed mechanisms of distributed computation are handled automatically. As a result, it facilitates the process of implementing EM algorithms and accelerates the algorithms through frequent updates.

1.2 Block-wise Updates for Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) [49] factorizes an original matrix into two low-rank factor matrices by minimizing a loss function that measures the discrepancy between the original matrix and the product of the two factor matrices. It has been applied with great success to many applications, including genome data analysis [16], text mining [71], recommendation systems [44], and social network analysis [67, 87]. For example, in the setting of recommender systems, matrix rows can be used to

represent users, and columns represent items (e.g., movies). Then entries are ratings provided by users for items. NMF is an effective tool for analyzing such dyadic data in order to discover the interactions between users and items.

NMF algorithms typically use update functions to iteratively and alternately refine factor matrices. Many practitioners have to deal with NMF on massive datasets. For example, recommendation systems in web services such as Netflix have been dealing with NMF on web-scale dyadic datasets, which involve millions of users, millions of movies, and billions of ratings. For such web-scale matrices, it is desirable to leverage a cluster of machines to speed up the factorization. Prior approaches (e.g., [57]) of handling NMF on MapReduce usually select an existing NMF algorithm and then focus on implementing matrix operations.

In this research, we present a new form of factor matrix update functions. This new form operates on blocks of matrices. In order to support the new form, we partition the factor matrices into blocks along the short dimension and split the original matrix into corresponding blocks. The new form of update functions allows us to update distinct blocks independently and simultaneously when updating a factor matrix. As a result, it also facilitates a distributed implementation. Different blocks of one factor matrix can be updated in parallel.

Moreover, under the new form of update functions, we can update only a subset of its blocks when we update a factor matrix, and the number of blocks in the subset can be adjusted. The only requirement is that when one factor matrix is being updated, the other one has to be fixed. For instance, we can update one block of a factor matrix and then immediately update all blocks of the other factor matrix. Frequent block-wise updates aim to utilize the most recently updated data whenever possible. As a result, frequent block-wise updates are more efficient than their traditional concurrent counterparts, which update all blocks of either factor matrix alternately.

1.3 Prioritized Block Updates for Belief Propagation

From forecasting the chance of rain to predicting the traffic on a road, probabilistic reasoning has been used widely. The probabilistic graphical model is one of the most influential techniques for probabilistic reasoning and has been used in a wide range of application domains [35, 43, 86, 99, 112]. Inference in these models, including marginalization and maximum a posteriori estimation, forms the basis of many statistical methods in knowledge management. Loopy belief propagation (BP) and its variants [39, 72, 82, 92] are popular message passing methods for performing approximate inference in these models.

It has been shown that the schedule for updating messages can make a huge difference to the running time of BP algorithms. Specifically, dynamic scheduling schemes, which transform original update functions by dynamically adjusting the order of updating messages can significantly speedup BP algorithms [26, 29, 30, 83]. Although dynamic scheduling schemes have potential to speedup BP algorithms, existing ones cannot fully utilize the potential. Most of them typically select one message for updating each time, e.g., the message with the highest priority value. As a result, many operations need to be performed so as to select next message. That is, the cost of realizing such a dynamic scheduling scheme is high.

In this research, we propose to select a set of messages instead of a single one to update at a time. Hence, the amortized cost of selecting one message is low. Moreover, a novel priority is leveraged to determine which messages are selected. The priority allows messages that are more useful towards achieving convergence to be selected, and the computation cost of the priority is low. To this end, we transform original updates again by introducing an efficient incremental update mechanism, which propagates only the changes of original messages. The change of a message is efficiently computed using the changes of original incoming messages.

As the probabilistic graphical models are applied to model large and complex applications, such as image restoration for high-resolution images, it is desirable to leverage the parallelism of a cluster of machines to reduce the inference time. Therefore, we design and implement a distributed framework, *Prom*, which facilitates the implementation of BP and other graph algorithms in a distributed environment. Prom uses the proposed scheduling scheme as its built-in scheduling and supports the incremental-update approach. We evaluate two BP algorithms, the sum-product algorithm and the max-product algorithm on Prom to show the performance of our scheduling scheme.

1.4 Incremental Computation for Graph Algorithms

Since graphs can capture complex dependencies and interactions between objects, graph algorithms have become an essential component in many real-world applications [6, 8, 15, 27, 34, 59, 81], including business intelligence, social sciences, data mining, and online machine learning. An essential property of graphs is that they are often dynamic. As new data and/or updates are being collected (or produced), the graph will evolve. For example, search engines periodically crawl the web, and the web graph is evolving as web pages and hyper-links are created and/or deleted. Many applications must utilize the up-to-date graph in order to produce results that can reflect the current state. However, rerunning the computation over the entire graph is not efficient, since it discards the work done in earlier runs no matter how little changes have been made.

The dynamic nature of graphs implies that performing incremental computation can improve efficiency dramatically. Incremental computation exploits the fact that only a small portion of the graph has changed. It reuses the result of the prior computation and perform computation only on the part of graph that is affected by the change. Although a number of distributed frameworks have been proposed to sup-

port incremental computation on massive graphs [11, 17, 60, 73], most of them apply synchronous updates to computation. Synchronous updates require that all the update operations in the previous iteration have to complete before any of the update operations in the next iteration can start. Consequently, the synchronization barriers might degrade performance, especially in heterogeneous distributed environments. In order to avoid the high-cost of synchronization barriers, asynchronous updates have been proposed [9]. In asynchronous updates model, a vertex performs the update using the most recent values instead of the values from the previous iteration. Intuitively, we can expect asynchronous updates outperform synchronous updates since more up-to-date values are used and the synchronization barriers are bypassed. However, asynchronous updates might require more communications and perform useless computations (e.g., when no values for a vertex are updated), and thus result in limited performance gain over synchronous updates.

In this research, we provide an approach to efficiently apply asynchronous updates to incremental computation. We first identify what kind of graph algorithms working with incremental computation. We then introduce a new form of the update function of the graph algorithm (i.e., transforming the original update function) to facilitate incremental computation. In order to address the challenge that the change in a small range of the graph may gradually propagate to affect the computation on a large portion of the graph, we present a scheduling scheme to coordinate asynchronous updates. Furthermore, we develop a distributed system to support incremental computation with asynchronous updates.

1.5 Contributions

The goal of this work is to explore new forms of update functions that reduce unnecessary computations so as to improve efficiency of iterative computations in a distributed environment. To this end, we propose the update function transformation

for several well-known data mining and machine learning algorithms and develop distributed frameworks to facilitate the implementation of the transformation.

More specifically, our main contributions are as follows.

- We propose two approaches to parallelize EM algorithms with frequent updates in a distributed environment so as to scale to massive datasets. Furthermore, we design and implement a distributed framework to support the implementation of frequent updates for the EM algorithms. Its efficiency is shown in the context of a wide class of well-known EM applications: k-means clustering, fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model, and Latent Dirichlet Allocation for topic modeling.

- We show that by leveraging a new form of update functions for nonnegative matrix factorization, we can perform local aggregation and fully explore parallelism in a distributed environment. Moreover, under the new form of update functions, we can perform frequent updates, which aim to use the most recently updated data whenever possible. As a result, frequent updates are more efficient than their traditional concurrent counterparts. We evaluate the efficiency provided by our implementation through a series of experiments on a local cluster as well as the Amazon EC2 cloud [1].

- We propose a new scheduling scheme to coordinate message updates for belief propagation. The scheme selects a set of messages to update at a time and leverages a novel priority to determine which messages are selected. An incremental update approach is introduced to accelerate the computation of the priority. Furthermore, we design a distributed framework to facilitate the implementation of BP algorithms. We evaluate the efficiency of the proposed scheduling scheme via extensive experiments.

- We apply asynchronous updates to incremental computation on evolving graphs. Comparing with its synchronous counterpart, asynchronous incremental computation can bypass synchronization barriers and always utilize the most recent values, and thus is more efficient. We develop a distributed framework to facilitate the implementation of graph algorithms with asynchronous incremental computation on massive evolving graphs. We evaluate the proposed asynchronous incremental computation approach via extensive experiments.

The rest of this dissertation is organized as follows. In Chapter 2, we present our technique of accelerating expectation-maximization algorithms with frequent updates in a distributed environment. Chapter 3 presents our approach of efficiently scaling nonnegative matrix factorization with block-wise updates. In Chapter 4, we illustrate our way of applying prioritized block updates to distributed belief propagation. Chapter 5 presents our approach of applying asynchronous incremental computation on evolving graphs. In Chapter 6, we conclude this dissertation.

CHAPTER 2

ACCELERATING DISTRIBUTED EXPECTATION-MAXIMIZATION ALGORITHMS WITH FREQUENT UPDATES

2.1 Introduction

Discovering knowledge from a large collection of datasets is one of the most fundamental problems in many applications, such as image understanding, document classification, and genome data analysis. Expectation-Maximization (EM) [23] is one of the most popular approaches in these applications [56, 89, 91, 98, 108]. It estimates parameters for hidden variables by maximizing the likelihood. EM is an iterative approach that alternates between performing an Expectation step (E-step), which computes the distribution for the hidden variables using the current estimates for the parameters, and a Maximization step (M-step), which re-estimates parameters to be those maximizing the likelihood found in the E-step.

Due to its popularity, many methods for accelerating EM algorithms have been proposed. Some of them [70, 85] show that a partial E-step may accelerate convergence. Such a partial E-step selects only a subset of data points for computing the distribution. The advantage of the partial E-step is that it allows the M-step to be performed more frequently, so that the algorithm can leverage more up-to-date parameters to process data points and to potentially accelerate convergence. Intuitively, updating the parameters frequently might incur additional overhead. However, the parameters typically depend on statistics of datasets that can be computed incrementally. That is, the cost of computing statistics grows linearly with the number of data points whose statistics have been changed in the E-step. As a result, performing

frequent updates on the parameters does not necessarily introduce additional cost. We refer to the EM algorithm that updates the parameters frequently as *the EM algorithm with frequent updates*. In contrast, the traditional EM algorithm, which computes the distribution for all data points and then updates the parameters, is referred to as *the EM algorithm with concurrent updates*.

Despite the fact that the EM algorithm with frequent updates has the potential to speedup convergence, parallelizing it can be challenging. Although computing the distribution and updating statistics can be performed concurrently, parameters such as centroids of clusters are global parameters. Updating these global parameters has to be performed in a centralized location and all workers have to be synchronized. Synchronization in a distributed environment may incur considerable overhead. Therefore, we have to control the frequency of parameter update to achieve a good performance.

In this chapter, we propose two approaches to parallelize the EM algorithm with frequent updates in a distributed environment: partial concurrent and subrange concurrent. In the *partial concurrent* approach, each E-step processes only a block of data points. The size of a block controls the frequency of parameter update. In the *subrange concurrent* approach, each E-step computes the distribution in a subrange of hidden variables instead of the whole range. The subrange size can determine the frequency of parameter update. We prove that both approaches maintain the convergence properties of the EM algorithms. We control the parameter update frequency by setting the block/subrange size, and provide strategies to determine the optimal values. Additionally, both approaches can scale to any number of workers/processors.

We design and implement a distributed framework, FreEM, for implementing the EM algorithm with frequent updates based on the two proposed approaches. FreEM eases the process of programming EM algorithms in a distributed environment. Programmers only need to specify the E-step and the M-step. The detailed mechanisms,

such as data distribution, communication among workers, and frequency of M-step, are all handled automatically. As a result, it facilitates the process of implementing EM algorithms and accelerates the algorithms through frequent updates. We evaluate FreEM in the context of a wider class of well-known EM applications: k-means clustering, fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model, and Latent Dirichlet Allocation for topic modeling. Our results show that the EM algorithm with frequent updates can run much faster than that with traditional concurrent updates. In addition, FreEM is more efficient than Hadoop [2], an open source implementation of the popular distributed framework MapReduce [22], in supporting the EM algorithms.

The rest of this chapter is organized as follows. Section 2.2 describes the EM algorithm with frequent updates. Section 2.3 exemplifies frequent updates through four EM applications. Section 2.4 presents our approaches to parallelize the EM algorithm with frequent updates. In Section 2.5, we present the design, implementation and API of FreEM. Section 2.6 is devoted to the evaluation results. Finally, we discuss related work in Section 2.7 and conclude this chapter in Section 2.8.

2.2 EM Algorithms

In a statistical model, suppose that we have observed the value of one random variable, X, which results from a parameterized family, $P(X|\theta)$. The value of another variable, Z, is hidden. Based on the observed data, we wish to find θ such that $P(X|\theta)$ is the maximum. In order to estimate θ, it is typical to introduce the log likelihood function: $L(\theta) = \log P(X|\theta)$. Suppose the data consists of n independent data points $\{x_1, ..., x_n\}$, and thereby the hidden variable can be decomposed as $\{Z_1, Z_2, ..., Z_n\}$. Then, $L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta)$. We assume that Z has a finite range for simplicity, but the result can be generalized. Thus, the probability $P(x_i|\theta)$ can be written in terms of possible value (z_i) of the hidden variable Z_i as: $P(x_i|\theta) = \sum_{z_i} P(x_i, z_i|\theta)$.

When it is hard to maximize $L(\theta)$ directly, an EM algorithm is usually used to maximize $L(\theta)$ iteratively.

The EM algorithm leverages an iterative process to maximize $L(\theta)$. Each iteration consists of an E-step and a M-step. The E-step estimates the distribution of hidden variables, given the data points and the current estimates of the parameters. The M-step updates the parameters to be those maximizing the likelihood found in the E-step.

2.2.1 The EM Algorithm with Concurrent Updates

The EM algorithm with concurrent updates computes the distribution for all data points in its E-step. Formally, let Q_i be some distribution over z_i ($\sum_{z_i} Q_i(z_i) = 1$, $Q_i(z_i) \geq 0$). Such an EM algorithm starts with some initial guess at the parameters $\theta^{(0)}$, and then seeks to maximize $L(\theta)$ by iteratively applying the following two steps:

E-step: For each $x_i \in X$, set $Q_i(z_i) = P(z_i|x_i, \theta^{(t-1)})$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

Here, the expectation E_{Q_i} is taken with respect to the distribution $Q_i(\cdot)$ over the range of Z in the E-step.

2.2.2 The EM Algorithm with Frequent updates

The EM algorithm with frequent updates attempts to accelerate the convergence by frequently updating the parameters. The intuition behind it is that the algorithm can leverage more up-to-date parameters to process data points and to potentially speedup convergence. However, updating parameters frequently may incur significant overhead if the update is done in the original way. In order to conquer this obstruction, we introduce a way of updating parameters incrementally. In the EM algorithm, the distribution influences the likelihood of the parameters via some sufficient statistics. The statistics is usually the summation over the statistics on each individual data point, and a summation can be incrementally updated (in Section 2.3, we will

illustrate what such statistics is and how to incrementally update the statistics for each individual algorithm). As a result, the cost of computing the sufficient statistics grows linearly with the number of data points whose statistics have been changed in the E-step. Therefore, performing frequent updates on the parameters does not necessarily introduce additional cost of computing statistics. However, it will incur extra overhead of deriving the parameters from the statistics. If the overhead is large, it is reasonable to compute the distribution for a subset of data points (or compute the distribution in a subrange of the hidden variable) and then update the parameters.

Updating the parameters frequently in the EM algorithm can be achieved by two approaches. One is *update by block*, which partition data points into mutually disjoint blocks and iterates through the blocks in a cyclic way. Each iteration processes a block of data points in the E-step and then perform the M-step immediately to update the parameters. Its E-step can utilize the up-to-date parameters to process another block of data points. Obviously, when selecting the whole set of data points as a block, the EM algorithm with update by block is actually the EM algorithm with concurrent updates. One iteration of the algorithm can be described as following:

E-step: Pick a block of data points, B_m ($B_m \subseteq X$), and for each $x_i \in B_m$,

Set $Q_i^{(t)}(z_i) = P(z_i|x_i, \theta^{(t-1)})$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

The other one is *update by subrange*, which recomputes the distribution over a subrange of the hidden variable and then updates the parameters. Its E-step can leverage the up-to-date parameters to recompute the distribution over another subrange. The EM algorithm with update by subrange starts with some initial guess at the parameters $\theta^{(0)}$ and some guess at the distribution $Q_i^{(0)}$, and then seeks to maximize $L(\theta)$ by iteratively applying the following two steps:

E-step: Select a subrange of Z, R_{sub}, for each $x_i \in X$,

Let $C_{R_{sub}} = \sum_{z_i \in R_{sub}} Q_i^{(t-1)}(z_i)$;

Set $Q_i^{(t)}(z_i) = P(z_i|x_i, \theta^{(t-1)}) * C_{R_{sub}}$.

M-step: Set $\theta^{(t)}$ to be the θ that maximizes $\sum_{i=1}^{n} E_{Q_i}[\log P(x_i, z_i|\theta)]$.

We can also combine the two approaches to achieve updating the parameters frequently. Such a combined version selects a subrange of Z and computes the distribution for a block of data points under the subrange in its E-step, and then performs the M-step to update the parameters. Obviously, either approach is a special case of the combined version. Furthermore, even the combined version maintains the convergence properties of the EM algorithm.

For proving the convergence of the EM algorithm with frequent updates, we first consider the following derivation:

$$\begin{aligned} L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta) &= \sum_{i=1}^{n} \log \sum_{z_i} Q_i(z_i) \frac{P(x_i, z_i|\theta)}{Q_i(z_i)} \\ &\geq \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}. \end{aligned}$$

The last step of this derivation is given by Jensen's inequality. When $Q_i(z_i) = P(z_i|x_i, \theta)$ for any i, the last step of the derivation holds with equality. Let

$$J(Q, \theta) = \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)},$$

then we have $L(\theta) \geq J(Q, \theta)$. We assume that $P(x_i, z_i|\theta)$ is a continuous function of θ. We can show that if the local maximum of $J(Q, \theta)$ occurs at Q^* and θ^*, the local maximum of $L(\theta)$ occurs at θ^* as well. Hence, if a variant of the EM algorithm gradually increase $J(Q, \theta)$, it will converge to a local maximum (or a saddle point) of $L(\theta)$. For simplicity, we ignore the possibility that it converges to a saddle point. Next, we will prove that each iteration of the EM algorithm with frequent updates either improves $J(Q, \theta)$ or leaves it unchanged, and thus it converges to a local maximum of $L(\theta)$.

Lemma 2.2.1. *Given a fixed value of θ, for each i, there is a unique distribution, $Q_i(\cdot)$, that maximizes $J(Q, \theta)$, achieved by $Q_i(z_i) = P(z_i|x_i, \theta)$. Moreover, the $Q_i(z_i)$ varies continuously with θ.*

Proof. We need to show that for any i, with respect to $Q_i(\cdot)$, $Q_i(z_i) = P(z_i|x_i, \theta)$ maximize $\sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}$. We know $\sum_{z_i} Q_i(z_i) = 1$. Therefore, the maximum can be found using a Lagrange multiplier. At such a maximum, we will have $Q_i(z_i) \propto P(x_i, z_i|\theta)$. Note that $\sum_{z_i} Q_i(z_i) = 1$. We have the unique solution $Q_i(z_i) = \frac{P(x_i, z_i|\theta)}{\sum_{z_i} P(x_i, z_i|\theta)} = P(z_i|x_i, \theta)$. Consequently, given a fixed value of θ, for each i, if $Q_i(z_i) = P(z_i|x_i, \theta)$, $J(Q, \theta)$ is maximized. Since $P(z_i|x_i, \theta)$ varies continuously with θ, $Q_i(z_i)$ varies continuously with θ. □

Lemma 2.2.2. *If $Q_i(z_i) = P(z_i|x_i, \theta)$ for each i, $L(\theta) = J(Q, \theta)$.*

Proof. If $Q_i(z_i) = P(z_i|x_i, \theta)$ (the equality in Jensen's inequality holds), we have

$$L(\theta) = \sum_{i=1}^{n} \log P(x_i|\theta) = \sum_{i=1}^{n} \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)} = J(Q, \theta).$$

□

Lemma 2.2.3. *If $J(Q, \theta)$ has a local maximum at Q^* and θ^*, then a local maximum of $L(\theta)$ occurs at θ^* as well.*

Proof. From Lemmas 2.2.1 and 2.2.2, we see that if $Q_i(z_i) = P(z_i|x_i, \theta)$ for each i, then $L(\theta) = J(Q, \theta)$ for any θ. Therefore, $L(\theta^*) = J(Q^*, \theta^*)$, where Q^* means $Q_i(z_i) = P(z_i|x_i, \theta^*)$ for each i. To show that a local maximum of $L(\theta)$ occurs at θ^*, we need to show that there is no θ' near to θ^* which lets $L(\theta') > L(\theta^*)$. If such a θ' existed, we would have $J(Q', \theta') > J(Q^*, \theta^*)$, where Q' means $Q_i(z_i) = P(z_i|x_i, \theta')$ for each i. From Lemma 2.2.1, we know that Q varies continuously with θ. Therefore, Q' must be near to Q^*. However, it contradicts that $J(Q, \theta)$ has a local maximum at Q^* and θ^*. □

Theorem 2.2.4. *The EM algorithm with frequent updates converges to a local maximum of $L(\theta)$.*

Proof. Let $F_i(x_i, Q_i, \theta) = \sum_{z_i} Q_i(z_i) \log \frac{P(x_i, z_i|\theta)}{Q_i(z_i)}$, then $J(Q, \theta) = \sum_{i=1}^{n} F_i(x_i, Q_i, \theta)$. In the E-step of the EM algorithm with frequent updates, we change the value of $F_i(x_i, Q_i, \theta)$ for a subset of data points (e.g., S_m) through changing $Q_i(\cdot)$. If we can show that $F_e(x_e, Q_e^{(t)}, \theta) \geq F_e(x_e, Q_e^{(t-1)}, \theta)$ for any $x_e \in S_m$, then we prove that the E-step increases $J(Q, \theta)$. Assume we aim to maximize $\sum_{z_e \in B} Q_e(z_e) \log \frac{P(x_e, z_e|\theta)}{Q_e(z_e)}$ with respect to Q_e (where B denotes a subrange of Z). We also know that $\sum_{z_e \in B} Q_e(z_e) = c_B$ (c_B is a constant). The maximum can be found using a Lagrange multiplier (maximize $\sum_{z_e \in B} Q_e(z_e) \log \frac{P(x_e, z_e|\theta)}{Q_e(z_e)}$, subject to $\sum_{z_e \in B} Q_e(z_e) = c_B$). At such a maximum, we will have $Q_e(z_e) \propto P(x_e, z_e|\theta)$ (for $z_e \in c_B$). Note that we also have $\sum_{z_e \in c_B} Q_e(z_e) = c_B$. We have the unique solution $Q_e(z_e) = \frac{P(x_e, z_e|\theta) * c_B}{\sum_{z_e} P(x_e, z_e|\theta)} = P(z_e|x_e, \theta) * c_B$ (for $z_e \in c_B$). Therefore, the E-step increases $F_e(x_e, Q_e, \theta^{(t-1)})$ by setting $Q_e(z_e) = P(z_e|x_e, \theta^{(t-1)})$. Consequently, it increases $J(Q, \theta)$. The M-step of the EM algorithm with frequent updates obtains $\theta^{(t)}$ by maximizing $J(Q, \theta)$. Hence, the M-step increases $J(Q, \theta)$ as well. Since both its E-step and its M-step increase $J(Q, \theta)$, the EM algorithm with frequent updates converges to a local maximum of $J(Q, \theta)$. By combining with Lemma 2.2.3, we know that the EM algorithm with frequent updates converges to a local maximum of $L(\theta)$. □

2.3 Applications of the EM Algorithm

In this section, we describe two categories of applications which the EM algorithm can be applied to, clustering and topic modeling. In the clustering category, we illustrate k-means clustering, Fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model. In the topic modeling category, we discuss variational inference for Latent Dirichlet Allocation. We illustrate how to incrementally compute the statistics and how to derive the parameters from the statistics when applying the

EM algorithm to these applications. By introducing the statistics, the operations of computing the parameters are divided into the operations of incrementally updating the statistics and the operations of deriving the parameters from the statistics. The cost of updating the statistics through a pass of all data points is fixed, no matter how frequently the algorithm updates the parameters. The frequent updates increase only the cost of deriving the parameters from the statistics. The more frequently it updates the parameter, the more cost the algorithm will incur. Also, we show the advantages of performing frequent updates.

2.3.1 Clustering

Clustering is one of the most important tasks of data mining. It has been leveraged in many fields, including pattern recognition, image analysis, information retrieval, and bioinformatics.

2.3.1.1 K-means

K-means clustering [63] aims to partition n data points $\{x_1, x_2, ..., x_n\}$ into k ($k \leq n$) clusters $\{c_1, c_2, ..., c_k\}$ so as to minimize the objective function:

$$f = \sum_{i=1}^{k} \sum_{x_j \in c_i} \parallel x_j - \mu_{c_i} \parallel^2,$$

where $\mu_{c_i} = \frac{1}{|c_i|} \sum_{x_j \in c_i} x_j$ is the centroid of cluster c_i.

The most common algorithm of k-means clustering, Lloyd's algorithm [58], can be considered as an application of the EM algorithm. Its E-step assigns points to the cluster with the closest mean. That is, a data point x_j is assigned to cluster c if $c = \arg\min_j ||x_i - \mu_{c_j}||^2$. Its M-step updates the centroids (parameters) for all clusters. Let S_i ($S_i = \sum_{x_j \in c_i} x_j$) and W_i ($W_i = |c_i|$) be the statistics. The centroid of one cluster (*e.g.*, i) can be easily obtained by $\mu_i = \frac{S_i}{W_i}$. If a particular point x_i

changes its cluster assignment from c to c', the statistics can be incrementally updated as follows:

$$S_c = S_c - x_i, \quad S_{c'} = S_{c'} + x_i;$$

$$W_c = W_c - 1, \quad W_{c'} = W_{c'} + 1.$$

We here analyze the space complexity and the time complexity of k-means with frequent updates. In order to perform incremental computation (for frequent updates), we need to store cluster assignments for all data points and the statistics S_c and W_c, which only take $O(n + kd)$ space, where d is the dimension of a data point (in contrast, storing data points in memory takes $O(nd)$ space). We next analyze the complexity of frequent updates. Take the update by block method for example. Suppose data points are equally split into b blocks (with each block having n/b data points). Performing the E-step on one block takes $O(nkd/b)$ time, since processing one data point takes $O(kd)$ time. The following M-step takes $O(nd/b + kd)$ time, in which updating statistics S_c and W_c needs $O(nd/b)$ time and deriving all centroids from the statistics (*e.g.*, $\mu_i = \frac{S_i}{W_i}$) takes $O(kd)$ time. As a result, processing all data points in one pass (including multiple E-steps and M-steps) requires $O(nkd + bkd)$ time. Since $b \leq n$, the time can be represented as $O(nkd)$. Furthermore, we can show the original k-means (i.e., k-means with concurrent updates) also needs $O(nkd)$ time to process all data points in one pass. In other words, with incremental computation, the update by block approach will not increase the asymptotic time complexity no matter how frequent the M-step is performed. A similar conclusion can be obtained for the update by subrange approach.

2.3.1.2 Fuzzy C-means

Given a set of data points $\{x_1, x_2, ..., x_n\}$, Fuzzy c-means (FCM) [10, 25] aims to assign these data points into C clusters so as to minimize the objective function:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{C} \mu_{ij}^m \parallel x_i - c_j \parallel^2,$$

where m ($m > 1$) is the fuzzy factor, μ_{ij} is the degree of membership of x_i belonging to cluster j, and c_j is the centroid of cluster j. The degree of membership μ_{ij} and the centroid c_j are computed by the equations:

$$\mu_{ij} = \frac{1}{\sum_{k=1}^{C} (\frac{\|x_i - c_j\|}{\|x_i - c_k\|})^{\frac{2}{m-1}}}, \; c_j = \frac{\sum_{i=1}^{N} \mu_{ij}^m x_i}{\sum_{i=1}^{N} \mu_{ij}^m}.$$

If we describe FCM in the EM setting, its E-step updates the degree of membership for all data points, and its M-step updates the centroids (parameters) for all clusters. Let W_j ($W_j = \sum_{i=1}^{n} \mu_{ij}^m$) and X_j ($X_j = \sum_{i=1}^{n} \mu_{ij}^m x_i$) be the statistics in FCM. The centroid of one cluster (*e.g.*, j) can be easily obtained by $c_j = \frac{X_j}{W_j}$. For a data point x_i, if its degree of membership to cluster j changes from μ_{ij} to μ'_{ij}, the statistics can be incrementally updated as follows:

$$W_j = W_j - (\mu_{ij})^m + (\mu'_{ij})^m, \; X_j = X_j + ((\mu'_{ij})^m - (\mu_{ij})^m) x_i.$$

We now analyze the space complexity and the time complexity of FCM with frequent updates. In order to perform incremental computation, we need to store the degree of membership for all data points and the statistics W_j and X_j, which take $O(kn + kd)$ space. Similar to the time complexity analysis for k-means, we can show that processing all data points in one pass (including multiple E-steps and M-steps) requires $O(nkd)$ time for the update by block approach. Furthermore, original FCM also needs $O(nkd)$ time to process all data points in one pass. As a result, FCM with frequent updates does not increase the asymptotic time complexity.

2.3.1.3 Gaussian Mixture Model

Given a set of data points $\{x_1, x_2, ..., x_n\}$ which are generated by a mixture of k Gaussians, parameter estimation for the Gaussian Mixture Model (GMM) aims to

find the means and covariances of the k Gaussians and the weights that specify how likely each Gaussian is to be chosen so as to maximize the objective function:

$$\ell = \frac{1}{n}\sum_{i=1}^{n}\log(\sum_{j=1}^{k}\omega_j\phi(x_i|\mu_j,\Sigma_j)),$$

where ϕ represents the probability of a point coming from a Gaussian source. It is given by: $\phi(x_i|\mu_j,\Sigma_j) = \frac{1}{\sqrt{(2\pi)^d\cdot|\Sigma_j|}}\cdot e^{-\frac{1}{2}(x_i-\mu_j)^T\cdot\Sigma_j^{-1}\cdot(x_i-\mu_j)}$.

The parameters (weight, mean and covariance) of a Gaussian (*e.g.*, j) are respectively computed by the following equations:

$$\omega_j = \frac{1}{n}\sum_{i=1}^{n}\gamma_{ij},\ \mu_j = \frac{\sum_{i=1}^{n}\gamma_{ij}x_i}{\sum_{i=1}^{n}\gamma_{ij}},\ \Sigma_j = \frac{\sum_{i=1}^{n}\gamma_{ij}(x_i-\mu_j)(x_i-\mu_j)^T}{\sum_{i=1}^{n}\gamma_{ij}},$$

where γ_{ij} represents the probability of a point coming from a Gaussian, which is given by: $\gamma_{ij} = \frac{\omega_j\phi(x_i|\mu_j,\Sigma_j)}{\sum_{j=1}^{k}\omega_j\phi(x_i|\mu_j,\Sigma_j)}$.

When describing GMM in the EM setting, its E-step estimates the probability of a point coming from a Gaussian for all points, and its M-step updates the parameters of Gaussians.

The covariance matrix Σ is typically assumed to be diagonal to facilitate the computation of its inverse and determinant. Under such assumption, the statistics in the GMM algorithm are as follows: $R_j = \sum_{i=1}^{n}\gamma_{ij}$, $X_j = \sum_{i=1}^{n}\gamma_{ij}x_i$, $S_j = \sum_{i=1}^{n}\gamma_{ij}x_i^2$. Note that in this chapter, square on a vector means element-wise square, i.e., if a vector $y = [y_1, y_2, ..., y_d]$, then $y^2 = [y_1^2, y_2^2, ..., y_d^2]$.

Given the statistics, the parameters $\omega_j = R_j/n$, $\mu_j = X_j/R_j$ and $\Sigma_j = S_j/R_j - X_j^2/R_j^2$ can be easily obtained (here / means element-wise division). For a point x_i, if its probability to the source j changes from γ'_{ij} to γ_{ij}, the statistics can be computed as follows: $R_j = R_j + \gamma_{ij} - \gamma'_{ij}$, $X_j = X_j + (\gamma_{ij} - \gamma'_{ij})x_i$, $S_j = S_j + (\gamma_{ij} - \gamma'_{ij})x_i^2$.

Similar to the complexity analysis for FCM, we can show that GMM with frequent updates (e.g., the update by block method) needs $O(kn + kd)$ space to cache γ_{ij}, R_j,

X_j, and S_j. Additionally, it requires $O(nkd)$ time to process all data points in one pass (including multiple E-steps and M-steps). Furthermore, original GMM also needs $O(nkd)$ time to process all data points in one pass. As a result, GMM with frequent updates does not increase the asymptotic time complexity.

2.3.2 Topic Modeling

An EM algorithm is also a powerful tool for statistical text analysis, such as topic modeling. Topic modeling provides a way to navigate large document collections by discovering the themes that permeate a corpus. In particular, Latent Dirichlet Allocation (LDA) [12] is a popular topic modeling approach. It provides a generative model that describes how the documents in a corpus were produced. First, we denote the M given documents represented as $d_1, d_2, ..., d_M$. Let V denote the number of words in the vocabulary, and let N_i represent the number of words in a document d_i. Moreover, we use w_j to denote the j-th word in the vocabulary and $w_{i,j}$ to represent the j-th word in the i-th document. Assume that the documents are represented as random mixtures over K topics. A topic is a K dimensional multinomial distribution over words, and the i-th topic is denoted as ϕ_i. We use θ_i to represent the topic distribution for a document d_i. Furthermore, assume $w_{i,j}$ is drawn form topic $z_{i,j}$. In addition, we use α and β to represent hyper parameters of the Dirichlet distribution. LDA assumes the following generative process.

1. For each topic index $k \in \{1, ..., K\}$, draw topic distribution $\phi_k \sim Dir(\beta)$.
2. For each document $d_i \in \{d_1, d_2, ..., d_M\}$:
   - Draw topic distribution $\theta_i \sim Dir(\alpha)$.
   - For $j \in \{1, 2, ..., N_i\}$
     - Draw $z_{i,j} \sim Mult(\theta_i)$.
     - Draw $w_{i,j} \sim Mult(\phi_{z_{i,j}})$.

In the process, $Dir()$ denotes a Dirichlet distribution, and $Mult()$ represents a multinomial distribution.

There are two widely used approximate inference techniques for LDA. One is Markov chain Monte Carlo (MCMC) sampling (e.g., Gibbs sampling) [32], and the other one is variational inference [12]. Even though MCMC is a powerful methodology, the convergence of the sampler to its stationary distribution is usually hard to diagnose, and sampling algorithms may converge slowly in high dimensional models. Variational inference methods have clear convergence criterion and provide efficiency advantages over sampling techniques in high dimensional problems [76].

The basic idea of variational inference is to leverage Jensen's inequality to obtain an adjustable lower bound on the log likelihood of the posterior distribution. The variational inference breaks the coupling between θ and β to make the inference tractable. As a result, this variational inference has a posterior for each document in the form: $q(\theta, z|\gamma, \phi) = q(\theta|\gamma)\prod_{n=1}^{N} q(z_n|\phi_n)$, where the Dirichlet parameter γ and the multinomial parameters ϕ_n are the free variational parameters.

Furthermore, finding an optimal lower bound on the log likelihood can be represented as $(\gamma^*, \phi^*) = \arg\min D_{KL}(q(\theta, z|\gamma, \phi)||p(\theta, z|w, \alpha, \beta))$, which is a minimization of the Kullback-Leibler divergence between the variational distribution and the original posterior distribution. In turn, the likelihood (i.e., the objective function) for one document that the variational inference aims to maximize is as follows [12]:

$$\begin{aligned} L(\gamma, \phi; \alpha, \beta) = &\log\Gamma(\sum_{j=1}^{K}\alpha_j) - \sum_{i=1}^{K}\log\Gamma(\alpha_i) + \sum_{i=1}^{K}(\alpha_i - 1)(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{K}\gamma_j)) \\ &+ \sum_{n=1}^{N}\sum_{i=1}^{K}\phi_{ni}(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{K}\gamma_j)) + \sum_{n=1}^{N}\sum_{i=1}^{K}\sum_{j=1}^{V}\phi_{ni}w_{nj}\log\beta_{ij} \\ &- \log\Gamma(\sum_{j=1}^{K}\gamma_j) + \sum_{i=1}^{K}\log\Gamma(\gamma_i) - \sum_{i=1}^{K}(\gamma_i - 1)(\Psi(\gamma_i) - \Psi(\sum_{j=1}^{K}\gamma_j)) \\ &- \sum_{n=1}^{N}\sum_{i=1}^{K}\phi_{ni}\log\phi_{ni}, \end{aligned}$$

where $\Gamma()$ is the Gamma function and $\Psi()$ is the first derivative of the $\log \Gamma()$ function.

One popular method to minimize the Kullback-Leibler divergence (i.e., to maximize the above objective function) is to use an EM approach (e.g., Variational EM). Variational EM alternates between updating the expectations of the variational distribution q and maximizing the probability of the parameters given the observed documents. Here each document is one data point. Its E-step is illustrated in Algorithm 1. Its M-step updates α and β.

Algorithm 1: E-step for LDA

```
Set t = 0;
Initialize φ^t_ni = 1/K for all i and n;
Initialize γ_i = α_i + M/K for all i;
for d = 1 to M do
    repeat
        for n = 1 to N_d do
            for i = 1 to K do
                φ^{t+1}_dni = β_{i w_n} exp(Ψ(γ^t_di));
            normalize φ^{t+1}_dni to sum to 1;
        γ^{t+1} = α + Σ_{n=1}^{N_d} φ^{t+1}_dn;
        t = t + 1;
    until convergence of φ_d and γ_d ;
```

The M-step of variational EM updates α using a Newton-Raphson method. For ease of exposition, we assume all elements of α are the same unless otherwise stated, and thus α can be simply a single value in the following updates. Updates are carried out in log-space, as follows:

$$\log(\alpha^{t+1}) = \log(\alpha^t) - \frac{\partial L}{\partial \alpha} / (\frac{\partial^2 L}{\partial \alpha^2}\alpha + \frac{\partial L}{\partial \alpha}), \tag{2.1}$$

$$\frac{\partial L}{\partial \alpha} = M(K\Psi(K\alpha) - K\Psi(\alpha)) + \sum_{d=1}^{M}(\sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi(\sum_{j=1}^{K}\gamma_{dj})), \tag{2.2}$$

$$\frac{\partial^2 L}{\partial \alpha^2} = M(K^2\Psi'(K\alpha) - K\Psi'(\alpha)). \tag{2.3}$$

From Eq. (2.1) - Eq. (2.3), we can see that only the second part of $\frac{\partial L}{\partial \alpha}$ depends on each individual document. Therefore, in order to incrementally update α, we let $R = \sum_{d=1}^{M}(\sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi(\sum_{j=1}^{K}\gamma_{dj}))$, $s_d = \sum_{i=1}^{K}\Psi(\gamma_{di}) - K\Psi(\sum_{j=1}^{K}\gamma_{dj})$.

When updating a document, i, if its s_d changes from s_i' to s_i, we can incremental update the statistics R using $R = R + s_i - s_i'$.

Next, we show how to update β incrementally. We have $\beta_{ij} = \sum_{d=1}^{M}\sum_{n=1}^{N_d}\phi_{dni}w_{dnj}$ (the step of normalizing β_i to sum to 1 is skipped for simplicity). One simple way to perform incremental updates is to cache ϕ_{dni}. Then when a document changes ϕ'_{dni} to ϕ_{dni}, we can update β_{ij} using $\beta_{ij} = \beta_{ij} + \sum_{n=1}^{N_d}(\phi_{dni} - \phi'_{dni})w_{dnj}$. However, caching ϕ_{dni} for all documents takes $O(MKV)$ space, which can be huge. In order to address the space issue, we present a space-efficient incremental scheme, which is suitable for the update by block approach. We divide documents into b blocks, $\{B_1, B_2, ..., B_b\}$. Let

$$\beta_{ij}^{(l)} = \sum_{d \in B_l}\sum_{n=1}^{N_d}\phi_{dni}w_{dnj}. \tag{2.4}$$

Then, we have

$$\beta_{ij} = \sum_{l=1}^{b}\beta_{ij}^{(l)}. \tag{2.5}$$

When the documents in block l are updated, we compute $\beta_{ij}^{(l)}$ from scratch with Eq. (2.4), and then recover β_{ij} using Eq. (2.5). In this way, we only need to cache $\beta_{ij}^{(l)}$, $1 \leq l \leq b$. When b is small (e.g., a constant less than 10), then caching only takes $O(KV)$ space.

Furthermore, we can show that LDA with frequent updates (e.g., the update by block approach) need $O(KV)$ space to cache R, $\beta_{ij}^{(l)}$, and β_{ij} in order to support incremental computation. Additionally, performing the E-step on one document takes $O(IKV)$ time, where I the number of iterations the E-step needs to converge on the document. With incremental computation, if there are m documents updated in the E-step, the following M-step takes $O(mKV)$ time. As a result, LDA with

frequent updates (e.g., the update by block approach) requires $O(MIKV)$ time to process all documents in one pass (including multiple E-steps and M-steps). Furthermore, original LDA also needs $O(MIKV)$ time to process all documents in one pass. Consequently, LDA with frequent updates does not increase the asymptotic time complexity.

2.3.3 Advantages of Performing Frequent Updates

Since the EM algorithm with frequent updates utilizes the up-to-date parameters to estimate the distribution, it intuitively outperforms their concurrent update counterpart. We have performed multiple experiments on a single machine to demonstrate the advantages of frequent updates. The results, which can be seen in Section 2.6.2, show the EM algorithm with frequent updates converges faster compared to that with concurrent updates.

Our single machine experiments have illustrated the advantages of the EM algorithm with frequent updates. Moreover, some previous results [70, 85] also showed the advantages of the frequent updates for EM algorithm in a single machine setting. However, the EM algorithm with frequent updates in a single machine does not scale. Parallelizing the EM algorithm with frequent updates is important for real-world applications on massive datasets. The rest of this chapter will focus on parallelizing the EM algorithm with frequent updates.

2.4 Parallelizing Frequent Updates

The previous sections illustrate the EM algorithm with frequent updates is more efficient than that with concurrent updates. However, parallelizing frequent updates in a distributed environment is challenging. Although computing the distribution and incrementally updating the local statistics can be performed concurrently in each worker, updating the parameters in the M-step, which is based on the global statis-

tics, needs to be done in a centralized way. When processing the distributed data points, the algorithm has to synchronize the global statistics frequently. Synchronizing the global resources in a distributed environment may result in considerable overhead. Therefore, we need to control the parameter update frequency to achieve a good performance. In this section, we first briefly illustrates a natural method to parallelize the EM algorithm with concurrent updates. Then, we present two methods to parallelize the EM algorithm with frequent updates. Both of them can control the parameter update frequency. Moreover, in all the parallel methods, the input data is divided into multiple equal size partitions, and each worker holds one partition. The data is kept in the same worker throughout the iterative process to avoid the expensive data shuffling among workers.

2.4.1 Concurrent Method

In the traditional method of parallelizing concurrent updates, each worker computes the distribution for its local data points and updates the local statistics concurrently based on the parameters. After each worker finishes processing its local data points, all of them synchronize to derive the parameters from the global statistics. Then, each worker utilizes the updated parameters to compute the distribution in the next iteration. We refer to this method as *concurrent* method.

2.4.2 Partial Concurrent Method

Our first method to parallelize the EM algorithm with frequent updates is a parallel version of the update by block approach in Section 2.2.2. Recall that the update by block approach selects a block of data points for computing the distribution and then updates the parameters. The block size can control the parameter update frequency. As shown in Figure 2.1, our first parallel method allows each worker to pick a block of its local data points for computing the distribution and updating the local statistics. After processing the data points in the picked blocks, all the workers syn-

chronize to derive the new parameters from the global statistics. Then each worker leverages the updated parameters to compute the distribution for another block. All the blocks are of the same size m. Each worker rotates the block on its local data points. Since the data points in the picked blocks can be processed concurrently, we refer to this method as *partial concurrent* method. Obviously, the concurrent method is an extreme case of the partial concurrent method (when each worker selects all its local data points as one block). Furthermore, either when each worker works individually to compute the distribution or when all workers synchronize to derive the new parameters, the objective function keeps increasing (or decreasing, we assume "increasing" for brevity in this section). Therefore, we have the following theorem.

Theorem 2.4.1. *The partial concurrent method maintains the convergence property of an EM algorithm.*

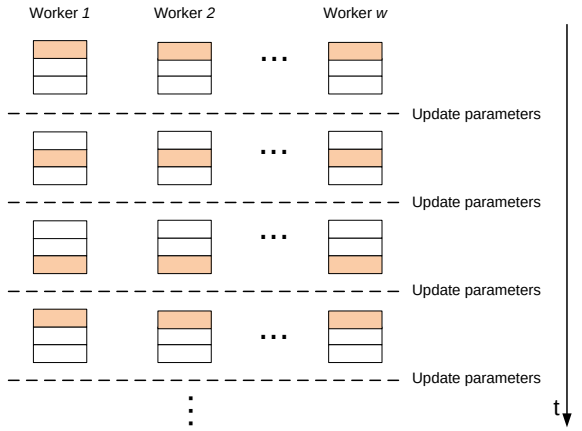


Figure 2.1. Process of the partial concurrent method. The colored box indicates the picked block of data points for computing the distribution.

The size of the block (*i.e.*, m) plays an important role on the efficiency of the partial concurrent method. It indicates the trade-off between the gain from computing the distribution with the frequently updated parameters and the cost from updating the parameters. Setting the size too small may incur considerable overhead

for updating the parameters. Setting the size too large may degrade the effect of the frequent updates. Nevertheless, a quite large range of the block size can improve the performance. The optimal block size will be discussed in Section 2.5.4. Our framework also provides a recommended block size.

2.4.3 Subrange Concurrent Method

Our second method to parallelize the EM algorithm with frequent updates corresponds to the update by subrange approach in Section 2.2.2. Recall that the update by subrange approach recomputes the distribution over the subrange of hidden variables. As shown in Figure 2.2, our second parallel method allows each worker to recompute the distribution among the subrange for its local data points and to update its local statistics. After each worker finishes recomputing the distribution among the subrange for all of its local data points, all the workers synchronize to compute the parameters based on the global statistics. Then, each worker utilizes the updated parameters to recompute the distribution under another subrange in the next iteration. Since all the data points can be processed concurrently under the subrange, we refer to the second method as *subrange concurrent* method. The subrange is randomly picked from the whole range of hidden variables. The concurrent method is an extreme case of the subrange concurrent method as well (when the whole range is picked as the subrange). Furthermore, either when each worker computes the distribution among the subrange or when all workers synchronize to derive the new parameters, the objective function keeps increasing. Therefore, we have the following theorem.

Theorem 2.4.2. *The subrange concurrent method maintains the convergence property of an EM algorithm.*

The subrange concurrent method might be more suitable for a "winner-take-all" version of EM application (*e.g.*, k-means), which constrains that one single value of the hidden variable is assigned probability 1 and all other values are assigned proba-

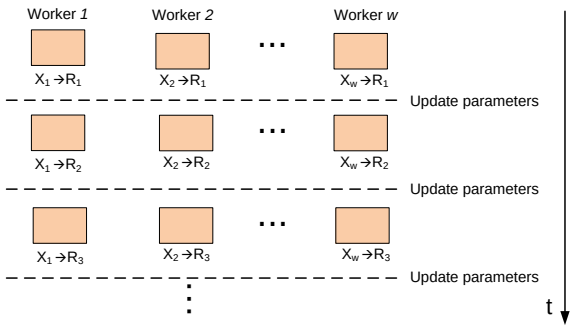


Figure 2.2. Process of the subrange concurrent method. Each worker recomputes the distribution among the subrange (R_i) for all of its local data points (X_j).

bility 0 (in k-means, a data point belongs to its current cluster in probability 1 and belongs to all other clusters in probability 0). In such an application, if a subrange does not include the value of probability 1, it is not necessary to recompute the distribution among the subrange. By avoiding unnecessary computation, a worker may dramatically reduce the time of processing data points in one iteration. Within the running time of one iteration of the concurrent method, the subrange concurrent method may proceed many iterations. Therefore, although the subrange concurrent method may increase the objective function less than the concurrent method in one single iteration, it still may increase the objective function faster (in terms of time). Moreover, the distribution for most of the data points usually will not change after first several iterations under the concurrent method, and thus the objective function probably increases slowly after first several iterations. Consequently, the concurrent method probably does not increase the objective function much more than the subrange concurrent method in one single iteration, which makes the subrange concurrent method more superior.

Like the block size in the partial concurrent method, the size of the subrange also impacts the efficiency of the subrange concurrent method. We will also discuss the optimal subrange size in Section 2.5.4.

2.5 FreEM

In this section, we propose FreEM, a distributed framework for efficiently implementing an EM algorithm. All the parallel methods mentioned in the previous section, including concurrent, partial concurrent, and subrange concurrent, are supported by our framework. FreEM is built on top of an in-memory version of iMapReduce [105]. The in-memory version of iMapReduce supports iterative process and loads data into memory for efficient data access. FreEM also provides high-level APIs, which are exposed to users for easily implementing EM algorithms.

2.5.1 Design of the Framework

Our framework consists of a number of *basic workers* and an *enhanced worker*. Each basic worker essentially leverages user-defined functions to compute the distribution and to update the parameters. Besides these operations, the enhanced worker also picks the subrange of hidden variable for all the workers under (and only under) the subrange concurrent method. Each worker stores a partition of the data points, the distribution of the corresponding hidden variables, the local statistics (the statistics for a worker's local data points), and the parameters, in memory. The partition of data points and the distribution are maintained in a key-value store, *point-based table*. Also, the local statistics and the parameters are maintained in a key-value store, *parameter-based table*.

2.5.2 Implementation of the Framework

Each worker in our framework has one pair of map and reduce tasks. In general, the map task performs the M-step, and the reduce task performs the E-step. The map task of the enhanced worker takes charge of picking the subrange of hidden variables. Both the point-based table and the parameter-based table of each worker is maintained by its reduce task.

To implement an EM algorithm, a user only needs to override several APIs. FreEM will automatically convert the EM algorithm to iMapReduce jobs. The first job is used to split the input data into multiple equal size partitions. The second job executes the EM algorithm, which consists of many iterations. In the first iteration, each map task utilizes a user-defined function (API 1) to obtain the initial guess of the parameters. Then, each map task sends the parameters to its paired reduce task. Each reduce task first loads one partition of the input data and then leverages a user-defined function (API 2) to compute the distribution and to initialize its local statistics. After that, a reduce task broadcasts its local statistics to all map tasks. In each of the following iterations, each map task uses a user-defined function (API 3) to accumulate the local statistics it received to the global statistics. When it receives the local statistics from all reduce tasks, a map task uses another user-defined function (API 4) to derive the parameters from the global statistics. Then, each map task sends the updated parameters to its paired reduce task. A reduce task leverages another user-defined function (API 5) to recompute the distribution (under a given subrange) and to incrementally update its local statistics based on the updated parameters. After it finishes processing the given block of data points, a reduce task broadcasts its updated local statistics to all map tasks again. Such iterative process continues until the number of iterations exceeds a threshold or the objective function reaches a specified value, when our framework terminates all the tasks. Note that the map task of the enhanced worker also picks a subrange and broadcasts it to all reduce tasks under the subrange concurrent method.

2.5.3 API

FreEM provides several high-level APIs, which are exposed to users for easily implementing an EM algorithm in a distributed environment. The APIs are as follows:
1. `void initPara(Para, Points)`: specify the initial guess at the parameters.

2. `void initLocalStat(Dist, LocalStat, Para, Points)`: compute the distribution based on the initial guess at the parameters, and initialize the local statistics.
3. `void accuStat(LocalStat, GlobalStat)`: accumulate the local statistics to the global statistics.
4. `void updatePara(GlobalStat, Para)`: update the parameters based on the global statistics.
5. `void Estep(Dist, Para, SubRange, LocalStat, Points)`: recompute the distribution under the given subrange based on current parameters, and incrementally update the local statistics.

2.5.4 Setting Parameters for Parallel Methods

The size of the block in the partial concurrent method and the size of the subrange in the subrange concurrent method can significantly impact the performance of the algorithm. In this section, we discuss how to determine the optimal block size and how to seek the optimal subrange size.

2.5.4.1 Optimal Block Size

For the partial concurrent method, let m be the block size. We use T_{sgl} to represent the average time of processing one data point, consisting of the time for computing the distribution and the time for updating local statistics, and use T_{vhd} to represent the time spending on updating the parameters, consisting of the time for accumulating the global statistics, the time for updating the parameters, and the time of synchronization. Let $F(m)$ be the total times of data points being processed in the E-step for reaching a specified objective function value (i.e., the pre-defined convergence point) when the block size is m. Then, $\frac{F(m)}{m}$ is the total number of iterations. Thus, the total running time for reaching the convergence point is $\{F(m) \cdot T_{sgl} + \frac{F(m)}{m} \cdot T_{vhd}\}$. Therefore, the optimal m is given by:

$$\arg\min_m \{F(m) \cdot T_{sgl} + \frac{F(m)}{m} \cdot T_{vhd}\},$$

where T_{sgl} and T_{vhd} can be measured. The key of finding the optimal m is the function $F(m)$.

The experimental results demonstrate that $F(m)$ is roughly a linear function of m, *i.e.*, $F(m) = a \cdot m + b$, as will be shown in Section 2.6.4. Then, we can derive the optimal block size m^*:

$$m^* = \sqrt{\frac{b \cdot T_{vhd}}{a \cdot T_{sgl}}}.$$

Among the factors determining the optimal block size, only T_{vhd} and T_{sgl} can be easily measured. Therefore, we consider m is linear in $\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. We can explore different settings of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ to seek the optimal block size. In our framework, we set $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ to be 300 by default. The default setting achieves near optimal performance as will be shown in Section 2.6.4.

Our framework measures T_{vhd} and T_{sgl} in the following way. When it executes an EM algorithm, FreEM first sets the block size as a pre-defined number (*e.g.*, $\frac{n}{4 \cdot w}$, where n is the total number of data points and w is the number of workers). Then, each worker measures its own T_{vhd} and T_{sgl}, and reports their values to the enhanced worker in each iteration. The enhanced worker accumulates both of them, respectively. After a few (*e.g.*, 3) iterations, the enhanced worker computes the average values of both T_{vhd} and T_{sgl} and specifies $300 \cdot \sqrt{\frac{T_{vhd}}{T_{sgl}}}$ as the optimal block size.

2.5.4.2 Optimal Subrange Size

For the subrange concurrent method, let s be the subrange size and r be the size of the whole range. Suppose $\Delta f(s)$ is the averaging increase of the objective function for computing the distribution among the subrange for all data points and updating the parameters when the subset size is s. Since the time of processing one data point is usually proportional to the subrange size, $\frac{s}{r} \cdot T_{sgl}$ is the time for processing one

data point under the subrange concurrent method. Therefore, $\frac{n}{w} \cdot \frac{s}{r} \cdot T_{sgl} + T_{vhd}$ is the running time of processing all data points in one iteration. Consequently, the optimal subrange size is given by:

$$\arg\max_{s}\{\frac{\Delta f(s)}{\frac{n}{w} \cdot \frac{s}{r} \cdot T_{sgl} + T_{vhd}}\}.$$

We can use empirical approaches to seek the optimal subrange size, as will be discussed in Section 2.6.4. Also, we provide a scheme to judge when the subrange concurrent method is superior to the concurrent method. Obviously, we can expect that the subrange concurrent method will outperform the concurrent method when the following inequality holds.

$$\frac{\Delta f(s)}{\frac{n}{w} \cdot \frac{s}{r} \cdot T_{sgl} + T_{vhd}} > \frac{\Delta f(r)}{\frac{n}{w} \cdot T_{sgl} + T_{vhd}}. \tag{2.6}$$

From Inequation (2.6), we can derive another inequation, which is easier to solve, as follows:

$$\frac{\Delta f(s)}{\Delta f(r)} > \frac{r}{s} + \frac{\frac{r-s}{r} \cdot T_{vhd}}{\frac{n}{w} \cdot T_{sgl} + T_{vhd}}. \tag{2.7}$$

All the factors in the right side of Inequation (2.7) either are known or can be measured. Accordingly, it provides a nice bound to estimate whether the subrange concurrent method achieves better performance than the concurrent method.

2.6 Evaluation

In this section, we evaluate the effectiveness and efficiency of EM algorithms with frequent updates on a single machine and in a distributed environment. All the applications described in Section 2.3 are evaluated. For the distributed environment, all the parallel methods, including concurrent, partial concurrent, and subrange concurrent, are implemented and evaluated on FreEM. We also compare the concurrent method on FreEM with that on Hadoop.

Table 2.1. Datasets for Clustering

Algorithm	Dataset	# Points	Dim
k-means/FCM	Covtype	581,012	54
	KDDCUP	4,898,431	42
GMM	Synth-M	400,000	60
	Synth-L	1,000,000	60

Table 2.2. Datasets for Topic Modeling

Dataset	# Documents	# Unique Words	# Total Words
KOS	3430	6906	467,714
Enron	39861	28102	6,400,000
NYTimes	300000	102660	100,000,000

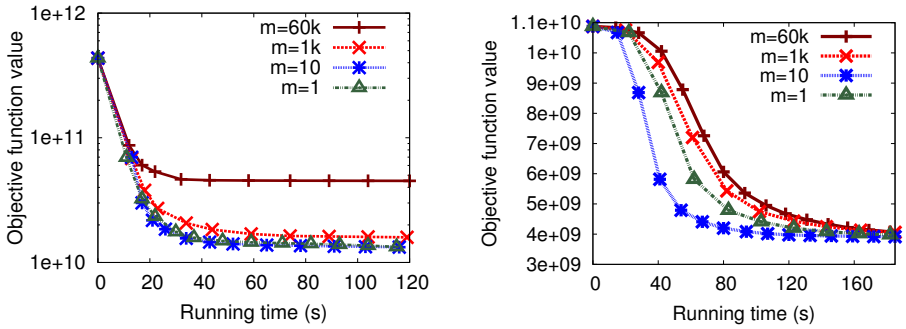
2.6.1 Experiment Setup

We build a small-scale cluster of local machines and a large-scale cluster on Amazon EC2 [1]. The small-scale cluster consists of 4 machines, and each one has a dual-core 2.66GHz CPU, 4GB of RAM, 1TB of disk. These 4 machines are connected through a switch with the bandwidth of 1Gbps. The Amazon cluster consists of 40 medium instances, each of which having 2 EC2 compute units, 3.75GB of RAM, and 400GB of hard disk.

Real-world datasets from UCI Machine Learning Repository [3] and synthetic datasets are leveraged to evaluate the EM applications. The synthetic datasets are generated in such a way: each dimension of one data point follows a Gaussian distribution with random mean and standard deviation 1.0. The datasets are summarized in Table 2.1 and Table 2.2.

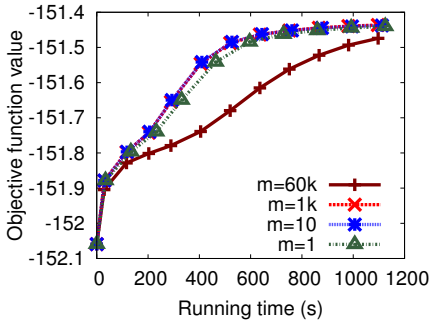
2.6.2 Single Machine Experiments

We first demonstrate the advantages of the EM algorithm with frequent updates on one single machine. The update by block approach is used as an example. All the EM applications described in Section 2.3 are implemented in Java.

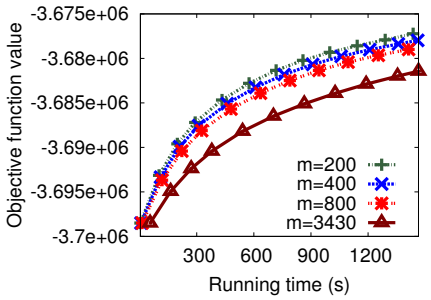


(a) K-means on sampled Covtype

(b) FCM on sampled Covtype



(c) GMM on sampled Synth-M



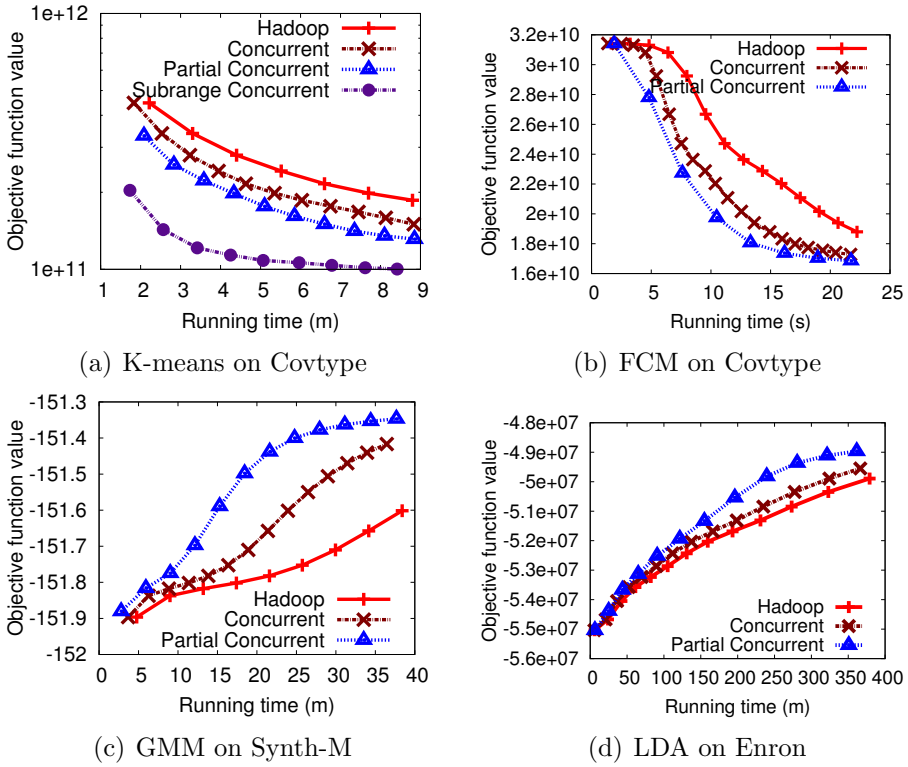
(d) LDA on KOS

Figure 2.3. Convergence speed on the single machine.

First, we perform the three clustering applications, k-means, FCM, and GMM, with various block size (m). For a fair comparison, each application runs on one dataset with the same initial start. Datasets sampled from the original datasets are used in the evaluation. Each dataset consists of $60,000$ data points. We sample the datasets since a single commodity machine cannot hold the whole dataset in memory. Figures 2.3(a) - 2.3(c) present the convergence speed. As shown, the EM algorithm with frequent updates ($m < 60k$) converges faster and may achieve a better convergence point, compared to that with concurrent updates ($m = 60k$). These figures also demonstrate the update frequency (determined by the block size) has a significant impact on the performance. Then, we perform LDA on the KOS dataset. Figure 2.3(d) plots the convergence speed with different block sizes. They further show that the EM algorithm with frequent updates converge faster than that with concurrent updates and that the update frequency impacts the performance.

2.6.3 Small-scale Cluster Experiments

FreEM allows the EM algorithm to frequently update the parameters in a distributed environment and leverage the up-to-date parameters in its E-step. Therefore, the EM algorithm with frequent updates has the potential to reach the convergence point with less workload, compared to that with concurrent updates. To evaluate the effect of frequent updates, we compare the convergence speed of the partial/subrange concurrent method with that of the concurrent method. In addition, since MapReduce is a popular framework, we utilize the convergence speed of the concurrent method on its open-source, implementation, Hadoop, as the base line.



(a) K-means on Covtype

(b) FCM on Covtype

(c) GMM on Synth-M

(d) LDA on Enron

Figure 2.4. Convergence speed on the small-scale cluster.

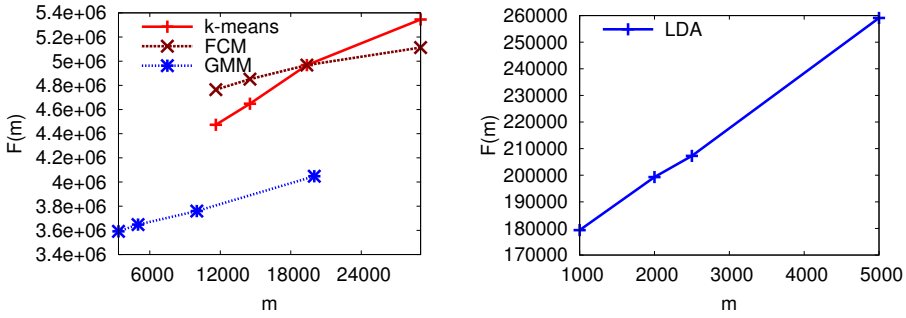
The convergence speed evaluation is first performed on the local cluster. All the methods start with the same initial setting, when compared on the same dataset. We set the number of clusters as 80 for all experiments of clustering applications, unless otherwise specified. Figure 2.4(a) - Figure 2.4(c) show the performance comparison.

We can see that the partial concurrent method converges faster than the concurrent method for all the three clustering applications. The subrange concurrent method converges faster and converges to a much better point than the concurrent method for k-means. Unfortunately, the subrange concurrent method seems to be slower than the concurrent method on FCM, GMM, and LDA, with several subrange sizes we test. Additionally, the convergence speed of the concurrent method on FreEM is much faster than that on Hadoop. The reasons are twofold. One reason is that our framework maintains data in memory and thus avoids repeatedly loading data. The other reason is that FreEM is built on top of iMapReduce, which is more efficient in supporting iterative process than Hadoop by using persistent map and reduce tasks. For example, iMapReduce is more efficient than Hadoop in supporting graph based iterative algorithms [105, 106]. Additionally, according to the experimental results, it seems that the subrange concurrent method is suitable for "winner-take-all" version of EM applications and the partial concurrent method is suitable for all EM applications. For LDA, we set the number of topics as 100. From Figure 2.4(d), we can see that the partial concurrent method converges faster than the concurrent method.

2.6.4 Optimal Block Size and Subrange Size

For the partial concurrent method, the block size significantly impacts the performance. In Section 2.5.4, we discussed the optimal block size depends on several factors. The key is to figure out the function $F(m)$. We estimate $F(m)$ for different applications of the EM algorithm on the small-scale cluster. The result, as shown in Figure 2.5, demonstrates that $F(m)$ is roughly a linear function of m.

Since only T_{vhd} and T_{sgl} can be easily measured, we set the block size based on them. Our framework sets the block size m in proportional to $\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. We perform experiments of all the four EM applications on our small-scale cluster to see the effects of various settings of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$. Figure 2.6 shows the speedup with different settings.



(a) Clustering Applications (b) LDA

Figure 2.5. Function F(m).

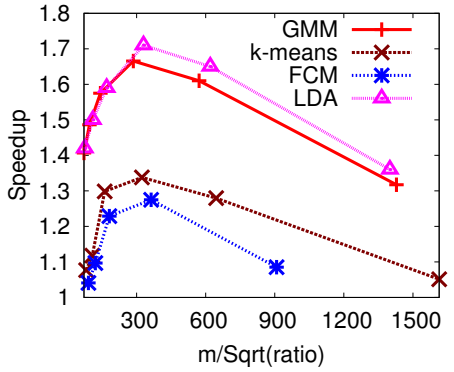


Figure 2.6. Speedup. X-axis represents the values of $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$.

From the figure, we can see that all the applications demonstrate the best speedup when $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}}$ is set to be around 300. This is the reason why our framework sets $m/\sqrt{\frac{T_{vhd}}{T_{sgl}}} = 300$ by default.

For the subrange concurrent method, we use empirical approaches to seek the optimal subrange size. Our experimental results reveal that if one subrange size is better than another during the initial iterations, it is also better in the following iterations (*e.g.*, as shown in Figure 2.7 for k-means). Given the observation, we can try several subrange sizes, and pick the one that achieves the best performance in the first several iterations.

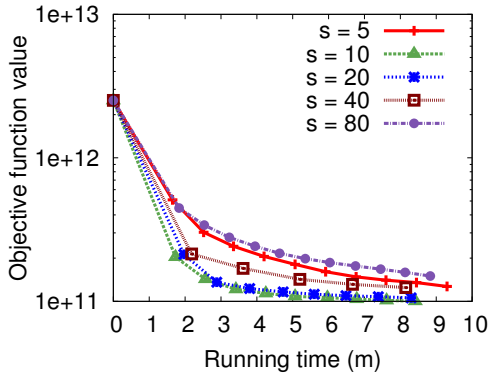


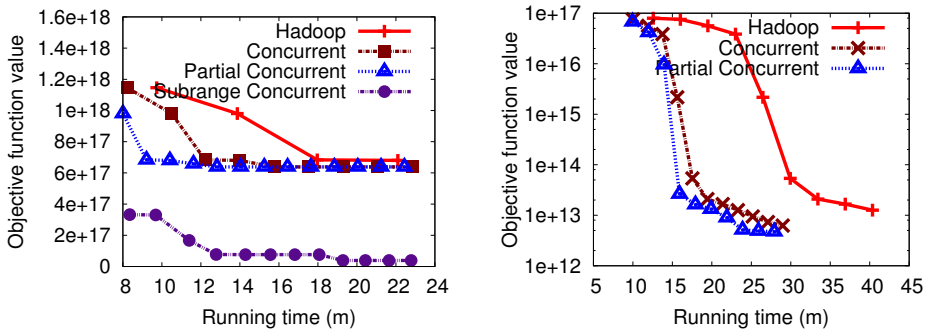
Figure 2.7. Varying subrange size.

2.6.5 Large-scale Cluster Experiments

In order to validate the scalability of FreEM, we also evaluate it on the Amazon EC2 cloud. We first show the performance comparison when all the 40 instances are used. From Figure 2.8, we can see that the partial concurrent method converges faster than the concurrent method for all the EM applications and that the subrange concurrent method converges faster and converges to a much better point than the concurrent method for k-means.

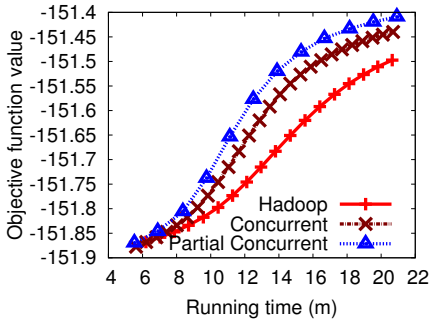
We then evaluate the scaling performance of FreEM as the number of workers increases from 10 to 40. The speedup is measured relative to the running time of 10 workers. Here the running time means the wall clock time that an EM application takes to reach a pre-defined objective function value. The speedup of the partial concurrent method is tested on GMM, and that of the subrange concurrent method is measured on k-means. The speedup of the concurrent method is also evaluated to be a reference point.

Figure 2.9 and Figure 2.10 show that both the concurrent method and the partial concurrent method exhibit good speedups. The concurrent method demonstrates a better speedup, as presented in Figure 2.9(a), since it updates the parameters only once through one pass of all data points and thus incurs less synchronization overhead. Note that the bases of computing speedups for both methods are different, and thus

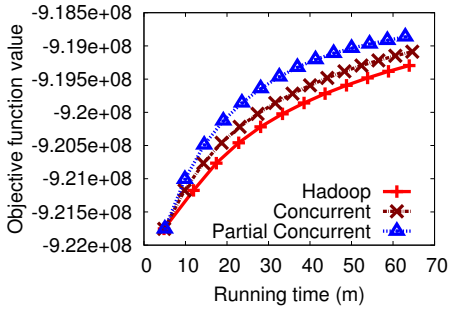


(a) K-means on KDDCUP

(b) FCM on KDDCUP

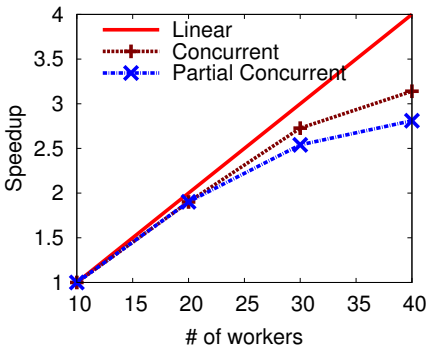


(c) GMM on Synth-L

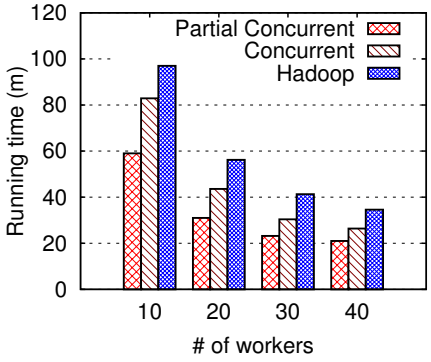


(d) LDA on NYTimes

Figure 2.8. Convergence speed on the Amazon EC2 cloud.



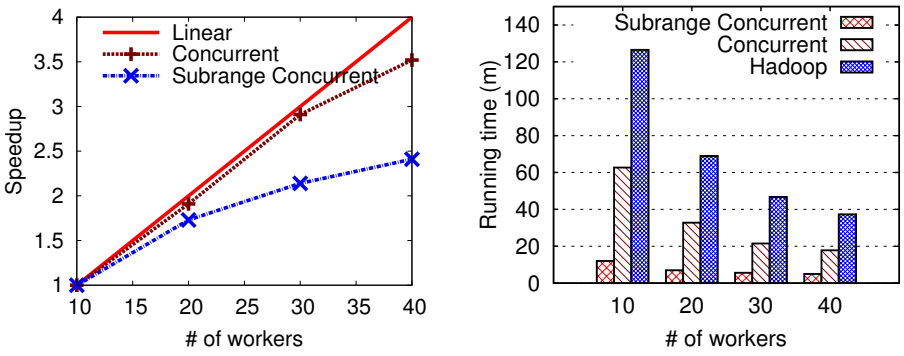
(a) Speedup of partial concurrent



(b) Performance comparison

Figure 2.9. Scaling performance of the partial concurrent method.

a better speedup does not necessarily mean a shorter running time. As shown in Figure 2.9(b), the partial concurrent method still converges faster than the concurrent method even on 40 workers. Since it has a better speedup, the concurrent method will obtain the same convergence speed as the partial concurrent method when the number of workers reaches some point. At that point, the partial concurrent method



(a) Speedup of subrange concurrent (b) Performance comparison

Figure 2.10. Scaling performance of the subrange concurrent method.

will degrade to the concurrent method by setting the right block size. For similar reasons, the concurrent method also exhibits a better speedup than the subrange concurrent method, as plotted in Figure 2.10(a). However, the subrange concurrent method still runs much faster than the concurrent method even on 40 workers, as shown in Figure 2.10(b).

2.7 Related Work

The EM algorithm has been applied very widely. Due to the popularity of the EM algorithm, many approaches for accelerating it have been proposed. For example, Dempster *et al.* [23] and Meng *et al.* [66] present a partial M-step may accelerate the algorithm when maximizing the likelihood in the M-step is inefficient. Such a partial M-step attempts to find the new estimates for the parameters improving the likelihood rather than maximizing it. In contrast, our work focuses on how to frequently perform the M-step to accelerate the algorithm. As the most relevant works, the works of Neal *et al.* [70] and Thiesson *et al.* [85] also show a partial E-step which selects a block of data points for computing the distribution may accelerate the EM algorithm in the single machine setting. Neal *et al.* [70] prove that such a variant of the EM algorithm converges. Thiesson *et al.* [85] provide an empirical

method to figure out the near optimal block size. Our proof is inspired by the work of Neal *et al.*, but goes further. Specifically, we prove that not only selecting a block of data points for computing the distribution but also computing the distribution under a subrange of hidden variables can guarantee the convergence. Compared to the work of Thiesson *et al.*, which is in the single machine setting, our work considers the scenario of a distributed environment. We propose a distributed framework for efficiently implementing the EM algorithm with frequent updates. Furthermore, these two pieces of work demonstrate the power of frequent update through only one EM application, parameter estimation for a finite mixture model, whereas our work covers more applications.

There are a number of efforts targeted on parallelizing the EM algorithm as well. Most of them focused on efficiently updating the parameters in the M-step. For examples, Wolfe *et al.* [90] propose an approach to distribute both the E-step and the M-step based on MapReduce. Kowalczyk *et al.* [45] present a gossip-based distributed implementation of the EM algorithm for GMM. Zhai *et al.* [103] introduce a MapReduce-based implementation of the EM algorithm for LDA. While our work has a different focus: we study how to frequently update the parameters to speed up convergence for a wide class of EM algorithms.

2.8 Conclusion

Motivated by the observations that an EM algorithm performing frequent updates is much more efficient than it performing concurrent updates, we propose two approaches to parallelize the EM algorithm with frequent updates in a distributed environment so as to scale to massive datasets. Furthermore, we formally prove that the EM algorithm with frequent updates converges. To support the efficient implementation of the EM algorithm with frequent updates, we design and implement a distributed framework, FreEM. We deploy FreEM on both a local cluster and the

Amazon EC2 cloud, and evaluate its performance in the context of two categories of EM applications, clustering and topic modeling. The evaluation results show that the EM algorithm with frequent updates can run much faster than the EM algorithm with traditional concurrent updates when both are implemented on FreEM. In addition, since FreEM is on top of iMapReduce which is more efficient than MapReduce in supporting iterative algorithms, FreEM is more efficient than MapReduce in supporting the EM algorithm.

CHAPTER 3

SCALABLE DISTRIBUTED NONNEGATIVE MATRIX FACTORIZATION WITH BLOCK-WISE UPDATES

3.1 Introduction

Nonnegative matrix factorization (NMF) [49] is a popular dimension reduction and factor analysis method that has attracted a lot of attention recently. It arises from a wide range of applications, including genome data analysis [16], text mining [71], and recommendation systems [97]. NMF factorizes an original matrix into two nonnegative low-rank factor matrices by minimizing a loss function, which measures the discrepancy between the original matrix and the product of the two factor matrices. Due to its wide applications, many algorithms [33, 40, 50, 54, 55, 109, 111] for solving it have been proposed. NMF algorithms typically leverage update functions to iteratively and alternatively refine factor matrices.

Many practitioners nowadays have to deal with NMF on massive datasets. For example, recommendation systems in web services such as Netflix have been dealing with NMF on web-scale dyadic datasets, which involve millions of users, millions of movies, and billions of ratings. For such web-scale matrices, it is desirable to leverage a cluster of machines to speed up the factorization process. MapReduce [22] has emerged as a popular distributed framework for data intensive computation. It provides a simple programming model where a user can focus on the computation logic without worrying about the complexity of parallel computation. Prior approaches (e.g., [57]) of handling NMF on MapReduce usually pick an NMF algorithm and then focus on implementing matrix operations on MapReduce.

In this chapter, we present a new form of factor matrix update functions. This new form operates on blocks of matrices. In order to support the new form, we partition the factor matrices into blocks along the short dimension to maximize the parallelism and split the original matrix into corresponding blocks. The new form allows us to update distinct blocks independently and simultaneously when updating a factor matrix. It also facilitates distributed implementations. Different blocks of one factor matrix can be updated in parallel, and can be distributed in memories of all machines of a cluster and thus avoid overflowing the memory of one single machine. Storing factor matrices in memory can support random access and local aggregation. As a result, the new form of update functions leads to an efficient MapReduce implementation. We illustrate that the new form works for NMFs with a wide class of loss functions.

Moreover, under the new form of update functions, we can update a subset of its blocks instead of all the blocks when we update a factor matrix. The number of blocks in the subset is adjustable, and the only requirement is that when one factor matrix is updated the other one is fixed. For instance, we can update one block of a factor matrix and then immediately update all the blocks of the other factor matrix. We refer to this kind of updates as *frequent block-wise updates*. Frequent block-wise updates aim to utilize the most recently updated data whenever possible. As a result, frequent block-wise updates are more efficient than their traditional concurrent counterparts, *concurrent block-wise updates*, which updates all the blocks of either factor matrix alternately. Additionally, frequent block-wise updates maintain the convergence property of the algorithm.

We present implementations of block-wise updates for two classical NMFs: one uses the square of Euclidean distance as the loss function, and the other uses the generalized KL-divergence. We implement concurrent block-wise updates on MapReduce, and implement both concurrent and frequent block-wise updates on an extended ver-

sion of MapReduce, iMapReduce [105], which supports iterative computations more efficiently. We evaluate these implementations on a local cluster as well as the Amazon EC2 cloud [1]. With both synthetic and real-world datasets, the evaluation results show that our MapReduce implementation for concurrent block-wise updates is 19x - 107x faster than the existing MapReduce implementation [57] with the traditional form of update functions, and our iMapReduce implementation further achieves up to 3x speedup over our MapReduce implementation. Furthermore, the iMapReduce implementation with frequent block-wise updates is up to 2.7x faster than that with concurrent block-wise updates. Accordingly, our iMapReduce implementation with frequent block-wise updates is up to two orders of magnitude faster than the existing MapReduce implementation.

The rest of this chapter is organized as follows. Section 3.2 briefly reviews the background of NMF. Section 3.3 introduces block-wise updates. Concurrent block-wise updates and frequent block-wise updates are presented in Section 3.4. Section 3.5 provides our efficient implementations of distributed block-wise updates. Section 3.6 presents the evaluation results. Section 3.7 surveys related work, and this chapter is concluded in Section 3.8.

3.2 Background

NMF aims to factorize an original matrix A into two nonnegative low-rank factor matrices W and H. Matrix A's elements must be nonnegative by assumption. The achieved factorization has the property of $A \simeq WH$. A loss function is leveraged to measure the discrepancy between A and WH. More formally:

Given $A \in \mathbb{R}_+^{m \times n}$ and a positive integer $k \ll \min\{m, n\}$, find $W \in \mathbb{R}_+^{m \times k}$ and $H \in \mathbb{R}_+^{k \times n}$, such that a loss function $L(A, W, H)$ is minimized.

The loss function $L(A, W, H)$ is typically not convex in both W and H together. Hence, it is unrealistic to have an approach of finding the global minimum. Fortunately, there are many techniques for finding local minima.

A general approach is to adopt the block coordinate descent rules [55]:

- Initialize W, H with nonnegative W^0, H^0, $t \leftarrow 0$.
- 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}: $L(A, W^{t+1}, H^{t+1}) \leq L(A, W^t, H^{t+1})$.

  When the matrix loss function is the square of the Euclidean distance, i.e.,

$$L(A, W, H) = ||A - WH||_F^2, \tag{3.1}$$

where $||\cdot||_F$ is the Frobenius norm, one of the most well-known algorithms for implementing the above rules is Lee and Seung's multiplicative update approach [50]. It updates W and H as follows:

$$H = H * \frac{W^T A}{W^T W H}, \qquad W = W * \frac{A H^T}{W H H^T}, \tag{3.2}$$

where the symbol "$*$" and the symbol "-" (or equivalently "/") are used to denote the element-wise matrix multiplication and division, respectively.

3.3 Distributed NMF

In this section, we present how to apply the block coordinate descent rules to NMF in a distributed environment.

3.3.1 Decomposition

A loss function is usually decomposable [79]. That is, it can be represented as the sum of losses for all the elements in the matrix. For example, the well adopted

loss function, the square of the Euclidean distance, is decomposable. We list several popular decomposable loss functions in Table 3.1. To achieve better sparsity in W and H, regularization terms have been proposed to add into loss functions [36]. For example, the square of the Euclidean distance with an L1-norm regularization on W and H can achieve a more sparse solution:

$$L(A, W, H) = ||A - WH||_F^2 + \alpha \sum_{(i,r)} W_{ir} + \beta \sum_{(r,j)} H_{rj},$$

where $\alpha > 0$ and $\beta > 0$ are regularization parameters which trade off the original loss function with the regularizer. Another common loss function with the regularization term [97] is as follows :

$$L(A, W, H) = ||A - WH||_F^2 + \lambda(||W||_F^2 + ||H||_F^2),$$

where λ is the regularization parameter. One can also replace $(||W||_F^2 + ||H||_F^2)$ with $\sum_{i,j}(||W_i||^2 + ||H_j||^2)$ (where $||\cdot||$ denotes the L2-norm of a vector) to obtain another loss function. The regularization term itself is usually decomposable as well. Therefore, the final loss function is decomposable. We focus on NMF with decomposable loss functions.

Table 3.1. Decomposable Loss Functions

Square of Euclidean distance	$\sum_{(i,j)}(A_{ij} - [WH]_{ij})^2$
KL-divergence	$\sum_{(i,j)} A_{ij} \log \frac{A_{ij}}{[WH]_{ij}}$
Generalized KL-divergence (I-divergence)	$\sum_{(i,j)}(A_{ij} \log \frac{A_{ij}}{[WH]_{ij}} - A_{ij} + [WH]_{ij})$
Itakura-Saito distance	$\sum_{(i,j)}(\frac{A_{ij}}{[WH]_{ij}} - \log \frac{A_{ij}}{[WH]_{ij}} - 1)$

Distributed NMF needs to partition the matrices W, H, and A across compute nodes. To this end, we leverage a well-adopted scheme in gradient descent algorithms [28, 84], which partitions W and H into blocks along the short dimension to

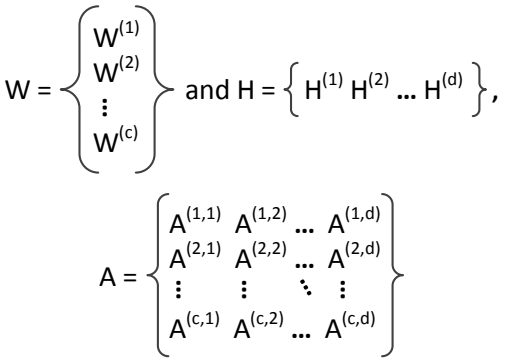


Figure 3.1. Block-wise partition scheme for distributed NMF.

maximize the parallelism and splits the original matrix A into corresponding blocks. We use symbol $W^{(I)}$ to denote the Ith block of W, $H^{(J)}$ to denote the Jth block of H, and $A^{(I,J)}$ to denote the corresponding block of A (i.e., the (I, J)th block). Under this partition scheme, $A^{(I,J)}$ is only related to $W^{(I)}$ and $H^{(J)}$ when computing the loss function and is independent of other blocks of W and H, in terms of loss value (computed by the loss function). We refer to the partition scheme as *block-wise partition*. The view of the block-wise partition scheme is shown in Figure 3.1. Previous work on distributed NMF [57] also proposes to partition W and H along the short dimension. The key difference between this partition scheme and the block-wise partition scheme is that the former splits W and H into row and column vectors, respectively, while the latter splits W and H into blocks. Since one block of W and one block of H can contain a set of row and column vectors, respectively, the block-wise partition scheme can be considered as a more general scheme. Moreover, the block-wise partition scheme splits A into blocks as well.

Due to its decomposability, loss function $L(A, W, H)$ can be expressed as

$$L(A, W, H) = \sum_{I} \sum_{J} L(A^{(I,J)}, W^{(I)}, H^{(J)}). \tag{3.3}$$

Let

$$F_I = \sum_{J} L(A^{(I,J)}, W^{(I)}, H^{(J)}), \tag{3.4}$$

$$G_J = \sum_I L(A^{(I,J)}, W^{(I)}, H^{(J)}), \tag{3.5}$$

then we have

$$L(A, W, H) = \sum_I F_I = \sum_J G_J. \tag{3.6}$$

F_I and G_J can be seen as local loss functions. The overall loss function L is a sum of local loss functions. By fixing H, F_I is independent of each other. Therefore, F_I can be minimized independently and simultaneously by fixing H. Similarly, G_J can be minimized independently and simultaneously by fixing W.

3.3.2 Block-wise Updates

The block-wise partition allows us to update its blocks independently when updating a factor matrix (by fixing the other factor matrix). In other words, each block can be treated as one update unit. We refer to this kind of updates as *block-wise updates*. In the following, we illustrate how to update one block of W (by minimizing F_I) and that of H (by minimizing G_J). We take the square of the Euclidean distance and the generalized KL-divergence as examples. Nevertheless, the techniques derived in this section can be applied to any other decomposable loss function.

3.3.2.1 Square of Euclidean Distance

Here we first show how to update one block of H (i.e., $H^{(J)}$) when the square of the Euclidean distance is leveraged as the loss function. We refer to this type of NMF as *SED-NMF*. When W is fixed, minimizing G_J can be expressed as follows:

$$\min_{H^{(J)}} G_J = \min_{H^{(J)}} \sum_I ||A^{(I,J)} - W^{(I)} H^{(J)}||_F^2.$$

We here leverage gradient descent to update $H^{(J)}$:

$$H_{uv}^{(J)} = H_{uv}^{(J)} - \eta_{uv} \frac{\partial G_J}{\partial H_{uv}^{(J)}},$$

where $H_{uv}^{(J)}$ denotes the element at the uth row and the vth column of $H^{(J)}$, and η_{uv} is an individual step size for the corresponding gradient element, and

$$\frac{\partial G_J}{\partial H_{uv}^{(J)}} = [\sum_I ((W^{(I)})^T W^{(I)} H^{(J)} - (W^{(I)})^T A^{(I,J)})]_{uv}.$$

If all step sizes are set to some sufficiently small positive number, the update should reduce G_J. However, if the number is too small, the decreasing speed can be very slow. To obtain a good speed, we derive step sizes by following Lee and Seung's approach:

$$\eta_{uv} = \frac{H_{uv}^{(J)}}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]_{uv}}.$$

Then, we have:

$$H_{uv}^{(J)} = H_{uv}^{(J)} \frac{[\sum_I (W^{(I)})^T A^{(I,J)}]_{uv}}{[\sum_I (W^{(I)})^T W^{(I)} H^{(J)}]_{uv}}. \tag{3.7}$$

Similarly, we can derive the update formula for $W^{(I)}$ as follows:

$$W_{uv}^{(I)} = W_{uv}^{(I)} \frac{[\sum_J A^{(I,J)} (H^{(J)})^T]_{uv}}{[\sum_J W^{(I)} H^{(J)} (H^{(J)})^T]_{uv}}. \tag{3.8}$$

We have derived the update formulae with the gradient descent method. It is important to note that we can also utilize other techniques, such as the active set method [40] and the block principal pivoting method [41], to derive the update formulae. Furthermore, we can even use different methods for different blocks at the same time. For example, we can use the gradient descent method to update half of blocks of H and use the active set method for the other half.

3.3.2.2 Generalized KL-divergence

Now we derive the update for one block of H when the generalized KL-divergence is used as the loss function. We refer to this type of NMF as *KLD-NMF*. When W is fixed, minimizing G_J can be expressed as follows:

$$\min_{H^{(J)}} G_J = \min_{H^{(J)}} \sum_I \sum_{i \in I, j \in J} (A_{ij} \log \frac{A_{ij}}{[WH]_{ij}} - A_{ij} + [WH]_{ij}).$$

We also leverage gradient descent to update $H^{(J)}$:

$$H^{(J)}_{uv} = H^{(J)}_{uv} - \eta_{uv} \frac{\partial G_J}{\partial H^{(J)}_{uv}},$$

where $\frac{\partial G_J}{\partial H^{(J)}_{uv}} = \sum_I \sum_{i \in I} [W_{iu} - W_{iu} \frac{A_{iv}}{[WH]_{iv}}]$. Again we derive step sizes by following Lee and Seung's approach: $\eta_{uv} = \frac{H^{(J)}_{uv}}{\sum_I \sum_{i \in I} W_{iu}}$.

Then, we have:

$$H^{(J)} = H^{(J)} * \frac{\sum_I (W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)} H^{(J)}}}{\sum_I (W^{(I)})^T E^{(I,J)}}, \tag{3.9}$$

where $E^{(I,J)}$ is a $a \times b$ matrix with all the elements being 1 (a is the number of rows in $W^{(I)}$ and b is the number of columns in $H^{(J)}$).

Similarly, we can derive the update formula for $W^{(I)}$:

$$W^{(I)} = W^{(I)} * \frac{\sum_J \frac{A^{(I,J)}}{W^{(I)} H^{(J)}} (H^{(J)})^T}{\sum_J E^{(I,J)} (H^{(J)})^T}. \tag{3.10}$$

3.4 Update Frequency

Block-wise updates can handle each block of one factor matrix independently. This flexibility allows us to have different ways to udpate blocks. We can simultaneously update all the blocks of one factor matrix and then update all the blocks of the other factor matrix. Also, we can update a subset of blocks of one factor matrix and then update a subset of blocks of the other one, and the number of blocks in the subset is adjustable. Furthermore, block-wise updates also facilitate distributed implementations. Different blocks of one factor matrix can be updated in parallel, and can be distributed in memories of all the machines and thus avoid overflowing the memory of one single machine (when there are large factor matrices). Storing factor matrices in memory supports random access and local aggregation, which are highly useful when updating them.

3.4.1 Concurrent Block-wise Updates

With block-wise updates, one basic way of fulfilling the block coordinate descent rules is to alternatively update all the blocks of H and all the blocks of W. Since this way updates all the blocks of one factor matrix concurrently, we refer to it as *concurrent block-wise updates*.

From the matrix operation perspective, we can show concurrent block-wise updates derived in the previous section are equivalent to the multiplicative update approach. Take SED-NMF for example. We can show that updates in Eq. (3.7) and Eq. (3.8) are equivalent to those in Eq. (3.2). Without loss of generality, we assume that the $H^{(J)}$ is one block of H from the J_0th column to the J_bth column. Let Y be one block of W^TWH from the J_0th column to the J_bth column, then we have $Y = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$, since $W^TW = \sum_I (W^{(I)})^T W^{(I)}$. Assuming that X is one block of W^TA from the J_0th column to the J_bth column, we can show that $X = \sum_I (W^{(I)})^T A^{(I,J)}$. Hence, for both concurrent block-wise updates and the multiplicative update approach, the formula for updating $H^{(J)}$ is equivalent to $H^{(J)} = H^{(J)} * \frac{X}{Y}$. That is, Eq. (3.7) is equivalent to the formula for updating H in Eq. (3.2). Similarly, we can show that Eq. (3.8) is equivalent to the formula for updating W.

3.4.2 Frequent Block-wise Updates

Since all the blocks of one factor matrix can be updated independently when the other matrix is fixed, another (more general) way of fulfilling block coordinate descent rules is to update a subset of blocks of H, and then update a subset of blocks of W. Since this way updates a factor matrix more frequently, we refer to it as *frequent block-wise updates*. Frequent block-wise updates aim to utilize the most recently updated data whenever possible, and thus can potentially accelerate convergence.

More formally, frequent block-wise updates start with some initial guess of W and H, and then seek to minimize the loss function by iteratively applying the following two steps:

Step I: Fix W, update a subset of blocks of H.

Step II: Fix H, update a subset of blocks of W.

In both steps, the subset's size is a parameter, and we rotate the subset on all the blocks to guarantee that each block has an equal chance to be updated. The subset's size controls the update frequency. In an extreme case, if we always set the subset to include all the blocks, frequent updates degrade to concurrent updates.

Frequent block-wise updates provide a high flexibility to update factor matrices. For simplicity, we update a subset of blocks of one factor matrix and then update all the blocks of the other one in each iteration. Here, we assume that we update a subset of blocks of W and then update all the blocks of H. Intuitively, updating H frequently might incur a large additional overhead. Fortunately, we next show that the formula for updating H can be incrementally computed. That is, the cost of updating H grows linearly with the number of W blocks that have been updated in the previous iteration.

3.4.3 Incremental Computation

In order to update H, we need to compute certain global statistics over all the blocks of W. This is because one block of H is related to all the blocks of W when calculating the loss function. For example, when calculating G_J (defined in Eq.(3.5)), a particular block of H (i.e., $H^{(J)}$) and all the blocks of W are involved. The global statistics over all the blocks of W can be expressed as a summation of local statistics over each individual block of W. If a block does not change, the corresponding local statistics does not change as well. As a result, if caching the local statistics for all the blocks, we do not need to recompute them for unchanged blocks. In this way,

unnecessary operations can be avoided. Furthermore, we can also cache the global statistics. Then, we can refresh it by accumulating the old value and the changes on local statistics. Next, we introduce incremental computation for SED-NMF and KLD-NMF, respectively, through identifying global statistics and local statistics.

3.4.3.1 Incremental Computation for SED-NMF

For SED-NMF, in order to incrementally update H when a subset of W blocks are updated, we introduce a few auxiliary matrices. Let $X^J = \sum_I (W^{(I)})^T A^{(I,J)}$, $X_I^J = (W^{(I)})^T A^{(I,J)}$, $S = \sum_I (W^{(I)})^T W^{(I)}$, and $S_I = (W^{(I)})^T W^{(I)}$. Among them, X^J and S can be considered as global statistics, and X_I^J and S_I can be seen as local statistics. Then, $H_{uv}^{(J)}$ can be updated by

$$H_{uv}^{(J)} = H_{uv}^{(J)} \frac{X_{uv}^J}{[SH^{(J)}]_{uv}}. \tag{3.11}$$

We next show how to incrementally calculate X_J and S by saving their values from last iteration. When a subset of $W^{(I)}$ ($I \in C$) have been updated, the new value of X_J and S can be computed as follows:

$$X_J = X_J + \sum_{I \in C} [(W^{(I)new})^T A^{(I,J)} - X_I^J]; \tag{3.12}$$

$$S = S + \sum_{I \in C} [(W^{(I)new})^T W^{(I)new} - S_I]. \tag{3.13}$$

From Eq. (3.11), Eq. (3.12), and Eq. (3.13), we can see that the cost of incrementally updating $H^{(J)}$ depends on the number of W blocks that have been updated rather than the total number of blocks that W has.

3.4.3.2 Incremental Computation for KLD-NMF

For KLD-NMF, we also introduce a few auxiliary matrices to incrementally update H when a subset of W blocks are updated. Let $X^J = \sum_I [(W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}]$, $X_I^J = (W^{(I)})^T \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}$, $S = \sum_I [(W^{(I)})^T E^{(I,J)}]$ (S is a vector), and $S_I = (W^{(I)})^T E^{(I,J)}$. Again, X^J and S can be considered as global statistics, and X_I^J and S_I can be seen as local statistics. Then, $H_{uv}^{(J)}$ can be updated by

$$H_{uv}^{(J)} = H_{uv}^{(J)} \frac{X_{uv}^J}{S_u}. \tag{3.14}$$

We next show how to incrementally calculate X_J and S by saving their values from last iteration. When a subset of $W^{(I)}$ ($I \in C$) have been updated, the new value of X_J and S can be computed as follows:

$$X_J = X_J + \sum_{I \in C} [(W^{(I)new})^T \frac{A^{(I,J)}}{W^{(I)new}H^{(J)}} - X_I^J]; \tag{3.15}$$

$$S = S + \sum_{I \in C} [(W^{(I)new})^T E^{(I,J)} - S_I]. \tag{3.16}$$

From Eq. (3.14), Eq. (3.15), and Eq. (3.16), we can again observe that the cost of incrementally updating $H^{(J)}$ depends on the number of W blocks that have been updated rather than the total number of W blocks.

3.4.4 Convergence of Frequent Block-wise Updates

Frequent block-wise updates maintain the convergence property. We here use SED-NMF as an instance. The proof for KLD-NMF can be derived similarly. For SED-NMF, we first prove that G_J and F_I are nonincreasing under formulae Eq. (3.7) and Eq. (3.8), respectively (as stated in the following two lemmas). We then show the overall loss function L is nonincreasing when frequent block-wise updates are applied.

Lemma 3.4.1. *G_J is nonincreasing under formula Eq. (3.7). G_J is constant if and only if $H^{(J)}$ is at a stationary point of G_J.*

Lemma 3.4.2. *F_I is nonincreasing under formula Eq. (3.8). F_I is constant if and only if $W^{(I)}$ is at a stationary point of F_I.*

Utilizing the concept of auxiliary functions [50], we can prove the above two lemmas. Then, we have the following theorem.

Theorem 3.4.3. *L is nonincreasing when frequent block-wise updates are applied. L is constant if and only if W and H are at a stationary point of L.*

Proof. As illustrated in Eq. (3.6), the overall loss function L is the sum of local loss functions, G_J or F_I. G_J is nonincreasing when $H^{(J)}$ is updated for any J. F_I is nonincreasing as well when $W^{(I)}$ is updated for any I. Therefore, frequent block-wise updates will not increase L when W (or H) is updated, no matter how many blocks of W (or H) are selected for updating in each iteration. Additionally, if and only if all the blocks of W (or H) are at a stationary point of L, L does not decrease. □

3.5 Implementations on Distributed Frameworks

MapReduce [22] and its extensions (e.g, [105]) have emerged as distributed frameworks for data intensive computation. MapReduce expresses a computation task as a series of jobs. Each job typically has one map operation (mapper) and one reduce operation (reducer). In this section, we illustrate the efficient implementation of concurrent block-wise updates on MapReduce. Also, we show how to implement frequent block-wise updates on an extended version of MapReduce, iMapReduce [105], which supports iterative computations more efficiently. To ground our discussion, we begin with an overview of the state-of-the-art work that implements the traditional form of update functions on MapReduce.

3.5.1 Traditional Updates on MapReduce

The previous effort by Liu et al. [57] is a piece of state-of-the-art work of implementing the traditional form of update functions on MapReduce. For performing matrix multiplication (with two large matrices), it needs to join a row (or column) of one matrix with each column (or row) of the other one with two MapReduce jobs. As a result, a huge amount of intermediate data have to be generated and shuffled. The intermediate data explosion is a huge issue in terms of performance.

In order to elaborate the intermediate data explosion issue of implementing the traditional form of update functions, we take SED-NMF as an instance. To implement the update for H (as shown in Eq. (3.2)) on MapReduce, the previous work [57] needs five jobs: two jobs for computing W^TA, two jobs for computing W^TWH, and one job for the final update. Among them, the two jobs for computing W^TA are the bottleneck. The first job generates the intermediate data $< j, A_{i,j}W_{i\cdot}^T >$ for any i and $j \in \mathbb{O}^i$, where $\mathbb{O}^i$ denotes the set of nonzero elements on the ith row of A. The second job takes the intermediate data as its input. The intermediate data take $O(\rho mnk)$ space (where ρ is the sparsity of A), which can be huge considering m and n are at the order of hundreds of thousands or even millions. Consequently, dumping and loading the intermediate data dominate the time of updating H. Similar conclusion can be reached for updating W.

3.5.2 Concurrent Block-wise Updates on MapReduce

Block-wise updates enable efficient distributed implementations. With block-wise updates, the basic computation units in the update functions (e.g., Eq. (3.7) and Eq. (3.8)) are blocks of factor matrices and blocks of the original matrix. The size of a block is adjustable. As a result, when performing an essential matrix operation, which involves two blocks of matrices (e.g., $(W^{(I)})^T$ and $A^{(I,J)}$), we can assume that at least the smaller block can be held in the memory of a single worker. Since W and

H are low-rank factor matrices, they usually are much smaller than A, and thus the assumption that one of their blocks can be held in the memory of one single worker is reasonable. The result matrix of an essential matrix operation (e.g., $(W^{(I)})^T A^{(I,J)}$) is usually relatively small and can be held in the memory of one single worker as well. Storing a matrix (or a block of a matrix) in memory efficiently supports random and repeated access, which is commonly needed in a matrix operation such as multiplication. Maintaining the result matrix in memory supports local aggregation. Therefore, one worker can complete an essential matrix operation locally and efficiently. Note that the other (larger) matrix (e.g., one block of A) is still in disk so as to scale to large NMF problems.

Accordingly, the MapReduce programming model fits block-wise updates well. An essential matrix operation with two blocks can be realized in one mapper, and the aggregation of the results of essential matrix operations can be realized in reducers. To realize matrix multiplication with two blocks of matrices in one mapper, we exploit the fact that a mapper can cache data in memory before processing input key-value pairs and that a mapper can maintain state across the processing of multiple input key-value pairs and defer emission of intermediate key-value pairs until all the input pairs have been processed. We next illustrate efficient implementations of concurrent block-wise updates for SED-NMF and KLD-NMF, respectively.

3.5.2.1 MapReduce Implementation for SED-NMF

Inspired by the previous work [57], which decomposes the update formula of SED-NMF for H into three components, we consider the update formula for $H^{(J)}$ (Eq. (3.7)) into three parts as well: $X^{(J)} = \sum_I (W^{(I)})^T A^{(I,J)}$, $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$, and $H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$. However, we have much more efficient implementation for each part than the previous work, as demonstrated in the following.

We have one job to compute $X^{(J)} = \sum_I X^{(I,J)} = \sum_I (W^{(I)})^T A^{(I,J)}$. Let $X^{(I,J)}_{\cdot j}$ represent the jth column of $X^{(I,J)}$, then

$$X^{(I,J)}_{\cdot j} = \sum_{i=1}^{a} A^{(I,J)}_{i,j} (W^{(I)}_{i\cdot})^T, \tag{3.17}$$

where a is the number of rows of $A^{(I,J)}$, and $W^{(I)}_{i\cdot}$ is the ith row of $W^{(I)}$. When holding $W^{(I)}$ in memory, a mapper can leverage Eq. (3.17) to compute $X^{(I,J)}$ via continuously reading elements of $A^{(I,J)}$. $X^{(I,J)}$ (which is usually small) stays in memory for local aggregation. After computing $X^{(I,J)}$ in the mapper, the aggregation $X^{(J)} = \sum_I X^{(I,J)}$ can be computed in a reducer. Different reducers compute $X^{(J)}$ for different J.

Two jobs are used to compute $Y^{(J)} = \sum_I (W^{(I)})^T W^{(I)} H^{(J)}$. We first compute $S = \sum_I (W^{(I)})^T W^{(I)}$ and then calculate $Y^{(J)} = SH^{(J)}$. $(W^{(I)})^T W^{(I)}$ (a $k \times k$ matrix) can be performed in one mapper as follows:

$$(W^{(I)})^T W^{(I)} = \sum_{i=1}^{a} (W^{(I)}_{i\cdot})^T W^{(I)}_{i\cdot}. \tag{3.18}$$

Then, all mappers send $(W^{(I)})^T W^{(I)}$ to one particular reducer for a global summation. After computing $S = \sum_I (W^{(I)})^T W^{(I)}$, calculating $Y^{(J)} = SH^{(J)}$ can be done in a job with the map phase only, by $Y^{(J)}_{\cdot j} = SH^{(J)}_{\cdot j}$.

Last, we have one job (with the map phase only) to compute $H^{(J)} \leftarrow H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$. In summary, the MapReduce operations for updating H are as follows.

- Job-I Map: Load $W^{(I)}$ in memory, calculate $X^{(I,J)}$ using Eq. (3.17) (take $A^{(I,J)}$ as input), and emit $< I, X^{(I,J)} >$.
- Job-I Reduce: Take $< I, X^{(I,J)} >$, and emit $< J, X^{(J)} >$.
- Job-II Map: Load $W^{(I)}$ in memory, calculate $(W^{(I)})^T W^{(I)}$ using Eq. (3.18), and emit $< I, (W^{(I)})^T W^{(I)} >$.
- Job-II Reduce: Take $< I, (W^{(I)})^T W^{(I)} >$, and emit $< 0, S >$.
- Job-III Map: Load S in memory. Emit tuples $< j, Y^{(J)}_{\cdot j} >$.

- Job-IV Map: Read $< j, H_{\cdot j}^{(J)} >$, $< j, X_{\cdot j}^{(J)} >$, and $< j, Y_{\cdot j}^{(J)} >$. Emit tuples in the form of $< j, H_{\cdot j}^{(J)new} >$, where $H_{\cdot j}^{(J)new} = H_{\cdot j}^{(J)} * \frac{X_{\cdot j}^{(J)}}{Y_{\cdot j}^{(J)}}$.

In the previous implementation, we try to minimize data shuffling by utilizing local aggregation. However, in each iteration it still needs four MapReduce jobs to update H. In addition, intermediate data (e.g., $X^{(J)}$) need to be dumped into disk and be reloaded in later jobs. We next illustrate how to minimize the number of jobs and the amount of intermediate data to be reloaded.

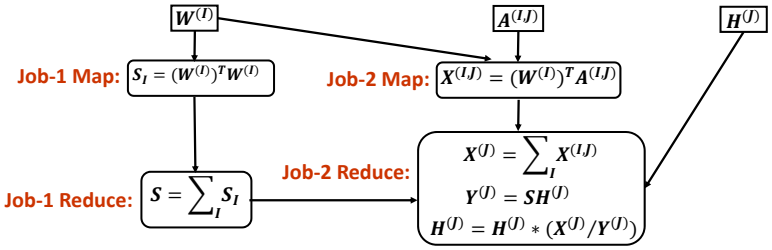


Figure 3.2. Overview of the optimized implementation for updating $H^{(J)}$ of SED-NMF on MapReduce.

Job-II can be kept (as Job-1), since it only produces a small $(k \times k)$ matrix and reloading its output does not take much time. Job-I, Job-III, and Job-IV can be integrated into one job so as to avoid dumping and reloading $X^{(J)}$ and $Y^{(J)}$. The integrated job has the same map phase with Job-I. In the reduce phase, besides computing $X_{\cdot j}^{(J)}$, it also computes $Y_{\cdot j}^{(J)}$ and finally calculates $H_{\cdot j}^{(J)new} = H_{\cdot j}^{(J)} * [X_{\cdot j}^{(J)} / Y_{\cdot j}^{(J)}]$. The overview of our optimized implementation is presented in Figure 3.2, and the MapReduce operations in the integrated job (Job-2) are described as follows.

- Job-2 Map: Load $W^{(I)}$ in memory, calculate $X^{(I,J)}$ using Eq. (3.17) (take $A^{(I,J)}$ as input), and emit $< I, X^{(I,J)} >$.
- Job-2 Reduce: Take $< I, X^{(I,J)} >$, and first calculate $X_{\cdot j}^{(J)}$. Load S in memory. Then, read $H_{\cdot j}^{(J)}$ and compute $Y_{\cdot j}^{(J)}$. Last, calculate $H_{\cdot j}^{(J)new}$.

In the above, we describe the MapReduce operations used to complete the update of H for one iteration. Updating W can be performed in the same fashion. We next provide a sketch of its design and omit the description of the operations.

The formula for updating W (Eq. (3.8)) can also be treated as three parts: $U^{(I)} = \sum_J A^{(I,J)}(H^{(J)})^T$, $V^{(I)} = \sum_J W^{(I)}H^{(J)}(H^{(J)})^T$, and $W^{(I)} = W^{(I)} * \frac{U^{(I)}}{V^{(I)}}$. Let $U^{(I,J)} = A^{(I,J)}(H^{(J)})^T$, then

$$U_{i\cdot}^{(I,J)} = \sum_{j=1}^{a} A_{i,j}^{(I,J)}(H_{\cdot j}^{(J)})^T. \tag{3.19}$$

To efficiently compute $V^{(I)}$, we compute $H^{(J)}(H^{(J)})^T$ first in the following way:

$$H^{(J)}(H^{(J)})^T = \sum_{j=1}^{b} H_{\cdot j}^{(J)}(H_{\cdot j}^{(J)})^T. \tag{3.20}$$

3.5.2.2 MapReduce Implementation for KLD-NMF

For KLD-NMF, we also decompose the update formula for $H^{(J)}$ (Eq. (3.9)) into three parts: $X^{(J)} = \sum_I[(W^{(I)})^T\frac{A^{(I,J)}}{W^{(I)}H^{(J)}}]$, $Y^{(J)} = \sum_I[(W^{(I)})^T E^{(I,J)}]$, and $H^{(J)} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}}$.

We use one job to compute $X^{(J)} = \sum_I[(W^{(I)})^T\frac{A^{(I,J)}}{W^{(I)}H^{(J)}}]$. Let $X_{\cdot j}^{(I,J)}$ represent the jth column of $X^{(I,J)}$, then

$$X_{\cdot j}^{(I,J)} = \sum_{i=1}^{a}(W_{i\cdot}^{(I)})^T\frac{A_{i,j}^{(I,J)}}{W_{i\cdot}^{(I)}H_{\cdot j}^{(J)}}. \tag{3.21}$$

When holding $W^{(I)}$ and $H^{(J)}$ in memory, a mapper can leverage Eq. (3.21) to compute $X^{(I,J)}$ via continuously reading elements of $A^{(I,J)}$. $X^{(I,J)}$ stays in memory for local aggregation. After computing $X^{(I,J)}$ in the mapper, the aggregation $X^{(J)} = \sum_I X^{(I,J)}$ can be computed in a reducer.

One job is used to compute $Y^{(J)} = \sum_I [(W^{(I)})^T E^{(I,J)}]$. Let $Y^{(I,J)} = (W^{(I)})^T E^{(I,J)}$. Computing $Y^{(I,J)}$ seems time-consuming because it multiplies two dense matrices. But since all elements of $E^{(I,J)}$ is 1, all the columns of $Y^{(I,J)}$ are the same. Therefore, we actually only need to calculate one column of $Y^{(I,J)}$. For example,

$$Y_{\cdot j}^{(I,J)} = \sum_{i=1}^{a} (W_{i\cdot}^{(I)})^T. \tag{3.22}$$

After computing $Y^{(I,J)}$ in the mapper, the aggregation $Y^{(J)} = \sum_I Y^{(I,J)}$ can be computed in a reducer.

The formula for updating W (Eq. (3.10)) can also be treated as three parts: $U^{(I)} = \sum_J \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}(H^{(J)})^T$, $V^{(I)} = \sum_J E^{(I,J)}(H^{(J)})^T$, and $W^{(I)} = W^{(I)} * \frac{U^{(I)}}{V^{(I)}}$. Let $U^{(I,J)} = \frac{A^{(I,J)}}{W^{(I)}H^{(J)}}(H^{(J)})^T$, then

$$U_{i\cdot}^{(I,J)} = \sum_{j=1}^{b} \frac{A_{i,j}^{(I,J)}}{W_{i\cdot}^{(I)} H_{\cdot j}^{(J)}} (H_{\cdot j}^{(J)})^T. \tag{3.23}$$

Let $V^{(I,J)} = E^{(I,J)}(H^{(J)})^T$, and it can be calculated in the following way:

$$V_{i\cdot}^{(I,J)} = \sum_{j=1}^{b} (H_{\cdot j}^{(J)})^T. \tag{3.24}$$

Then $V^{(I)}$ can be computed through $V^{(I)} = \sum_J V^{(I,J)}$.

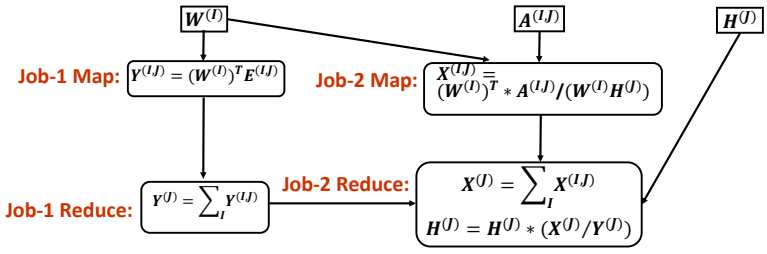


Figure 3.3. Overview of the implementation for updating $H^{(J)}$ of KLD-NMF on MapReduce.

After decomposing the update formulae for KLD-NMF, the MapReduce operations can be easily derived by following the way of achieving operations for SED-NMF. The overview of these operations is shown in Figure 3.3, while the details are omitted.

3.5.2.3 Analysis

The intermediate data and the memory usage of implementing concurrent block-wise updates on MapReduce are analyzed here. Assume that W has c blocks and H has d blocks. Take SED-NMF as an example. Similar conclusion can be obtained for KLD-NMF. We first analyze the intermediate data. For updating H, the main intermediate data it generates are $X^{(I,J)}$ (for any I and J, cd copies in total), which take $O(k\frac{n}{d})$ space. Therefore, the main intermediate data take $O(knc)$ space in total. Similarly, we can show that the main intermediate data of updating W take $O(kmd)$ space in total. We can control the values of c and d and typically have $c \ll m$ and $d \ll n$. Therefore, the implementation of concurrent updates doest not suffer from the intermediate data explosion issue, and is much more efficient than the implementation of the traditional form of updates.

We then analyze the memory usage. For updating H, the main memory usage happens in the map phase. A mapper at most needs to cache $W^{(I)}$ and $X^{(I,J)}$ in memory, which take $O(k\frac{m}{c} + k\frac{n}{d})$ space. Similarly, we can show that for updating W a mapper at most needs $O(k\frac{m}{c} + k\frac{n}{d})$ memory space as well. We know k is typically small (since NMF is a low-rank approximation). Therefore, for m and n even at the order of millions, a commodity server does not have the memory overflow problem.

3.5.3 Frequent Block-wise Updates on iMapReduce

Although frequent updates have potential to speed up NMF, parallelizing frequent updates in a distributed environment is challenging. Computations such as global summations need to be done in a centralized way. When processing the distributed

blocks of factor matrices, the system has to synchronize the global summations frequently. Synchronizing the global resources in a distributed environment may result in considerable overhead, especially on MapReduce. MapReduce starts a new job for each computation errand. Each job needs to be initialized and load its input data, even when the data are from a previous job. Frequent updates bring more jobs. As a result, the initialization overhead and the cost of repeatedly loading data may vanish the benefit of frequent updates.

In this subsection, we propose an implementation of frequent block-wise updates on iMapReduce [105], which uses persistent mappers and reducers to avoid job initialization overhead. Each mapper is paired with one reducer. One pair of mapper and reducer can be seen as one logical worker. Data shuffling between mappers and reducers is the same with that of MapReduce. In addition, a reducer of iMapReduce can redirect its output to its paired mapper. Since mappers and reducers are persistent, data can be maintained in memory across different iterations, and thus can avoid repeatedly loading data. As a result, iMapReduce decreases the overhead of frequent block-wise updates. Therefore, it provides frequent block-wise updates with an opportunity to achieve good performance.

We implement frequent block-wise updates on iMapReduce in the following way. H is evenly split into r blocks, and W is evenly partitioned into $p * r$ blocks, where r is the number of workers and p is a parameter used to control update frequency. Each worker handles p blocks of W and one block of H. In each iteration, a worker updates its H block and one selected W block. That is, there are r blocks of W in total to be updated in each iteration. Each worker rotates the selected W block on all its W blocks. The setting of p plays an important role on frequent block-wise updates. Setting p too large may incur considerable overhead for synchronization. Setting it too small may degrade the effect of the frequent updates. In an extreme case, we can set $p = 1$, then frequent block-wise updates degrade to concurrent block-wise updates.

Nevertheless, we will show in experiments (Section 3.6.4) that a quite large range of p can enable frequent block-wise updates to have better performance than concurrent block-wise updates. The operations of iMapReduce are as follows. Note that Map-1x represents different stages of a mapper, and Reduce-1x represents different stages of a reducer.

3.5.3.1 iMapReduce Implementation for SED-NMF

We first show how to implement frequent updates for SED-NMF on iMapReduce.

- Map-1a: Load a subset (i.e., p) of W blocks (e.g., $(W^{(B)new})$) in memory (1st iteration only) or receive one updated W block from last iteration. For all loaded or received blocks, compute S_l via $S_l = \sum_B (W^{(B)new})^T W^{(B)new}$ (1st iteration) or $S_l = S_l + ((W^{(B)new})^T W^{(B)new} - (W^{(B)})^T W^{(B)})$, and replace $W^{(B)}$ with $W^{(B)new}$. Broadcast $< d, S_l >$ to all reducers, where d is the corresponding reducer ID.
- Reduce-1a: Take $< d, S_l >$, compute $S = \sum_l S_l$, and store S in memory.
- Map-1b: For each loaded/received W block in the previous phase (e.g., $(W^{(B)new})$), read $A^{(B,J)}$ and emit tuples in the form of $< B, X^{(B,J)} >$ where $X^{(B,J)}$ is calculated using Eq. (3.17) (1st iteration) or in the form of $< B, \Delta X^{(B,J)} >$ where $\Delta X^{(B,J)} = (W^{(B)new})^T A^{(B,J)} - X^{(B,J)}$.
- Reduce-1b: Take $< B, X^{(B,J)} >$ and calculate $X^{(J)} = \sum_B X^{(B,J)}$ (1st iteration) or take $< B, \Delta X^{(B,J)} >$ and calculate $X^{(J)} = X^{(J)} + \sum_B \Delta X^{(B,J)}$. Then, load $H^{(J)}$ into memory (1st iteration) and compute $Y^{(J)} = SH^{(J)}$. Last, calculate $H^{(J)new}$ by $(H^{(J)new} = H^{(J)} * \frac{X^{(J)}}{Y^{(J)}})$, store it in memory, and pass one copy to Map-1c in the form of $< J, H^{(J)new} >$.
- Map-1c: Receive (updated) $H^{(J)}$ from Reduce-1b. Broadcast $< J, H^{(J)}(H^{(J)})^T >$ to all reducers.
- Reduce-1c: Take $< J, H^{(J)}(H^{(J)})^T >$, compute $Z = \sum_J H^{(J)}(H^{(J)})^T$, and store Z in memory.

- Map-1d: For a W block that is selected in current iteration (e.g., $(W^{(B)})$), read $A^{(B,J)}$ and emit tuples in the form of $< J, U^{(B,J)} >$, where $U^{(B,J)}$ is calculated using Eq. (3.19).
- Reduce-1d: Take $< J, U^{(B,J)} >$, and calculate $U^{(B)} = \sum_J U^{(B,J)}$. Then, compute $V^{(B)} = W^{(I)}Z$. Last, calculate $W^{(B)new} = W^{(B)} * \frac{U^{(B)}}{V^{(B)}}$, store it in memory, and pass one copy to Map-1a.

3.5.3.2 iMapReduce Implementation for KLD-NMF

We then show how to implement frequent updates for KLD-NMF on iMapReduce.

- Map-1a: Load a subset (i.e., p) of W blocks (e.g., $(W^{(B)new})$) in memory (1st iteration only) or receive one updated W block from last iteration. For all loaded or received blocks, compute S_l via $S_l = (W^{(B)new})^T E^{(B,J)}$ (1st iteration) or $S_l = S_l + ((W^{(B)new})^T E^{(B,J)} - (W^{(B)})^T E^{(B,J)})$, and replace $W^{(B)}$ with $W^{(B)new}$. Broadcast $< d, S_l >$ to all reducers, where d is the corresponding reducer ID.
- Reduce-1a: Take $< d, S_l >$, compute $S = \sum_l S_l$, and store S in memory.
- Map-1b: For each loaded/received W block in the previous phase (e.g., $(W^{(B)new})$), read $A^{(B,J)}$ and $H^{(J)}$, and then emit tuples in the form of $< B, X^{(B,J)} >$ with $X^{(B,J)}$ computed by Eq. (3.21) (1st iteration) or in the form of $< B, \Delta X^{(B,J)} >$ where $\Delta X^{(B,J)} = (W^{(B)new})^T \frac{A^{(B,J)}}{W^{(B)new} H^{(J)}} - X^{(B,J)}$.
- Reduce-1b: Take $< B, X^{(B,J)} >$ and calculate $X^{(J)} = \sum_B X^{(B,J)}$ (1st iteration) or take $< B, \Delta X^{(B,J)} >$ and calculate $X^{(J)} = X^{(J)} + \sum_B \Delta X^{(B,J)}$. Then, calculate $H^{(J)new}$ by $(H^{(J)new} = H^{(J)} * \frac{X^{(J)}}{S})$, store it in memory, and pass one copy to Map-1c in the form of $< J, H^{(J)new} >$.
- Map-1c: Receive (updated) $H^{(J)}$ from Reduce-1b. Broadcast $< J, E^{(I,J)}(H^{(J)})^T >$ to all reducers.
- Reduce-1c: Take $< J, E^{(I,J)}(H^{(J)})^T >$, compute $Z = \sum_J E^{(I,J)}(H^{(J)})^T$, and store Z in memory.

- Map-1d: For a W block that is selected in current iteration (e.g., $(W^{(B)})$), read $A^{(B,J)}$ and $H^{(J)}$, and then emit tuples in the form of $< J, U^{(B,J)} >$, where $U^{(B,J)}$ is calculated using Eq. (3.23).
- Reduce-1d: Take $< J, U^{(B,J)} >$, and calculate $U^{(B)} = \sum_J U^{(B,J)}$. Then, calculate $W^{(B)new} = W^{(B)} * \frac{U^{(B)}}{Z}$, store it in memory, and pass one copy to Map-1a.

We can show that our implementation of frequent block-wise updates takes only $O(km + kn)$ aggregate memory of the cluster for either SED-NMF or KLD-NMF. Since k is typically small, even a small cluster of commodity servers can handle the NMF problem with m and n at the order of millions without memory overflow.

3.6 Evaluation

In this section, we evaluate the effectiveness and efficiency of block-wise updates on both synthetic and real-word datasets. For MapReduce, we use its open source implementation, Hadoop [2]. Experiments are performed on both small-scale and large-scale clusters.

3.6.1 Experiment Setup

We build both a small-scale cluster of local machines and a large-scale cluster on the Amazon EC2 cloud [1]. The local cluster consists of 4 machines, and each one has dual-core 2.66GHz CPU, 4GB of RAM, 1TB hard disk. These 4 machines are connected through a switch with a bandwidth of 1Gbps. The Amazon cluster consists of 100 medium instances, and each instance has one core, 3.7GB of RAM, and 400GB of hard disk.

Table 3.2. Dataset Summary

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	$0.1 * m * n$

Both synthetic and real-word datasets are used in our experiments. We use two real-world datasets. One is a document-term matrix, NYTimes, from UCI Machine Learning Repository [3]. The other one is a user-movie matrix from the Netflix prize [44]. We also generate several matrices with different choices of m (the number of rows) and n (the number of columns). The sparsity is set to 0.1, and each element is a random integer number uniformly selected from range 1 to 5. The datasets are summarized in Table 3.2.

Unless otherwise specified, we use rank $k = 10$, and use $p = 8$ for frequent block-wise updates (which means each worker updates $\frac{1}{8}$ of its W blocks in each iteration).

3.6.2 Comparison with Existing Work

The first set of experiments focus on demonstrating the advantage of our (optimized) implementation of concurrent block-wise updates on MapReduce. We compare it with a piece of state-of-the-art work of implementing the traditional form of update functions, which is discussed in Section 3.5.1. The implementation of concurrent block-wise updates on iMapReduce is added into the comparison to show iMapReduce's superiority over MapReduce. For a comprehensive comparison, the iMapReduce implementation of the traditional form is taken into consideration as well. As described in Section 3.4.1, concurrent block-wise updates are equivalent to the multiplicative update, and thus we leverage the time taken in a single iteration to directly compare performance.

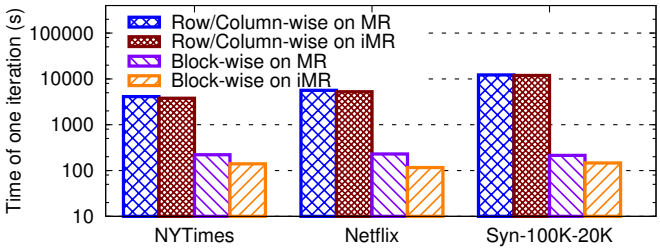


Figure 3.4. Time taken in one iteration for SED-NMF on the local cluster. The y-axis is in log scale.

Figure 3.4 shows the time taken in one iteration of all the four implementations for SED-NMF on both synthetic and real-word datasets. Note that the y-axis is in log scale. Our implementation on MapReduce (denoted by "Block-wise on MR") is 19x - 57x faster than the existing approach (denoted by "Row/Column-wise on MR"). Moreover, for the block-wise updates, the implementation on iMapReduce (denoted by "Block-wise on iMR") is up to 2x faster than that on MapReduce, since iMapReduce can eliminate the job initialization overhead and the cost of repeatedly dumping/loading factor matrices (note that the original matrix still needs to be loaded from the file system at each iteration). For the traditional form of update functions, the improvement by iMapReduce (denoted by "Row/Column-wise on iMR") is quite limited. The reasons are twofold. One reason is that its implementation does not store factor matrices in memory, and thus there is no benefit of eliminating the cost of repeatedly dumping/loading factor matrices. The other reason is that compared to the long running time of a job, the job initialization overhead is almost ignorable, and thus eliminating the job initialization overhead does not make a huge difference.

Figure 3.5 shows the time taken in one iteration of all the four implementations for KLD-NMF. Similar to SED-NMF, our implementation on MapReduce is 20x - 59x faster than the existing approach. Furthermore, for the block-wise updates, the implementation on iMapReduce is up to 3x faster than that on MapReduce; while for the traditional form of update functions, the improvement by iMapReduce is ignorable.

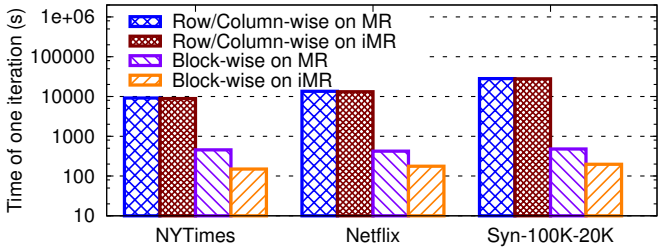
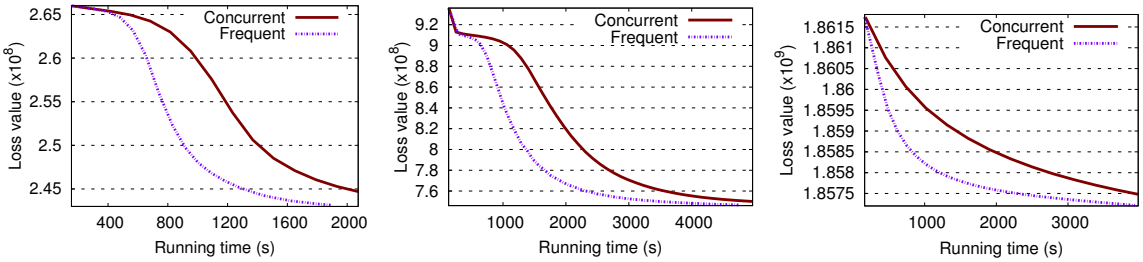


Figure 3.5. Time taken in one iteration for KLD-NMF on the local cluster. The y-axis is in log scale.

3.6.3 Effect of Frequent Updates

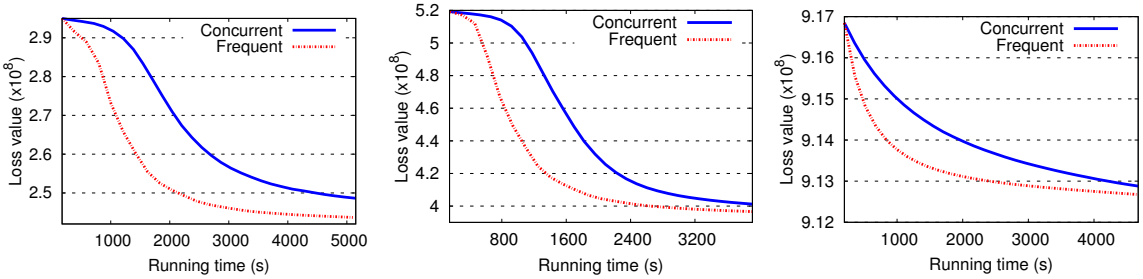
Frequent block-wise updates leverage more up-to-date H to update W than concurrent block-wise updates, since they update H more frequently. Therefore, they have the potential to reach the convergence criterion with less workload. To evaluate their effect, we compare frequent block-wise updates with concurrent block-wise updates when both implemented on iMapReduce.



(a) NYTimes (b) Netflix (c) Syn-100K-20K

Figure 3.6. Convergence speed of SED-NMF on the local cluster.

Both update approaches start with the same initial values when compared on the same dataset. Figure 3.6 plots the performance comparison for SED-NMF. We can see that frequent block-wise updates ("Frequent") converge faster than concurrent block-wise updates ("Concurrent") on all the three datasets. In other words, if we use a predefined loss value as the convergence criterion, frequent block-wise updates would have much shorter running time. Similar phenomena are observed for KLD-NMF, as shown in Figure 3.7.



(a) NYTimes (b) Netflix (c) Syn-100K-20K

Figure 3.7. Convergence speed of KLD-NMF on the local cluster.

3.6.4 Tuning Update Frequency

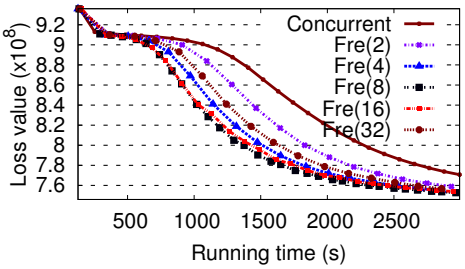


Figure 3.8. Convergence speed vs. update frequency. The numbers associated with "Fre" represent settings of p.

As stated in Section 3.5.3, the update frequency can make a huge impact on the performance of frequent block-wise updates. In experiments, we find that a quite large range of p can allow frequent block-wise updates to have better performance than their concurrent counterparts, and the best setting of p stays in the range from 4 to 16. That is also why we set $p = 8$ by default. For example, Figure 3.8 shows the convergence speed with different settings on dataset Netflix for SED-NMF. Another interesting finding is that if a setting is better during the first few iterations, it will continue to be better. Hence, another way of obtaining a good setting of p is to test several candidate settings, each for a few iterations, and then choose the best one. Similar trends are observed for KLD-NMF and are omitted here.

3.6.5 Different Data Sizes

We then measure how block-wise updates scale with increasing size of the original matrix A. We generate synthetic datasets of different sizes by fixing the number of $(100k)$ rows and increasing the number of columns. Figure 3.9 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates as the dataset size varies. The time of either implementation increases as the number of columns increases, and the time of the latter increases much faster. When the number of columns is $100k$, our implementation of block-wise updates is 90x faster

than the implementation of the traditional updates (compared to 57x speedup when the number of columns is $20k$).

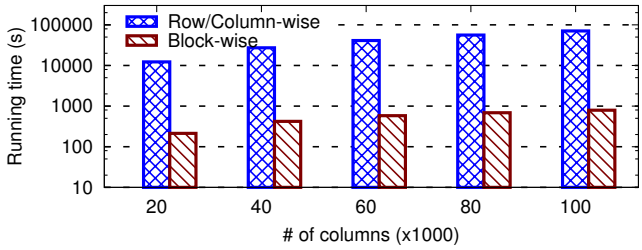


Figure 3.9. Comparing Row/Column-wise updates with block-wise updates through varying dataset size.

We next compare the running time of concurrent block-wise updates with that of frequent block-wise updates. We use the loss value when concurrent block-wise updates run for 25 iterations as the convergence point. Then the time used to reach this convergence point is measured as the running time. This criterion also applies to later comparisons. As presented in Figure 3.10, the running time of either type of updates increases sub-linearly with the size of the dataset. Moreover, frequent block-wise updates are up to 2.7x faster than concurrent block-wise updates. The results for KLD-NMF have similar trends and are omitted here.

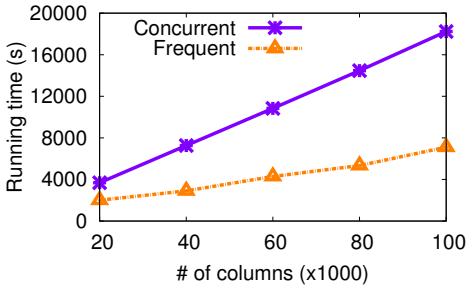


Figure 3.10. Comparing concurrent block-wise updates with frequent block-wise updates through varying dataset size.

3.6.6 Different Settings of Rank

We also measure how block-wise updates scale with different settings of the rank. We here present the results for SED-NMF (the performance comparison for KLD-NMF is similar). Figure 3.11 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates on dataset Syn-100K-20K as k varies from 10 to 50. It can be seen that the time of either implementation increases as k increases, and the time of the latter increases much faster. When $k = 50$, our implementation of block-wise updates is 107x faster than the implementation of the traditional updates (compared to 57x speedup when $k = 10$).

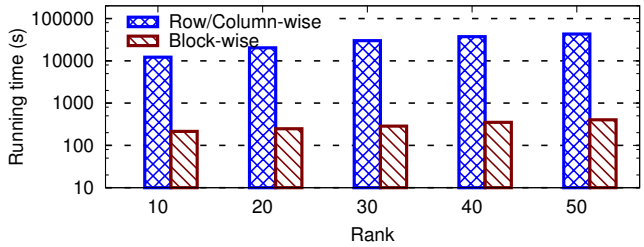


Figure 3.11. Time taken in one iteration vs. different settings of rank on the local cluster for SED-NMF on MapReduce.

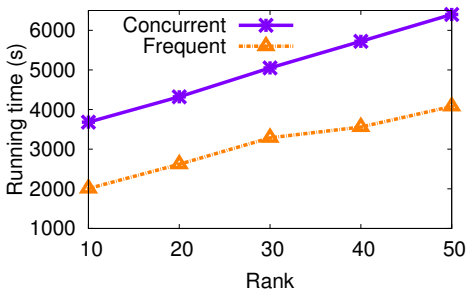


Figure 3.12. Running time vs. different settings of rank on the local cluster for SED-NMF on iMapReduce.

We then compare the running time of concurrent block-wise updates with that of frequent block-wise updates as k varies. As plotted in Figure 3.12, the running time of either type of updates increases sub-linearly with k. Furthermore, the running time of concurrent block-wise updates increases faster.

3.6.7 Scaling Performance

To validate the scalability of our implementations, we evaluate them on the Amazon EC2 cloud. The results of SED-NMF are reported. We use dataset Syn-1M-20K, which has 1 million rows, 20 thousand columns, and 2 billion nonzero elements. Figure 3.13 plots the time taken in a single iteration when all four implementations running on 100 nodes (i.e., instances). Our implementation on MapReduce is 23x faster than that of the existing approach. For block-wise updates, the implementation on iMapReduce is 1.5x faster than that on MapReduce.

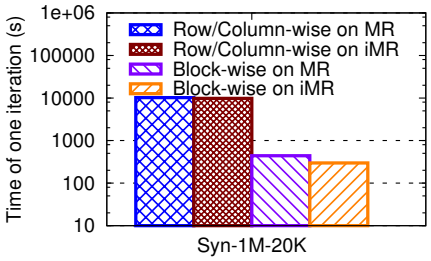


Figure 3.13. Time taken in one iteration for KLD-NMF on Amazon EC2 cloud. The y-axis is in log scale.

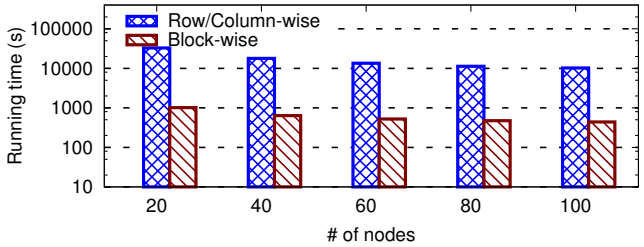


Figure 3.14. Scaling performance of MapReduce implementations on Amazon EC2 cloud. The y-axis is in log scale.

Figure 3.14 shows the time taken in one iteration of the block-wise updates and the traditional row/column-wise updates as the number of nodes being used increases from 20 to 100. The time of either implementation decreases as the number of nodes increases. Figure 3.15 compares the running time of concurrent block-wise updates with that of frequent block-wise updates as the number of nodes increases. We can see that the running time of either frequent block-wise updates or concurrent block-wise

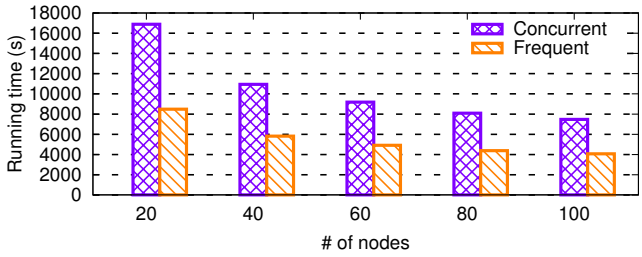


Figure 3.15. Scaling performance of iMapReduce implementations on Amazon EC2 cloud.

updates decreases smoothly as the number of nodes increases. In addition, frequent block-wise updates outperform concurrent block-wise updates with any number of nodes in the cluster.

3.7 Related Work

Matrix factorization has been applied very widely [16, 44, 67, 71, 87, 109]. Due to its popularity and increasingly larger datasets, many approaches for paralleling it have been proposed. Zhou et al. [110] and Schelter et al. [78] show how to distribute the alternating least squares algorithm for matrix factorization. Both approaches require that each worker has a copy of one factor matrix when the one is updated. This requirement limits its scalability. For large matrix factorization problems, it is important that factor matrices can be distributed. Several efforts handle matrix factorization using distributed gradient descent methods, which can distribute factor matrix updates across a cluster of machines [28, 53, 84, 97, 100]. These approaches mainly focus on in-memory implementations, in which both the original matrix and factor matrices are in the aggregate memory of the cluster, and use the forms of update functions that are different from our presented form. Additionally, our approach puts the original matrix on disk so as to scale to large NMF problems using commodity servers. A closely related work is from Liu et al. [57]. They propose a scheme of implementing the multiplicative update approach on MapReduce. Their scheme is

based on the traditional form of update functions and thus has the intermediate data explosion issue, which results in poor performance.

It has been shown that frequent updates can accelerate expectation maximization (EM) algorithms [18, 70, 85, 96]. Somewhat surprisingly, there has been no attempt to apply this method to NMF, even though there is equivalence between certain variations of NMF and some EM algorithms like K-means [24]. Our work demonstrates that frequent updates can also accelerate NMF.

3.8 Conclusion

In this chapter, we find that by leveraging a new form of factor matrix update functions, block-wise updates, we can perform local aggregation and thus have an efficient MapReduce implementation for NMF. Moreover, we propose frequent block-wise updates, which aim to use the most recently updated data whenever possible. As a result, frequent block-wise updates can further improve the performance, compared with concurrent block-wise updates. We implement concurrent block-wise updates on MapReduce and implement frequent block-wise updates on iMapReduce for two classical NMFs: one uses the square of Euclidean distance as the loss function, and the other uses the generalized KL-divergence. With both synthetic and real-world datasets, the evaluation results show that our iMapReduce implementation with frequent block-wise updates is up to two orders of magnitude faster than the existing MapReduce implementation with the traditional form of update functions.

CHAPTER 4

SCALABLE DISTRIBUTED BELIEF PROPAGATION WITH PRIORITIZED BLOCK UPDATES

4.1 Introduction

Probabilistic graphical models have been used for reasoning in a wide range of application domains [35, 43, 86, 99, 112]. Inference in these models, including marginalization and maximum a posteriori estimation, forms the basis of many statistical methods in knowledge management. Usually, exact inference in a probabilistic graphical model is NP-hard. As a result, there have been many approaches on introducing both variational and sampling approximations to inference. Among them, loopy belief propagation (BP) and its variants [39,72,82,92] are popular message passing methods for performing approximate inference.

It has been shown that the schedule for updating messages can make a huge difference to the running time of BP algorithms. Specifically, dynamic scheduling schemes, which determine the order of updating messages by the changes of message values, can significantly speedup BP algorithms [26, 29, 30, 83]. Although dynamic scheduling schemes have potential to speedup BP algorithms, existing ones cannot fully utilize the potential. Most of them typically select one message for updating each time, e.g., the message with the highest priority value. As a result, many operations need to be performed so as to select next message. That is, the cost of realizing such a dynamic scheduling scheme is high.

In this research, we propose to select a set of messages instead of a single one to update at a time. Hence, the amortized cost of selecting one message is low.

Moreover, a novel priority is leveraged to determine which messages are selected. In other words, we present a *prioritized block scheduling* scheme, which selects a block of messages to update via a priority. The priority allows messages that are more useful towards achieving convergence to be selected, and the computation cost of the priority is low. To this end, we introduce an efficient incremental update mechanism, which propagates only the changes of original messages. The change of a message is efficiently computed using the changes of original incoming messages. Also, the change can be directly utilized to calculate the priority. We refer to this mechanism as an *incremental-update* approach.

As the probabilistic graphical models are applied to model large and complex applications, such as image restoration for high-resolution images, it is desirable to leverage the parallelism of a cluster of machines to reduce the inference time. Therefore, we design and implement a distributed framework, *Prom*, which facilitates the implementation of BP and other graph algorithms in a distributed environment. Prom uses the proposed scheduling scheme as its built-in scheduling and supports the incremental-update approach. We evaluate two BP algorithms, the sum-product algorithm and the max-product algorithm on Prom, on a local cluster of machines as well as the Amazon EC2 cloud [1].

More specifically, our main contributions are as follows:

- We propose a novel scheduling scheme for BP algorithms. It selects a set of vertices to update at a time (in turn, a set of messages are selected, since all its outgoing messages are selected when a vertex is selected). As a result, it performs the selection of vertices for many message updates simultaneously instead of for one message update, and thus reduces the overhead of scheduling (since the amortized cost of selecting one message is low).

- We present a novel priority, which is leveraged to determine which messages are selected. The priority is vertex-based and can well capture the gain of updating

a vertex (updating its outgoing messages). In other words, updating a vertex with large priority value will send out highly useful outgoing messages towards achieving convergence. To keep the computation of the priority inexpensive, an incremental-update approach is introduced. The message computed by the incremental-update approach can be directly used to derive priority. Furthermore, the message update in the incremental-update approach can be done by accumulating incoming changes rather than by computing from scratch.

- We develop an asynchronous distributed framework, Prom, to support the proposed scheduling scheme and the incremental-update approach. Prom eases the process of programming BP and other graph algorithms in a distributed environment and does not require users to have distributed programming experience. Prom is evaluated via extensive experiments with both synthetic and real-world data. The evaluation results show that the proposed scheduling scheme outperforms the state-of-the-art counterpart and the incremental-update approach can further boost it. Moreover, a scalability test on a 50-node cluster demonstrates nearly linear scaling performance for large graphical models.

The rest of this chapter is organized as follows. Section 4.2 briefly reviews the background of BP. Section 4.3 introduces an incremental update mechanism for BP algorithms. Our scheduling scheme is presented in Section 4.4. Section 4.5 provides the design and implementation of Prom. Section 4.6 presents the evaluation result, and Section 4.7 discusses related work. This chapter is concluded in Section 4.8.

4.2 Belief Propagation

Probabilistic graphical models, such as Bayesian networks, factor graphs, and pairwise Markov Random Fields (MRFs), are popular tools to capture uncertainty in real-world applications. Without loss of generality, we consider factor graphs, since

any other graphical models can be converted to factor graphs [43]. A factor graph is a bipartite graph with two types of vertices: variable vertices and factor vertices. Each variable vertex represents a single random variable (e.g., x_i). Each factor vertex (e.g., f_j) denotes a function that maps a subset of random variable values (e.g., X_j) to a non-negative real-valued number so as to capture the compatibility of an assignment to those variables. The arguments are graphically represented by edges, which connect a particular function vertex with its variable vertices. Therefore, a factor graph is a factored representation of a joint probability distribution: $P(x_1, x_2, ..., x_n) = \frac{1}{Z}\prod_{j\in J} f_j(X_j)$, where Z is the normalization constant.

We next briefly review two BP algorithms, the sum-product algorithm and the max-product algorithm, and then discuss asynchronous BP algorithms.

4.2.1 Sum-Product Algorithm

Marginal probabilities of the distribution represented by a factor graph are central to inference. The sum-product algorithm provides an efficient way to compute marginal probabilities on a factor graph. It propagates messages in both directions along edges. Each vertex sends and receives messages till reaching a stable situation, and then the incoming messages are used to estimate the marginal probabilities of the vertex. Let $\mathfrak{m}_{i\to a}(x_i)$ and $\mathfrak{m}_{a\to i}(x_i)$ denote the message sent from variable vertex x_i to factor vertex f_a and the message sent from f_a to x_i, respectively. They can be updated by the following equations:

$$\mathfrak{m}^t_{i\to a}(x_i) = \lambda \prod_{k\in N(i)\backslash a} \mathfrak{m}^{t-1}_{k\to i}(x_i), \tag{4.1}$$

$$\mathfrak{m}^t_{a\to i}(x_i) = \lambda \sum_{X_j\backslash x_i} f(X_j) \prod_{j\in N(a)\backslash i} \mathfrak{m}^{t-1}_{j\to a}(x_j), \tag{4.2}$$

where $N(i)\backslash a$ denotes the set of neighbors of a given vertex i (x_i) excluding vertex a (f_a), and λ is a normalization factor to ensure all elements of the messages sum to 1.

The belief at a variable vertex (e.g., i) is proportional to the product of all the messages coming to the vertex: $b_i(x_i) \propto \prod_{k \in N(i)} \mathfrak{m}_{k \to i}(x_i)$. Then, the estimate of the marginal probability is $P(x_i) \approx b_i(x_i)$. While the sum-product algorithm converges to the exact marginal probabilities in acyclic graphs, there are no guarantees of convergence or correctness for graphs with loops. Nonetheless, the sum-product algorithm is widely applied on cyclic graphs for approximate inference with great success [20, 65, 92].

4.2.2 Max-Product Algorithm

In some cases, we are interested in determining which valid configuration has the largest probability rather than determining the marginal probabilities. The max-product algorithm addresses this problem efficiently. Message updates in the max-product algorithm are similar with those in the sum-product algorithm. In fact, we only need to replace $\sum$ with max in computing factor-to-variable messages. The message updates in the max-product algorithm are as follows:

$$\mathfrak{m}_{i \to a}^{t}(x_i) = \lambda \prod_{k \in N(i) \setminus a} \mathfrak{m}_{k \to i}^{t-1}(x_i), \tag{4.3}$$

$$\mathfrak{m}_{a \to i}^{t}(x_i) = \lambda \max_{X_j \setminus x_i} f(X_j) \prod_{j \in N(a) \setminus i} \mathfrak{m}_{j \to a}^{t-1}(x_j). \tag{4.4}$$

4.2.3 Asynchronous BP

We can represent each message as a vector in the vector space $\mathfrak{S} \subset R^d$, and represent an entire set $\mathfrak{M}$ of messages as a vector in $\mathfrak{S}^{|\mathfrak{M}|}$. The BP algorithm can be considered as the iterative algorithm with an update function $F : \mathfrak{S}^{|\mathfrak{M}|} \to \mathfrak{S}^{|\mathfrak{M}|}$, i.e. $\mathfrak{m}^t = F(\mathfrak{m}^{t-1})$.

BP aims to find a fixed point $\mathfrak{m}^*$ where $\mathfrak{m}^* = F(\mathfrak{m}^*)$. BP is guaranteed to converge to a unique $\mathfrak{m}^*$, if the update function F is a contraction under a message norm,

$$\|F(\mathfrak{m}) - \mathfrak{m}^*\| \leq \alpha \|\mathfrak{m} - \mathfrak{m}^*\|, 0 \leq \alpha < 1,$$

where the message norm $\|\cdot\|$ measures the distance between messages. If F is a max-norm contraction, then we have $\|F(\mathfrak{m}) - \mathfrak{m}^*\|_\infty \leq \alpha\|\mathfrak{m} - \mathfrak{m}^*\|_\infty$, where the max-norm $\|\cdot\|_\infty$ is defined as the maximum of the individual message norms, $\|\mathfrak{m}^t - \mathfrak{m}^{t-1}\|_\infty = \max_{i,j} \|\mathfrak{m}_{i\to j}^t - \mathfrak{m}_{i\to j}^{t-1}\|$. In this research, we use the max-norm to measure the convergence of BP. Mooij and Kappen [68] present sufficient conditions for F to be a contraction under the max-norm.

Function F can also be viewed as a set of individual functions, and each individual function F_i applies to one message. These individual update functions can be used to define synchronous BP and asynchronous BP. In *synchronous BP*, the functions compute the new values of all messages simultaneously at every iteration using their values from last iteration. In *asynchronous BP*, the functions update messages using the most recent values. The convergence rate of asynchronous BP (with a pre-defined update order) is proven to be at least as good as that of synchronous BP [26].

For asynchronous BP, it has been shown that the dynamic scheduling, which uses a priority to determine the order of updating messages, converges much faster than the static scheduling [26, 29, 30, 83]. The intuition behind the dynamic scheduling is that sending a message whose current value is very different from its previous value is perhaps more useful, and thus leads to more rapid transfer of information across the graph, while sending a message whose value does not change is useless.

4.3 Incremental Updates

The general techniques of incremental updates have shown efficiency in many algorithms, such as Nonnegative Matrix Factorization [95] and Expectation-Maximization [96]. In this section, we present an incremental update mechanism for BP algorithms, referred to as an *incremental-update* approach. In contrast, the traditional way of updating messages (described in the previous section) is referred to as a *basic-update* approach. The incremental-update approach propagates only the incremental part

(change) of the original message. The message update in the incremental-update approach can be performed by accumulating incoming changes instead of computing from scratch, and thus is much more efficient than that in the basic-update approach. Furthermore, since it usually calculates the priority value using the changes of messages, the dynamic scheduling can benefit from the incremental-update approach.

The basic idea of the incremental update is inspired by the Hugin architecture [21], an approach proposed for the exact inference. It uses an efficient way to update messages, which computes the marginal of a vertex as the product of messages once and then divides a message out from the marginal when one needs to update a message. However, the incremental update we proposed aims to support asynchronous computation. The order of asynchronous computations is based on a priority-based scheduling. The message computed by our incremental update can be directly used to derive priority, while there is no concept of priority in the Hugin architecture. Furthermore, our incremental update performs log-space calculations, so it can use addition/subtraction to update messages, while the Hugin architecture uses more expensive multiplication/division.

To derive an incremental update mechanism for a BP algorithm, we treat messages in log-space. A message in log-space is the logarithmic equivalent of the original message, i.e., $m(x_i) = \ln \mathfrak{m}(x_i)$.

4.3.1 Incremental Updates for Sum-Product

When the messages are in log-space, the message computation for the sum-product algorithm is as follows:

$$m_{i\to a}^{t}(x_i) = \sum_{k\in N(i)\backslash a} m_{k\to i}^{t-1}(x_i) + \beta, \tag{4.5}$$

$$m_{a\to i}^{t}(x_i) = \ln(\lambda \sum_{X_j\backslash x_i} f(X_j) e^{g_{a\to i}^{t-1}(x_j)}), \tag{4.6}$$

where $m(x_i) = \ln \mathfrak{m}(x_i)$, $\beta = \ln(\lambda)$, and $g^{t-1}_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} m^{t-1}_{j\to a}(x_j)$. Then, the belief at a variable vertex (e.g., i) can be computed as: $b_i(x_i) \propto e^{\sum_{k\in N(i)} m_{k\to i}(x_i)}$.

We can make a slight modification to Eq. (4.5) in which we omit normalization factor β. As Pearl [72] pointed out, normalizing the messages is only to avoid numerical underflow and makes no differences to the final beliefs. Since we still keep the normalization factor in Eq. (4.6) and messages are in log-space, there is no numerical underflow problem. Then, the message computation can be performed incrementally. The message $m^t_{i\to a}(x_i)$ can be incrementally computed as follows:

$$\Delta m^t_{i\to a}(x_i) = \sum_{k\in N(i)\setminus a} \Delta m^{t-1}_{k\to i}(x_i), \tag{4.7}$$

$$m^t_{i\to a}(x_i) = m^{t-1}_{i\to a}(x_i) + \Delta m^t_{i\to a}(x_i), \tag{4.8}$$

where $m^0_{i\to a}(x_i) = 0$, and $\Delta m^0_{k\to i}(x_i) = m^0_{k\to i}(x_i)$ is the initial message.

In our incremental-update approach, a vertex sends the incremental part of the original message instead of the message itself. For example, vertex x_i sends message $\Delta m^t_{i\to a}(x_i)$ to factor vertex f_a. In order to compute the belief, variable vertex x_i also accumulates the messages received from its neighbors, e.g., $m^t_{k\to i}(x_i) = m^{t-1}_{k\to i}(x_i) + \Delta m^t_{k\to i}(x_i)$.

The function $g_{a\to i}(x_j)$ in Eq. (4.6) can be also incrementally computed. We have

$$\Delta g^t_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} \Delta m^t_{j\to a}(x_j), \tag{4.9}$$

$$g^t_{a\to i}(x_j) = g^{t-1}_{a\to i}(x_j) + \Delta g^t_{a\to i}(x_j), \tag{4.10}$$

where $g^0_{a\to i}(x_j) = 0$.

Then, the incremental message sent from factor vertex f_a to variable vertex x_i can be computed as follows:

$$\Delta m^t_{a\to i}(x_i) = m^t_{a\to i}(x_i) - m^{t-1}_{a\to i}(x_i) \tag{4.11}$$
$$= \ln(\lambda \sum_{X_j \setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}) - m^{t-1}_{a\to i}(x_i),$$

where $m^0_{a\to i}(x_i)$ is the initial message. Factor vertex f_a also keeps records of $g^{t-1}_{a\to i}(x_j)$ and $m^{t-1}_{a\to i}(x_i)$.

Since the incremental-update approach uses only new incoming incremental messages to compute outgoing incremental messages, the complexity of computing an outgoing message for a vertex depends on the number of new incoming messages the vertex has received (since last update) rather than the vertex's degree. This is highly useful especially in the asynchronous communication model (e.g., under the dynamic scheduling), in which only part of a vertex's incoming messages may be updated when the algorithm computes its outgoing messages. In contrast, the basic-update approach always computes messages from scratch no matter how many incoming messages are updated. Its computation complexity is determined by the vertex's degree.

4.3.2 Incremental Updates for Max-Product

When the messages are in log-space, the message computation for the max-product algorithm is as follows:

$$m^t_{i\to a}(x_i) = \sum_{k\in N(i)\setminus a} m^{t-1}_{k\to i}(x_i) + \beta, \tag{4.12}$$

$$m^t_{a\to i}(x_i) = \ln(\lambda \max_{X_j \setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}), \tag{4.13}$$

where $m^t_{i\to a}(x_i) = \ln \mathfrak{m}^t_{i\to a}(x_i)$, $m^{t-1}_{k\to i}(x_i) = \ln \mathfrak{m}^{t-1}_{k\to i}(x_i)$, $m^t_{a\to i}(x_i) = \ln \mathfrak{m}^t_{a\to i}(x_i)$, $\beta = \ln(\lambda)$, and $g^{t-1}_{a\to i}(x_j) = \sum_{j\in N(a)\setminus i} m^{t-1}_{j\to a}(x_j)$.

The only difference in computing messages between the max-product algorithm and the sum-product algorithm is that the former one replaces $\sum$ with max in computing factor-to-variable messages. As a result, the message update for the max-

product algorithm can be performed incrementally as well. Computing the incremental variable-to-factor message is the same with that in the sum-product algorithm (so is $g_{a\to i}(x_j)$). Here, we only show how to incrementally compute the factor-to-variable message. The incremental message sent from factor vertex f_a to variable vertex x_i can be computed as follows:

$$\begin{aligned}\Delta m^t_{a\to i}(x_i) &= m^t_{a\to i}(x_i) - m^{t-1}_{a\to i}(x_i) \\ &= \ln(\lambda \max_{X_j \setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}) - m^{t-1}_{a\to i}(x_i),\end{aligned} \tag{4.14}$$

where $m^0_{a\to i}(x_i)$ is the initial message. Factor vertex f_a also keeps records of $g^{t-1}_{a\to i}(x_j)$ and $m^{t-1}_{a\to i}(x_i)$.

Using mathematical induction, it is straightforward to verify that performing message updates traditionally (i.e., the basic-update approach) and performing message updates incrementally (i.e., the incremental-update approach) are equivalent.

4.4 Our Scheduling Scheme

In this section, we present our scheduling scheme, which is inspired by the *residual scheduling* [26]. The residual scheduling leverages the difference in values of the message before and after the update as the residual of the message. By giving the message with high residual a high execution priority, the BP algorithm can potentially converge fast. The residual scheduling uses a priority queue to order all outgoing messages' residuals. Every time it sends out the outgoing message with the largest residual in the priority queue and then updates the queue.

The issue of the residual scheduling is that it has high overhead. It always selects one message to update at a time. Once the message is updated, it needs to recompute the priorities of the messages that have been affected and maintain the priority queue so as to select next message. Moreover, the residual scheduling determines a message's

priority by actually computing the message. Many messages are computed only for the purpose of obtaining their priority values, and are never sent out. As a result, in order to select one message, many operations have to be performed.

Our scheduling scheme selects a set of messages instead of a single one to update each time so as to reduce the cost. It utilizes a priority to determine which messages are selected. In addition, we also present a novel priority, which allows messages that are more useful towards achieving convergence to be selected (without actually computing the messages in advance).

4.4.1 Prioritized Block Scheduling

Our scheduling scheme is over vertices. That is, when a selected vertex is updated, all its outgoing messages will be computed and sent out. Scheduling over vertices rather than messages can reduce the cost of selecting messages, since a vertex usually has at least several messages. Updating a vertex always uses the most recently available data (i.e., incoming messages). Our scheduling scheme selects a block of k vertices to update each time. Once the block of selected vertices are updated, it selects another block of vertices to update. A priority is used to determine which vertices are selected. Every time our scheduling scheme selects the top-k vertices in terms of the priority value. Since our scheduling scheme selects a block of vertices to update via a priority, we refer to it as the *prioritized block scheduling*.

The size of the block (i.e., k) balances the tradeoff between the gain from the prioritized block scheduling and the cost of selecting the k vertices. Setting k too small may incur considerable cost, e.g., when $k = 1$, the prioritized block scheduling can be in principle seen as a vertex-based version of the residual scheduling (since it selects one vertex to update at a time). Setting k too large may degrade the effect of the prioritized block scheduling, e.g., if setting k as the number of vertices, it degrades to the round-robin scheduling. We will show in experiments (Section 4.6.3)

that a quite large range of k can allow the prioritized block scheduling to have better performance than the round-robin scheduling.

The prioritized block scheduling uses an efficient way to select the top-k vertices. The naive way is to first sort all the vertices by their priority values and then pick the top ones. However, sorting all the vertices can be expensive and time consuming (at least $O(n \log n)$ time). Instead, the prioritized block scheduling first finds the vertex with the k-th largest priority value. Then, it utilizes the k-th largest priority value as a threshold to filter the vertices. That is, it scans all the vertices once and picks only the vertices with larger or equivalent priority values. Randomized-Select [19] is utilized to find the k-th largest value. It has an expected running time of $O(n)$. In this way, the prioritized block scheduling takes $O(n)$ time (including the time in scanning all the vertices) in extracting the top-k vertices.

Our prioritized block scheduling has much lower cost of selecting one message than the residual scheduling. Updating one message in the residual scheduling needs to reset its residual and adjust the dependent messages' residuals (the messages that will be sent from the updated message's destination vertex). Assuming the degree of the message's destination vertex is d, there are $(d-1)$ dependent messages. We know that adjusting an element's priority value in a priority queue with n elements typically needs $O(\log n)$ time. Given a factor graph with $|V|$ vertices and $|E|$ edges, there are $O(|E|)$ messages in the priority queue. Hence, selecting a message to update in the residual scheduling needs $O(d{*}\log|E|)$ time, $O(\log|E|)$ for the selected message itself and $(O(d-1)*\log|E|)$ for the $(d-1)$ dependent messages. In our prioritized block scheduling, selecting k vertices to update only needs $O(|V|)$ time. Suppose the averaged degree of these k vertices is d'. Then, $(k*d')$ messages will be updated once the k vertices are selected. As a result, the amortized cost of selecting one message to update is $O(\frac{|V|}{k*d'})$. For a reasonably large k (e.g., k is one tenth of $|V|$), the cost is low and much lower than that in the residual scheduling.

4.4.2 Priority

We define the residual of an incremental message $\Delta m(x_i)$ as its $L1$-norm (in log-space),

$$r(\Delta m) = \sum_{x_i} |\Delta m(x_i)|.$$

Next, we derive the priority utilized in our prioritized block scheduling for the sum-product algorithm and for the max-product algorithm, respectively. The priority is vertex-based, and the priority of a vertex is directly computed from the residuals of its incoming messages.

4.4.2.1 Priority in Sum-Product

For any outgoing message sending from a variable vertex (e.g., i), its residual can be computed as follows:

$$r(\Delta m_{i\to a}) = \sum_{x_i} |\Delta m^t_{i\to a}(x_i)| = \sum_{x_i} |\sum_{k\in N(i)\backslash a} \Delta m^{t-1}_{k\to i}(x_i)|.$$

Therefore, we use the summation over all assignments of incoming messages in log-space,

$$pr_i = \sum_{x_i} |\sum_{k\in N(i)} \Delta m^{t-1}_{k\to i}(x_i)|,$$

as the priority of a variable vertex (i), which well approximates the residual of each individual outgoing message of the variable vertex.

For any outgoing message sending from a factor vertex (e.g., a), its residual can be computed as follows:

$$\begin{aligned} r(\Delta m_{a\to i}) &= \sum_{x_i} |\Delta m^t_{a\to i}(x_i)| \\ &= \sum_{x_i} |\ln \frac{\sum_{X_j\backslash x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\sum_{X_j\backslash x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{y_1+y_2}{z_1+z_2} \leq \max\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we have

$$\begin{aligned}
r(\Delta m_{a\to i}) &\leq \sum_{x_i} |\ln \max_{X_j \backslash x_i} \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&\leq \sum_{x_i} |\max_{X_j \backslash x_i} \ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&\leq \sum_{x_i} \max_{X_j \backslash x_i} |\ln \frac{e^{g^{t-1}_{a\to i}(x_j)}}{e^{g^{t-2}_{a\to i}(x_j)}}| \\
&= \sum_{x_i} \max_{X_j \backslash x_i} |\Delta g^{t-1}_{a\to i}(x_j)| \\
&= \sum_{x_i} \max_{X_j \backslash x_i} |\sum_{j \in N(a) \backslash i} \Delta m^{t-1}_{j\to a}(x_j)|.
\end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{y_1+y_2}{z_1+z_2} \geq \min\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we have

$$\begin{aligned}
r(\Delta m_{a\to i}) &\geq \sum_{x_i} |\ln \min_{X_j \backslash x_i} \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&\geq \sum_{x_i} |\min_{X_j \backslash x_i} \ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&\geq \sum_{x_i} \min_{X_j \backslash x_i} |\ln \frac{e^{g^{t-1}_{a\to i}(x_j)}}{e^{g^{t-2}_{a\to i}(x_j)}}| \\
&= \sum_{x_i} \min_{X_j \backslash x_i} |\Delta g^{t-1}_{a\to i}(x_j)| \\
&= \sum_{x_i} \min_{X_j \backslash x_i} |\sum_{j \in N(a) \backslash i} \Delta m^{t-1}_{j\to a}(x_j)|.
\end{aligned}$$

We have derived the lower bound and the upper bound for $r(\Delta m_{a\to i})$. Then, we use a value between these two bounds to approximate $r(\Delta m_{a\to i})$. Let $v_{a\to i} = \sum_{x_i} \frac{1}{s} \sum_{X_j \backslash x_i} |\sum_{j \in N(a) \backslash i} \Delta m^{t-1}_{j\to a}(x_j)|$, where s is the number of possible states of $X_j \backslash x_i$. We can see that (since $v_{a\to i}$ is the average) $v_{a\to i}$ is between those bounds. Therefore, we use $v_{a\to i}$ to approximate $r(\Delta m_{a\to i})$, and use the summation of averaged values over all assignments of incoming messages in log-space,

$$pr_a = \sum_{x_i} \frac{1}{s} \sum_{X_j \backslash x_i} | \sum_{j \in N(a)} \Delta m_{j \to a}^{t-1}(x_j)|,$$

as the priority of a factor vertex (a). Intuitively, this priority well captures the importance of new incoming messages available to the factor vertex.

4.4.2.2 Priority in Max-Product

The message update for a variable vertex in the max-product algorithm is the same with that in the sum-product algorithm. Accordingly, the priority for a variable vertex defined in the sum-product algorithm also applies to the max-product algorithm. Next, we derive the priority for a factor vertex in the max-product algorithm.

For any outgoing message sending from a factor vertex (e.g., a), its residual can be computed as follows:

$$\begin{aligned} r(\Delta m_{a \to i}) &= \sum_{x_i} |\Delta m_{a \to i}^{t}(x_i)| \\ &= \sum_{x_i} |\ln \frac{\max_{X_j \backslash x_i} f(X_j) e^{g_{a \to i}^{t-1}(x_j)}}{\max_{X_j \backslash x_i} f(X_j) e^{g_{a \to i}^{t-2}(x_j)}}|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{\max\{y_1, y_2\}}{\max\{z_1, z_2\}} \leq \max\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we can derive the following inequations:

$$\begin{aligned} r(\Delta m_{a \to i}) &= \sum_{x_i} |\ln \frac{\max_{X_j \backslash x_i} f(X_j) e^{g_{a \to i}^{t-1}(x_j)}}{\max_{X_j \backslash x_i} f(X_j) e^{g_{a \to i}^{t-2}(x_j)}}| \\ &\leq \sum_{x_i} \max_{X_j \backslash x_i} |\ln \frac{f(X_j) e^{g_{a \to i}^{t-1}(x_j)}}{f(X_j) e^{g_{a \to i}^{t-2}(x_j)}}| \\ &= \sum_{x_i} \max_{X_j \backslash x_i} |\sum_{j \in N(a) \backslash i} \Delta m_{j \to a}^{t-1}(x_j)|. \end{aligned}$$

Applying the fact for any $y_1 > 0, y_2 > 0, z_1 > 0, z_2 > 0$, $\frac{\max\{y_1, y_2\}}{\max\{z_1, z_2\}} \geq \min\{\frac{y_1}{z_1}, \frac{y_2}{z_2}\}$, we can derive the following inequations:

$$
\begin{aligned}
r(\Delta m_{a\to i}) &= \sum_{x_i} |\ln \frac{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{\max_{X_j\setminus x_i} f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&\geq \sum_{x_i} \min_{X_j\setminus x_i} |\ln \frac{f(X_j) e^{g^{t-1}_{a\to i}(x_j)}}{f(X_j) e^{g^{t-2}_{a\to i}(x_j)}}| \\
&= \sum_{x_i} \min_{X_j\setminus x_i} |\sum_{j\in N(a)\setminus i} \Delta m^{t-1}_{j\to a}(x_j)|.
\end{aligned}
$$

From the above inequations, we can see that the max-product algorithm has the same bounds for the residual of an outgoing message sending from a factor vertex as the sum-product algorithm. Accordingly, the priority for a factor vertex defined in the sum-product algorithm applies to the max-product algorithm as well.

The defined priority uses summation to aggregate incoming messages, and thereby we call it the *sum priority*. From the above derivation, we can see that the sum priority has strong connections with the residuals of its outgoing messages and thus well captures the gain of updating the vertex. That is, updating a vertex with large sum priority will send out highly useful outgoing messages. In contrast, updating a vertex with zero sum priority will waste a update, since the outgoing messages will not change.

4.4.3 Convergence

The prioritized block scheduling guarantees that BP algorithms converge if update function F is a max-norm contraction. It has been shown that when F is a max-norm contraction, if a scheduling scheme can guarantee that every message is updated infinitely often (until convergence), the BP algorithm will converge [26]. We first show that our prioritized block scheduling can fulfill this requirement.

Lemma 4.4.1. *If update function F is a max-norm contraction, the prioritized block scheduling guarantees that every message is updated infinitely often.*

Proof. We prove this lemma by contradiction. Assume there are a set of messages that belong to (sent from) a set of vertices, C, which are updated only before a time

point t. We use pr_i to denote the priority value of vertex i. Since update function F is a contraction, the messages that are updated will move towards their fixed points. Consequently, at some time point after t, for any vertex that does not belong to C (i.e., $i \in (V - C)$, where V is the whole set of vertices), its outgoing messages can reach the fixed points (since they are always being updated). At that time, for any $i \in (V - C)$, we have $pr_i = 0$; if we also have $pr_i = 0$ for any $i \in C$, the BP algorithm has converged; otherwise, a vertex in C (e.g., j, $pr_j > 0$) must be selected to update, which contradicts with the assumption that any vertex in C is updated only before time point t. □

Therefore, we have the following theorem.

Theorem 4.4.2. *If update function F is a max-norm contraction, BP algorithms with the prioritized block scheduling converge.*

4.5 Distributed Framework

BP algorithms and its variants are commonly used to perform inference on large real-world probabilistic graphical models. It is desirable to leverage the parallelism of a cluster of machines to reduce the completion time, and to have a general framework to facilitate the implementation in a distributed environment. BP algorithms (and its many extensions) are graph algorithms. Actually, graph algorithms have become an essential component in knowledge discovery, since graphs can capture complex dependencies and interactions. Therefore, we propose *Prom*, an asynchronous distributed framework for graph algorithms.

Prom provides several high-level APIs to users for implementing BP or other graph algorithms without worrying about the complexity of parallel computation. Prom supports asynchronous executions on graphs, in which vertices are updated using the latest available values, and leverages the proposed prioritized block scheduling as its default scheduling in order to efficiently order vertex updates.

Prom is built upon *Maiter* [107], an open-source graph processing framework. Maiter has shown good performance for several graph algorithms. In Maiter, users specify the application logic simply through a vertex update function. However, Maiter assumes that each vertex (or message) has only one scalar value (e.g. a floating-point number), and thus cannot support algorithms with vector values, such as BP and Personalized PageRank [37]. Additionally, Maiter assumes that the update function has only one operation (e.g., addition) with commutative and associative properties, but there are many graph algorithms with more than one operations in the update function (e.g., sum-product has addition and multiplication). These limitations need to be removed so as to accommodate more graph algorithms. To this end, Prom extends Maiter to support a broader class of graph algorithms efficiently. Prom makes two basic assumptions: (1) the graph structure is static and will not change during execution; (2) asynchronous execution with dynamically ordering vertex updates does not affect the correctness of the algorithm. Graph algorithms satisfying these two assumptions can be implemented on Prom and can benefit from the efficient prioritized block scheduling.

A vertex-centric programming model (which has been shown to be efficient for many graph algorithms) is adopted by Prom. That is, each vertex is considered as an independent computing unit, and the operations are performed over vertices until termination. Vertex updates are performed on workers, and there is a master controlling the flow of computation. All workers (and the master) run in parallel and communicate through MPI.

4.5.1 Data Partition and Storage

The input graph is split into partitions and each worker is responsible for one partition. Each partition consists of a set of vertices and all their (outgoing) edges. Each worker leverages an in-memory table, *info table*, to store the vertices in its

partition. For graph algorithms under the vertex-centric programming model, storing the following information is typically sufficient for a vertex: ID, incoming messages, outgoing messages, priority, state, and edges (with edge data associated with each edge). Hence, as shown in Figure 4.1, Prom represents a vertex by a tuple with six fields, $\{v, im, om, pr, st, sd\}$, where field v for the vertex ID, im for the incoming messages, om for the outgoing messages, pr for the priority value, st for the state, and sd for the static data (e.g., edges and their associated data).

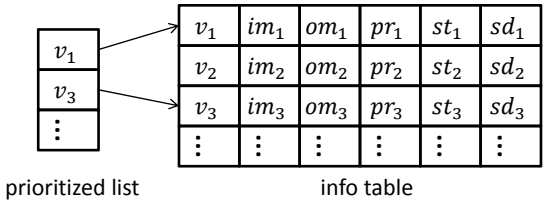


Figure 4.1. Data storage in a worker.

Prom allows users to define each field of the info table. For example, to implement the incremental-update approach for BP, we can define the incoming message field (im) of a vertex with $[\Delta m_{v_a}, \Delta m_{v_b}, \ldots, \Delta m_{v_l}, m_{v_a}, m_{v_b}, \ldots, m_{v_l}]$ (each item can be a vector), where Δm_{v_a} stores the new incoming incremental message from neighbor v_a, and m_{v_a} accumulates the incoming messages already received from v_a. The static data (sd) is usually defined to contain edges and the data associated with edges (e.g., factor functions of the factor graph). Each tuple is stored in one entry of the info table, which is indexed by the vertex ID (v).

4.5.2 Vertex Operation

Each worker has two main operations for its stored vertices: the catch operation and the update operation. The catch operation uses a user-defined function ($c\_fun()$) to aggregate a new incoming message for a vertex (say v_j) to its stored incoming messages. That is, function $c\_fun()$ needs to update the incoming message field (im_j) of vertex v_j, upon receiving a new incoming message. Also, it needs to update

the priority field (pr_j) to aggregate the importance of the new incoming message. By defining function $c\_fun()$ in different ways, users can realize different update approaches (e.g., incremental-update or basic-update) and priorities.

The update operation uses another user-defined function ($u\_fun()$) to compute outgoing messages (and the state) for scheduled vertices. When it is performed on a vertex, function $u\_fun()$ computes outgoing messages and updates the state (e.g., the belief distribution of the vertex) by incorporating the latest incoming messages, and modifies the incoming message field if necessary as well as resets the priority value to zero.

Prom uses MPI to transmit messages between workers. All messages during transmission are in the format (*dst*, *src*, *cnt*), where *dst* denotes the message's destination vertex, *src* indicates the source vertex, and *cnt* denotes the message's content. The catch operation and the update operation are realized in two threads for asynchronous execution.

4.5.3 Distributed Prioritized Execution

Prom leverages the prioritized block scheduling (described in Section 4.4.1) as its default scheduling scheme. Since a centralized ordering is inefficient in a distributed environment, Prom allows each worker to build its own prioritized block scheduling. Round by round, each worker selects its local top-k vertices in terms of the priority value as a block to update. All workers selects vertices independently.

A worker puts the block of selected vertices into a list, *prioritized list*. To minimize the copy cost, only vertex IDs are put in the prioritized list, as shown in Figure 4.1. Vertex IDs are used to locate corresponding vertices in the info table. All the vertices in the prioritized list will be updated by the update operation during the round. In the first round, all vertices are put into the prioritized list to guarantee that each vertex is updated at least once before convergence.

4.5.4 Distributed Termination Check

Prom adopts a passively monitoring model to perform termination check. Each worker utilizes a user-defined function ($m\_fun()$) to periodically measure its local progress by scanning the info table (typically looking at the incoming message field), and reports the progress to the master. The master aggregates the local progress reports from workers (in the way that a user specifies) so as to obtain the global progress, and in turn determines whether the termination condition is satisfied. If yes, the master sends termination signals to all workers. Upon receiving the terminate signal, a worker stops updating its info table and dumps the table to a distributed file system (i.e., HDFS) so as to reliably store the converged results.

We use the following convergence criterion (max-norm) for BP algorithms (where $\varepsilon \geq 0$ is a small constant):

$$\max_{i,j} \|\Delta m_{i \to j}\|_1 \leq \varepsilon.$$

4.6 Evaluation

In this section, we evaluate the proposed prioritized block scheduling and the priority. Both the sum-product algorithm and the max-product algorithm are implemented on Prom. For the comparison purpose, both the incremental-update approach and the basic-update approach are used. To show the performance of the prioritized block scheduling, we compare it with the round-robin scheduling (static scheduling). We also compare the prioritized block scheduling with the state-of-the-art dynamic scheduling.

4.6.1 Experiment Setup

The experiments are performed on a local cluster and a large-scale cluster on Amazon EC2 [1]. The local cluster consists of 4 machines, and each of them has Intel E8200 dual-core 2.66GHz CPU, 4GB of RAM, and 1TB of hard disk. These 4

machines are connected through a Gbit switch. The large-scale cluster consists of 50 medium instances.

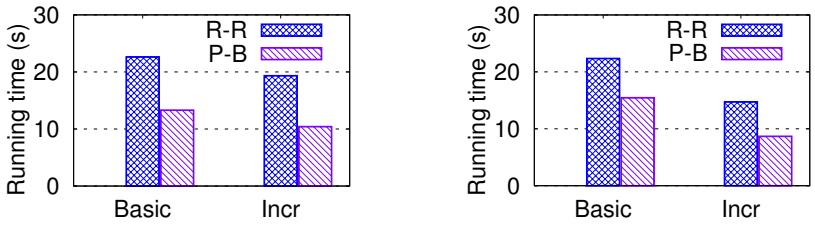
Table 4.1. Factor Graph Summary

Dataset	# of Vertices	Description
gird-n	$4 * n^2 - 2 * n$	$n \times n$ grid MRF
uw-theory	133,999	uw-theory MLN
uw-systems	414,340	uw-systems MLN

Both synthetic and real-world factor graphs are used. We generate one type of pairwise MRFs, random grids with binary variables (parameterized by the Ising model) [26], and convert them into factor graphs. Random grids are chosen because they are standard benchmarks for evaluating BP algorithms. For real-world graphs, we consider Markov Logic Networks (MLNs) [75]. *Alchemy* is leveraged to compile the MLNs from the UW-CSE data collection [4] into factor graphs. After compiling, the factor functions will be adjusted if BP algorithms on the compiled graphs do not converge. The factor graphs are summarized in Table 4.1. In order to load the strongly connected vertices to the same worker and thus reduce across-worker communication, we utilizes METIS [38] to split a graph into partitions.

Each worker by default sets k as 10% of the number of its local vertices. The convergence criterion is set to $\varepsilon = 10^{-4}$. Running times are averaged over 10 runs.

4.6.2 Efficiency of Prioritized Block Scheduling



(a) Sum-Product on grid-200 (b) Max-Product on grid-100

Figure 4.2. BP algorithms with different scheduling schemes and update approaches.

We first show the running time of BP algorithms with the prioritized block scheduling on the local cluster. The running time is measured as the wall-clock time that BP uses to reach the convergence criterion. The round-robin scheduling is also evaluated as a reference point. For the sum-product algorithm as well as the max-product algorithm, the prioritized block scheduling is faster than the round-robin scheduling with either the incremental-update approach or the basic-update approach, as presented in Figure 4.2. For example, the prioritized block scheduling is 1.9x faster for the sum-product algorithm on *grid-200* when the incremental-update approach is utilized. In addition, the incremental-update approach is always superior to the basic-update approach. Note that, in all figures, "P-B" indicates the prioritized block scheduling; "R-R" represents the round-robin scheduling; "Incr" and "Basic" denote the incremental-update approach and the basic-update approach, respectively.

Table 4.2. Vertex Degree Comparison

Graph	**overall avg. deg.**	**variable avg. deg.**
gird-200	2.5	5.0
uw-theory	3.8	55.7
uw-systems	3.8	78.8

To further show the advantage of the prioritized block scheduling, we evaluate both scheduling schemes for the sum-product algorithm on real-world factor graphs. The performance comparison for the max-product algorithm is similar and therefore omitted here. As plotted in Figure 4.3, the speedup of the prioritized block scheduling over the round-robin scheduling is up to 2.1x on real-world factor graphs (when the incremental-update approach is used). Moreover, compared with the basic-update approach, the incremental-update approach allows the prioritized block scheduling to achieve up to 4x speedup, much higher than that on the synthetic factor graphs (Figure 4.2(a)). The different speedups can be attributed to different structures of the factor graphs. For instance, the real-world factor graphs have much higher degrees for variable vertices, as shown in Table 4.2.

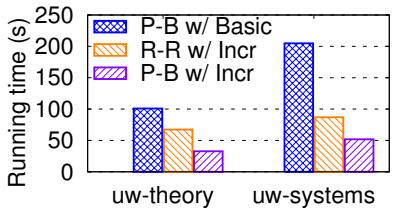


Figure 4.3. Prioritized block scheduling on real-world graphs.

We also measure the convergence speed of the different scheduling schemes (when the incremental-update approach is used). The test is performed on the real-world factor graph, *uw-theory*, and the max-norm ($\max_{i,j} |\Delta m_{i \to j}(x_i)|$) is used to measure the convergence progress. As shown in Figure 4.4, the prioritized block scheduling converges much more rapidly than the round-robin scheduling.

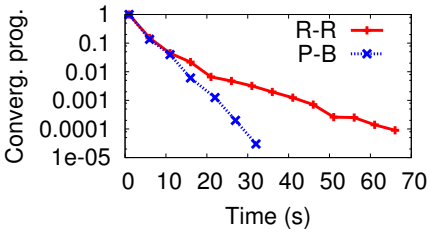
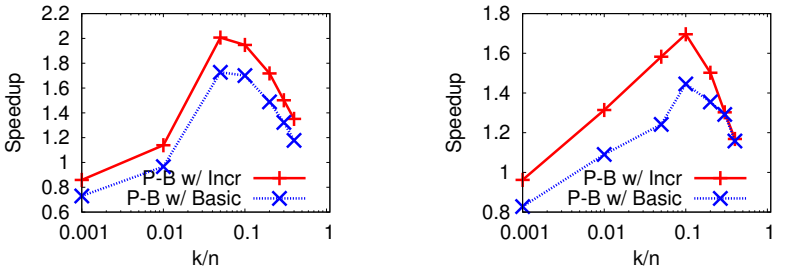


Figure 4.4. Convergence progress vs. time.

4.6.3 Impact of k

The block size (i.e., k) balances the tradeoff between the gain from the prioritized block scheduling and the cost of preparing the prioritized list. Figure 4.5 shows the convergence speedup results with different k. The speedup is measured over the running time when k is the number (n) of a worker's local vertices (i.e., the round-robin scheduling). From the figures, we can see that a quite large range of k can allow the prioritized block scheduling to have better performance than the round-robin scheduling (when either the incremental-update approach or the basic-update approach is used), and that the optimal speedup happens at around $k/n = 0.1$. This is also why we set $k/n = 0.1$ by default.



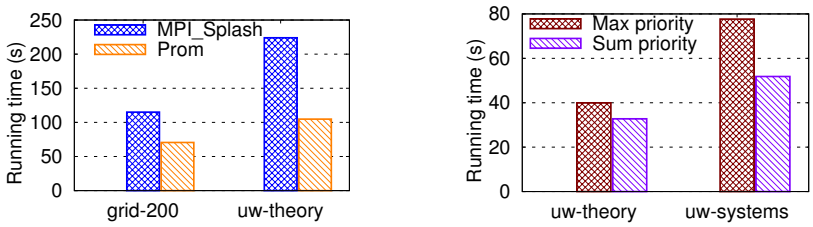
(a) Sum-Product on grid-200 (b) Max-Product on grid-100

Figure 4.5. The impact of k (varying k/n).

4.6.4 Comparison with Other Schedules

To further demonstrate the efficiency of its built-in prioritized block scheduling, Prom is also compared with another distributed implementation of the sum-product algorithm, MPI_Splash [30], on the local cluster. MPI_Splash utilizes the *DBRSplash scheduling*, a distributed version of the *ResidualSplash scheduling* [29]. The ResidualSplash scheduling applies a variation of the residual scheduling in a single machine (multiple-core) environment, and it has been shown that ResidualSplash is more efficiently than the original residual scheduling. By recognizing the high overhead of the residual scheduling, ResidualSplash also defines the residual over vertices instead of messages and selects a set of vertices to update at a time via a Splash operation. The Splash operation uses the vertex with the largest residual as a root and updates vertices around the root. However, not all vertices covered by the Splash operation have large residuals, and thus some updates might not be useful. ResidualSplash defines a vertex's priority as the maximum of the residuals of its incoming messages. To differentiate this priority with our sum priority, we refer to it as the *max priority*. The DBRSplash scheduling is the state-of-the-art dynamic scheduling for BP in a distributed environment.

To fairly compare Prom with MPI_Splash, we instruct Prom to use the same priority and termination condition as MPI_Splash. To compare scheduling schemes only, we leverage the basic-update approach to implement the sum-product algorithm



(a) Different schedules (b) Different priorities

Figure 4.6. Performance comparison with the state-of-the-art dynamic scheduling.

on Prom. As presented in Figure 4.6(a), Prom can be up to 2x faster than MPI_Splash, indicating that the prioritized block scheduling outperforms DBRSplash. In order to verify that the superiority of Prom over MPI_Splash stems from its scheduling scheme, we implement both the prioritized block scheduling and the ResidualSplash scheduling (single machine version of DBRSplash) in a single machine environment and evaluate them with the same settings. The prioritized block scheduling is 1.8x faster on *grid-200* and 2.3x faster on *uw-theory* than the ResidualSplash scheduling.

In order to show the performance of our sum priority, we compare it with the max priority. We evaluate these two priorities (when both are utilized by the prioritized block scheduling) for the sum-product algorithm on real-world graphs. As presented in Figure 4.6(b), the prioritized block scheduling with our sum priority is 1.2x faster on *uw-theory* and 1.5x faster on *uw-systems* than that with the max priority.

4.6.5 Accuracy

We also assess accuracy of the beliefs computed by Prom (using the prioritized block scheduling with the incremental-update approach) for the sum-product algorithm. We first compare with the exact result. Since exact inference is intractable on large graphical models, we here use a small factor graph, *grid-10*. The beliefs (of all variable vertices) computed by Prom are compared against the exact beliefs computed by the junction tree algorithm [48]. We use MPI_Splash as a reference point. Kullback-Leibler (KL) divergence is leveraged to measure the difference. From Figure

4.7, we can see that both Prom and MPI_Splash achieve high accuracy. For example, for more than 90% variable vertices, the KL divergence of the beliefs computed by Prom from the exact beliefs is less than 0.01.

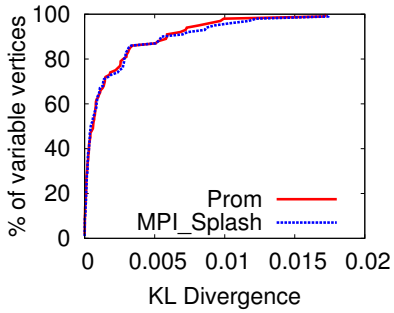
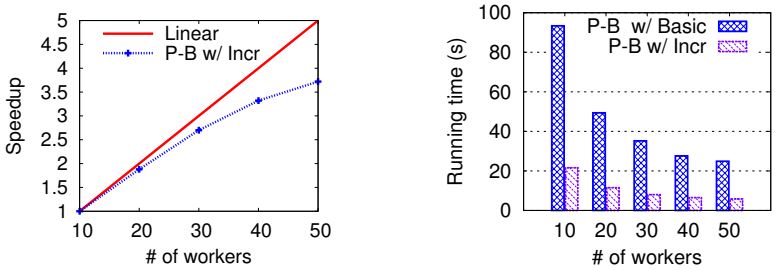


Figure 4.7. Cumulative percentage of variable vertices as a function of the KL divergence.

For large graphs, since exact inference is intractable, we only compare Prom with MPI_Splash. We evaluate both Prom and MPI_Splash on *grid-200*. Beliefs from both systems are compared by calculating the L^1-difference averaged over all variable vertices. The difference in beliefs computed by the two systems is less than 0.02 in terms of averaged L^1-difference per variable vertex.

4.6.6 Scaling Performance



(a) Speedup (b) Performance comparison

Figure 4.8. Scalability test on uw-systems.

Figure 4.8 presents the scaling performance of the prioritized block scheduling (for the sum-product algorithm) on Prom as the number of workers increases from 10 to 50 on the Amazon EC2 cloud. The real-world factor graph, *us-systems*, is used.

The speedup is calculated over the running time of 10 workers. We can see that the prioritized block scheduling exhibits nearly linear speedup, and that it always converges faster when the incremental-update approach is utilized than when the basic-update approach is utilized.

4.7 Related Work

Several works [26, 29, 30, 83] have shown that BP algorithms with the dynamic scheduling converge faster than those with the static scheduling. The earliest work [26] proposes the *residual scheduling*, which selects the outgoing message with largest residual to update each time. It uses a priority queue to order messages. Besides the large priority queue maintenance overhead, the problem of the residual scheduling is that it determines an outgoing message's residual by actually computing it. Later, Sutton and McCallum [83] propose to approximate the residual of an outgoing message rather than compute it in order to reduce the computation overhead. However, the cost of ordering messages so as to select the one with the largest residual is still high. Our prioritized block scheduling scheme selects a set of messages to update each time in order to reduce the cost.

The *ResidualSplash scheduling* [29] applies a variation of the residual scheduling in the multiple-core environment. It defines the residual over vertices instead of messages. The residual of a vertex is used to determine the Splash ordering, and a Splash operation uses the vertex with the largest residual as a root and propagates messages around the root (i.e., among the neighbors within fixed number hops). That is, it selects a set of messages to update at a time. The ResidualSplash scheduling outperforms the residual scheduling, since it reduces the cost of selecting one single message. However, not all vertices covered by the Splash operation have large residuals, and thus some updates might not be useful. The *DBRSplash scheduling* [30] extends the idea of the ResidualSplash scheduling to a distributed environment. In contrast,

our prioritized block scheduling selects vertices with high residuals uniformly, and therefore all scheduled updates are potentially useful.

Since massive graphs become increasingly popular, a series of parallel frameworks have emerged to scale graph processing. Among them, *Priter* [106], *Maiter* [107], *GRACE* [88], and *GraphLab* [61, 62] support prioritized execution. Priter is a MapReduce-based framework, which requires synchronous iterations. Maiter presents asynchronous execution but assumes that each vertex (or message) has only one scalar value. As a result, neither of them supports BP with dynamic scheduling. GRACE and GraphLab can support BP. GRACE relies on users to implement their own scheduling schemes and its prototype is built on a shared-memory architecture. GraphLab uses a general asynchronous model for graph algorithms and provides the Splash scheduling (based on ResidualSplash) for BP. In comparison, Prom provides a more efficient scheduling scheme, the prioritized block scheduling.

4.8 Conclusion

In this research, we propose an efficient dynamic scheduling scheme, the prioritized block scheduling, with a novel priority for BP algorithms. In order to efficiently compute the priority and update messages, we introduce an incremental-update approach, which is much more efficient than the traditional basic-update approach. In addition, to facilitate the implementation of BP algorithms and other graph algorithms in a distributed environment, we design and implement an asynchronous distributed framework, Prom. Prom uses the prioritized block scheduling as its default scheduling scheme. We implement two BP algorithms, the sum-product algorithm and the max-product algorithm, on Prom. With both synthetic and real-world datasets, the evaluation results show that the prioritized block scheduling outperforms the state-of-the-art dynamic scheduling scheme, and that the incremental-update approach can further accelerate the prioritized block scheduling.

CHAPTER 5

ASYNCHRONOUS DISTRIBUTED INCREMENTAL COMPUTATION ON EVOLVING GRAPHS

5.1 Introduction

A large class of data routinely produced and collected by large corporations can be modeled as graphs, such as web pages crawled by Google (e.g., the web graph) and tweets collected by Twitter (e.g., the mention graph for users). Since graphs can capture complex dependencies and interactions, graph algorithms have become an essential component in many real-world applications [5, 8, 34, 59, 74, 104], including business intelligence, social sciences, and data mining.

An essential property of graphs is that they are often dynamic. As new data and/or updates are being collected (or produced), the graph will evolve. For example, search engines periodically crawl the web, and the web graph is evolving as web pages and hyper-links are created and/or deleted. Many applications must utilize the up-to-date graph in order to produce results that can reflect the current state. However, rerunning the computation over the entire graph is not efficient (considering the huge size of the graph), since it discards the work done in earlier runs no matter how little changes have been made.

The dynamic nature of graphs implies that performing incremental computation can improve efficiency dramatically. Incremental computation exploits the fact that only a small portion of the graph has changed. It reuses the result of the prior computation and performs computation only on the part of the graph that is affected by the change. Although a number of distributed frameworks have been proposed

to support incremental computation on massive graphs [11, 17, 60, 69, 73, 102], most of them apply synchronous updates, which require expensive synchronization barriers. In order to avoid the high synchronization cost, asynchronous updates have been proposed [9]. In the asynchronous update model, a vertex performs the update using the most recent values instead of the values from the previous iteration (and there is no waiting time). Intuitively, we can expect asynchronous updates outperform synchronous updates since more up-to-date values are used and the synchronization barriers are bypassed. However, asynchronous updates might require more communications and perform useless computations (e.g., when no new value available to a vertex), and thus result in limited performance gain over synchronous updates.

In this chapter, we provide an approach to efficiently apply asynchronous updates to incremental computation. We first describe a broad class of graph algorithms targeted by this chapter. We then present our incremental computation approach through illustrating how to apply asynchronous updates to incremental computation. In order to address the challenge that asynchronous updates might require more communications and computations, we present a scheduling scheme to coordinate updates. Furthermore, we develop a distributed system to support our proposed asynchronous incremental computation approach. We evaluate our approach on a local cluster of machines as well as the Amazon EC2 cloud [1]. More specifically, our main contributions are as follows:

- We propose an approach to efficiently apply asynchronous updates to incremental computation on evolving graphs for a broad class of graph algorithms. In order to improve efficiency, a scheduling scheme is presented to coordinate asynchronous updates. The convergence of our proposed asynchronous incremental computation approach is proved.

- We develop an asynchronous distributed framework, GraphIn, to support incremental computation. GraphIn eases the process of implementing graph al-

gorithms with incremental computation in a distributed environment and does not require users to have the distributed programming experience.

- We extensively evaluate our asynchronous incremental computation approach with several real-world graphs. The evaluation results show that our approach can accelerate the convergence speed by as much as 14x when compared to recomputation from scratch. Moreover, a scalability test on a 50-machine cluster demonstrates our approach works with massive graphs having tens of millions of vertices and a billion of edges.

The rest of this chapter is organized as following. Section 5.2 formally defines the problem targeted by this chapter. Section 5.3 proposes our asynchronous incremental computation approach. The distributed framework for supporting the proposed asynchronous incremental computation approach is presented in Section 5.4. Section 5.5 presents the evaluation results. Section 5.6 surveys related work, and this chapter is concluded in Section 5.7.

5.2 Problem Setting

In this section, we first define the problem of performing algorithms on evolving graphs. We then describe a broad class of graph algorithms which we target.

5.2.1 Problem Formulation

Many graph algorithms leverage iterative updates to compute states (e.g., scores of importance, closenesses to a specified vertex) of the vertices until convergence points are reached. For example, PageRank iteratively refines the rank scores of the vertices (e.g., web pages) of a graph. Such a graph algorithm typically starts with some initial state and then iteratively refines it until convergence. We refer to this kind of graph algorithms as *iterative graph algorithms*.

We are interested in how to efficiently perform iterative graph algorithms on evolving graphs. More formally, if we use G to denote the original graph and G' to represent the new graph, the question we ask is: for an iterative graph algorithm, given G' and the convergence point on G, how to efficiently reach the convergence point on G'.

5.2.2 Iterative Graph Algorithms

We here describe the iterative graph algorithms targeted by this chapter. Typically, the update function of an iterative graph algorithm has the following form:

$$x^{(k)} = f(x^{(k-1)}), \tag{5.1}$$

where the n-dimensional vector $x^{(k)}$ presents the state of the graph at iteration k, each of its elements is the state for one vertex (e.g., $x^{(k)}[i]$ for vertex i), and $x^{(0)}$ is the initial state. A convergence point is a fixed point of the update function. That is, if $x^{(*)}$ is a convergence point, we have $x^{(*)} = f(x^{(*)})$.

The update function usually can be decomposed into a series of individual functions. In other words, we can update a vertex's state (e.g., x_j) as follows:

$$x_j^{(k)} = c_j \star \sum_{i=1}^{n} \star f_{\{i,j\}}(x_i^{(k-1)}), \tag{5.2}$$

where '$\star$' is an abstract operator ($\sum_{i=1}^{n} \star$ represents an operation sequence of length n by '$\star$'), c_j is a constant, and $f_{\{i,j\}}(x_i^{k-1})$ is an individual function denoting the impact from vertex i to vertex j in the k^{th} iteration. The operator '$\star$' typically has three candidates, '+', 'min', and 'max'. In this chapter, we target the iterative graph algorithm that can compute the state in the form of Eq. (5.2).

5.2.3 Example Graph Algorithms

We next illustrate a series of well-known iterative graph algorithms, the update functions of which can be converted into the form of Eq. (5.2).

PageRank and Variants: PageRank is a well-known algorithm, which ranks vertices in a graph based on the stationary distribution of a random walk on the graph. Each element (e.g., r_j) of the score vector r can be computed iteratively as follows: $r_j^{(k)} = \sum_{\{i|\{i\to j\}\in E\}} \frac{dr_i^{(k-1)}}{|N(i)|} + (1-d)e_j$, where d ($d < 1$) is the damping factor, $\{i \to j\}$ represents the edge from vertex i to vertex j, E is the set of edges, $|N(i)|$ is the number of outgoing edges of vertex i, and e is a size-n vector with each entry being $\frac{1}{n}$. We can convert the update function of PageRank into the form of Eq. (5.2). If there is an edge from vertex i to vertex j, $f_{\{i,j\}}(x_i^{(k-1)}) = dx_i^{(k-1)}/|N(i)|$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = 0$, $c_j = (1-d)e_j$, and '$\star$' is '+'.

The update function of Personalized PageRank [37] differs from that of PageRank only at vector e. Vector e of Personalized PageRank assigns non-zero values only to the entries indicating the personally preferred pages. Rooted PageRank [80] is a special case of Personalized PageRank. It captures the probability for two vertices to run into each other and uses this probability as the similarity score of those two vertices.

Shortest Paths: The shortest paths algorithm is a simple yet common graph algorithm which computes the shortest distances from a source vertex to all other vertices. Given a weighted graph, $G = (V, E, W)$, where V is the set of vertices, E is the set of edges, and W is the weight matrix of the graph (if there is no edge between i and j, $W[i,j] = \infty$). Then the shortest distance (i.e., d_j) from the source vertex s to a vertex j can be calculated by performing the iterative updates: $d_j^{(k)} = \min\{d_j^{(0)}, \min_i(d_i^{(k-1)} + W[i,j])\}$. For the initial state, we usually set $d_s^{(0)} = 0$ and $d_j^{(0)} = \infty$ for any vertex j other than s. We can map the update function of the shortest paths algorithm into the form of Eq. (5.2). If there is an edge from vertex i to vertex j, $f_{\{i,j\}}(x_i^{(k-1)}) = x_i^{(k-1)} + W[i,j]$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = \infty$, $c_j = d_j^{(0)}$, and '$\star$' is 'min'.

Connected Components: The connected components algorithm is an important algorithm for understanding graphs. It aims to find the connected components in a graph. The main idea of the algorithm is to label each vertex with the maximum vertex id across all vertices in the component which it belongs to. To this end, each vertex iteratively updates its component id as the maximum vertex id that it has seen. Initially, a vertex j sets its component id $p_j^{(0)}$ as its own vertex id, i.e., $p_j^{(0)} = j$. Then the component id of vertex j can be iteratively updated by $p_j^{(k)} = \max\{p_j^{(0)}, \max_i(p_i^{(k-1)})\}$. When no vertex in the graph changes its component id, the algorithm converges. As a result, the vertices having the same component id belong to the same component. We can map the update function of the connected components algorithm into the form of Eq. (5.2). If there is an edge from vertex i to vertex j, $f_{\{i,j\}}(x_i^{(k-1)}) = x_i^{(k-1)}$, otherwise $f_{\{i,j\}}(x_i^{(k-1)}) = -\infty$, $c_j = j$, and '$\star$' is 'max'.

Other Algorithms: There are many more iterative graph algorithms, update functions of which can be mapped into the form of Eq. (5.2). We name several ones here. Hitting time is a measure based on a random walk on the graph. Hitting time between vertices i and j is defined as the expected number of steps in a random walk starting from i to first time reach j. Penalized hitting probability [34] and discounted hitting time [77] are variants of hitting time. The former penalizes the random walk for each additional step with a damping factor, and the latter penalizes the transition probability. The Katz Measure is a similarity measure between two vertices. The measure is computed as the sum over the collection of paths between two vertices, exponentially damped by the path length to count short paths more heavily. The Adsorption algorithm [8] is a graph-based label propagation algorithm proposed for personalized recommendation. A vertex's label distribution is the convex combination of the labels of other vertices. Effective Importance [13] is a proximity measure on a graph, and can capture the local community structure of a vertex.

It can be considered as a degree normalized version of random walk with restart. HITS [42] utilizes a two-phase iterative update approach (the authority update and the hub update) to rank web pages of a web graph. SALSA [51] is another link-based ranking algorithm for web graphs. Like HITS, SALSA also iteratively updates two scores associated with each vertex, the hub score and the authority score.

5.3 Asynchronous Incremental Computation

As the underlying graph evolves, the states of the vertices also change. Obviously, rerunning the computation from scratch over the new graph is not efficient, since it discards the work done in earlier runs. Intuitively, performing computations incrementally can improve efficiency. In this section, we present our asynchronous incremental computation approach. The convergence of our approach is proved.

5.3.1 Asynchronous Updates

In order to describe our asynchronous incremental computation approach, we define a time sequence $\{t_0, t_1, \ldots, t_\infty\}$. Let $\hat{x}^{(k)}$ denote the state vector at time t_k. Also, we introduce the delta state vector $\Delta\hat{x}^{(k)}$ to represent the difference between $\hat{x}^{(k+1)}$ and $\hat{x}^{(k)}$ in the operator '$\star$' manner, i.e., $\hat{x}^{(k+1)} = \hat{x}^{(k)} \star \Delta\hat{x}^{(k)}$. The goal of introducing $\Delta\hat{x}^{(k)}$ is to perform accumulative computations. When the operator '$\star$' has the commutative property and the associative property and the function $f_{\{i,j\}}(x_i)$ has the distributive property over '$\star$', the computation can be performed accumulatively. All the graph algorithms discussed in Section 5.2.3 satisfy these properties. It is straightforward to verify that accumulative computations are equivalent to normal computations. The benefit of performing accumulative computations is that only changes of the states (i.e., delta states) are used to compute new changes. If there is no change for the state of a vertex, no communication or computation is necessary. The general idea of separating fixed parts from changes and leveraging changes to

compute new changes also shows efficiency in many other algorithms, such as Non-negative Matrix Factorization [95] and Expectation-Maximization [96].

In our asynchronous incremental computation approach, each vertex i updates its $\Delta\hat{x}_i^{(k)}$ and $\hat{x}_i^{(k)}$ independently and asynchronously, starting from $\Delta\hat{x}_i^{(0)}$ and $\hat{x}_i^{(0)}$ (we will illustrate how to construct them soon). In other words, there are two separate operations for vertex j:

- *Accumulate* operation: whenever receiving a value (e.g., $f_{\{i,j\}}(\Delta\hat{x}_i)$) from a neighbor (e.g., i), perform $\Delta\hat{x}_j = \Delta\hat{x}_j \star f_{\{i,j\}}(\Delta\hat{x}_i)$;
- *Update* operation: perform $\hat{x}_j = \hat{x}_j \star \Delta\hat{x}_j$; for any neighbor l, if $f_{\{j,l\}}(\Delta x_j^{(1)}) \neq o$, send $f_{\{j,l\}}(\Delta\hat{x}_j)$ to l; and then reset $\Delta\hat{x}_j$ to o;

where o is the identity value of the operator '$\star$'. That is, for $\forall z \in R$, $z = z \star o$ (if '$\star$' is '+', $o = 0$; if '$\star$' is 'min', $o = \infty$; if '$\star$' is 'max', $o = -\infty$). Basically, the *accumulate* operation accumulates received values between two consecutive updates on $\hat{x}_j$. The *update* operation adjusts $\hat{x}_j$ by absorbing $\Delta\hat{x}_j$, sends useful values to other vertices, and resets $\Delta\hat{x}_j$.

We now illustrate how to construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ by leveraging the computation result on the previous graph, G. We need to make sure that the constructed $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ can guarantee the correctness of the result on the new graph. Let $\bar{x}^{(*)}$ denote the convergence point on G. We next show how to construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ when the operator '$\star$' is '+' (for all the graph algorithms discussed in Section 5.2.3 except shortest paths and connected components) and when '$\star$' is 'min/max' (shortest paths and connected components), respectively.

For an iterative graph algorithm with the operator '$\star$' as '+', we first leverage $\bar{x}^{(*)}$ to construct $\hat{x}^{(0)}$ in the following way: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = 0$. In contrast, *recomputation from scratch* typically utilizes $\mathbf{0}$ as $\hat{x}^{(0)}$ (where $\mathbf{0}$ is a vector with all its elements being

zero). In order to construct $\Delta\hat{x}^{(0)}$, we compute $\hat{x}^{(1)}$ using $\hat{x}^{(1)} = f(\hat{x}^{(0)})$ and then construct $\Delta\hat{x}^{(0)}$ by making sure $\Delta\hat{x}^{(0)}$ satisfying $\hat{x}^{(1)} = \hat{x}^{(0)} \star \Delta\hat{x}^{(0)}$. Since '$\star$' is '+', we can calculate $\Delta\hat{x}^{(0)}$ by $\Delta\hat{x}^{(0)} = \hat{x}^{(1)} - \hat{x}^{(0)}$. It is important to note that here the deleted vertices and/or edges do not affect the way we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$. In other words, no matter whether there are deleted vertices and/or edges, the way we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ can guarantee the correctness of the result on the new graph.

For an iterative graph algorithm with the operator '$\star$' as 'min/max', we construct $\hat{x}_i^{(0)}$ and $\Delta\hat{x}_i^{(0)}$ as follows. When the operator '$\star$' is 'min' (e.g., shortest paths), if any vertex's initial state is not smaller than its final converged state, the algorithm will converge. This is because of the following reason. When the algorithm has not converged, in each iteration there must be at least one vertex whose state is becoming smaller, and thus the overall state vector is becoming closer to the final converged state vector. When there is no vertex changing its state, the algorithm converges. Generally, it is hard to know the final converged state vector. Therefore, for the shortest paths algorithm, recomputation from scratch usually sets the initial state of a vertex (other than the source vertex) as ∞ to guarantee that it is not smaller than the final converged state. Fortunately, when the graph grows (vertices and/or edges are added and no vertices or edges are deleted), the previous converged state of a kept vertex must be not smaller than its converged state on the new graph. Therefore, for the graph growing scenario, we construct $\hat{x}_i^{(0)}$ in the following way: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = \infty$. Similarly, for the connected component algorithm, whose operator '$\star$' is 'max', we can construct $\hat{x}_i^{(0)}$ (for the graph growing scenario) as follows: for a kept vertex (e.g., i), we set $\hat{x}_i^{(0)} = \bar{x}_i^{(*)}$; for a newly added vertex (e.g., j), we set $\hat{x}_j^{(0)} = j$. To construct $\Delta\hat{x}^{(0)}$, we also compute $\hat{x}^{(1)}$ using $\hat{x}^{(1)} = f(\hat{x}^{(0)})$ and then simply set $\Delta\hat{x}_j^{(0)} = \hat{x}_j^{(1)}$. It can satisfy $\hat{x}^{(1)} = \hat{x}^{(0)} \star \Delta\hat{x}^{(0)}$, no matter '$\star$' is 'min' or 'max'.

5.3.2 Selective Execution

One potential problem of basic asynchronous updates is that they might require more computations and communications when compared to their synchronous counterparts. This is because vertices are updated in a round-robin manner no matter how many new values available to a vertex. To solve this problem, instead of updating vertices in a round-robin manner, we update vertices selectively by identifying their importance. The motivation behind it is that not all vertices contributes the same to the convergence. We refer to this scheduling scheme as *selective execution*. The vertices are selected according to their importance (in terms of contribution to the convergence).

Our selective execution scheduling scheme selects a block of m vertices (instead of one) to update each round. The reason is that if only one vertex is chosen to update at a time, the scheduling overhead (e.g., maintaining a priority queue to always choose the vertex with the highest importance) is high. Once the block of the selected vertices are updated, it selects another block to update. Every time our scheme selects the top-m vertices in terms of the importance value. The size of the block (i.e., m) balances the tradeoff between the gain from selective execution and the cost of selecting vertices. Setting m too small may incur considerable overhead, while setting m too large may degrade the effect of selective execution, e.g., if setting m as the number of total vertices, it degrades to the round-robin scheduling. We will discuss how to determine m in Section 5.4.1.

We then illustrate how to quantify a vertex's importance when the operator '$\star$' is 'min/max' and when '$\star$' is '+', respectively. Ideally, the vertex whose update decreases the distance to the fixed point (i.e., $||x^{(*)} - \hat{x}^{(k)}||_1$) most should have the highest importance (note that we use L1-norm to measure the distance). For an iterative graph algorithm with the operator '$\star$' as 'min/max', the iterative updates either monotonically decrease (e.g., shortest paths) or monotonically increase (e.g.,

connected components) any element of $\hat{x}^{(k)}$. For ease of exposition, we assume the monotonically decreasing case. In this case, $x_j^{(*)} \leq \hat{x}_j^{(k)}$ for any j, and thus we have $||x^{(*)} - \hat{x}^{(k)}||_1 = ||\hat{x}^{(k)}||_1 - ||x^{(*)}||_1$. An update on vertex j decrease $||\hat{x}^{(k)}||_1$ by $|\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$. Therefore, we use $|\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$ to represent the importance of the vertex j (denoted as η_j), i.e. $\eta_j = |\hat{x}_j^{(k)} \star \Delta\hat{x}_j^{(k)} - \hat{x}_j^{(k)}|$.

For an iterative graph algorithm with the operator '$\star$' as '+', it is difficult to directly measure how the distance to the fixed point decreases. Update one single vertex may even increase the distance to the fixed point. Fortunately, for such an algorithm, its update function ($f()$) typically can be seen as a $||\cdot||$-*contraction mapping*. That is, there exists an α ($0 \leq \alpha < 1$), such that $||f(x) - f(y)|| \leq \alpha||x - y||, \forall x, y \in R^n$. Therefore, we can provide an upper bound on it, as stated in Theorem 5.3.1. We then analyze how the upper bound decreases.

Theorem 5.3.1. $||x^{(*)} - \hat{x}^{(k+1)}||_1 \leq \frac{||\Delta\hat{x}^{(k+1)}||_1}{1-\alpha}$.

Proof. Consider the situation that from time t_{k+1}, synchronous updates are performed. Let x and Δx to represent the states and the delta states under synchronous updates, respectively. That is, at the beginning, we have $\mathrm{x}^{(0)} = \hat{x}^{(k+1)}$, $\Delta\mathrm{x}^{(0)} = \Delta\hat{x}^{(k+1)}$, and then $\mathrm{x}^{(r+1)} = \mathrm{x}^{(r)} + \Delta\mathrm{x}^{(r)}$, $\Delta\mathrm{x}^{(r+1)} = f(\Delta\mathrm{x}^{(r)})$, where r (≥ 1) is used to index iterations of synchronous updates.

Since $f()$ is a contraction mapping, for $r \geq 1$, we have $||\mathrm{x}^{(r+1)} - \mathrm{x}^{(r)}||_1 = ||f(\mathrm{x}^{(r)}) - f(\mathrm{x}^{(r-1)})||_1 \leq \alpha||\mathrm{x}^{(r)} - \mathrm{x}^{(r-1)}||_1 \leq \alpha^2||\mathrm{x}^{(r-1)} - \mathrm{x}^{(r-2)}||_1 \leq \alpha^r||\mathrm{x}^{(1)} - \mathrm{x}^{(0)}||_1 = \alpha^r||\Delta\mathrm{x}^{(0)}||_1$. Thus,

$$
\begin{aligned}
||\mathrm{x}^{(r)} - \mathrm{x}^{(0)}||_1 &= ||\mathrm{x}^{(r)} - \mathrm{x}^{(r-1)} + \mathrm{x}^{(r-1)} - \cdots + \mathrm{x}^{(1)} - \mathrm{x}^{(0)}||_1 \\
&\leq ||\mathrm{x}^{(r)} - \mathrm{x}^{(r-1)}||_1 + \cdots + ||\mathrm{x}^{(1)} - \mathrm{x}^{(0)}||_1 \\
&\leq \alpha^{r-1}||\Delta\mathrm{x}^{(0)}||_1 + \cdots + \alpha^1||\Delta\mathrm{x}^{(0)}||_1 + ||\Delta\mathrm{x}^{(0)}||_1 \\
&= ||\Delta\mathrm{x}^{(0)}||_1(1 + \alpha + \cdots + \alpha^{r-1}).
\end{aligned}
$$

Since $x^* = \lim_{r\to\infty} \mathrm{x}^{(r)}$, we have

$$\begin{aligned}
||x^{(*)} - \mathrm{x}^{(0)}||_1 &= \lim_{r\to\infty} ||\mathrm{x}^{(r)} - \mathrm{x}^{(0)}||_1 \\
&\leq \lim_{r\to\infty} (||\Delta\mathrm{x}^{(0)}||_1 \sum_{i=0}^{r-1} \alpha) \\
&\leq ||\Delta\mathrm{x}^{(0)}||_1 \sum_{i=0}^{\infty} \alpha^i \\
&= \frac{1}{1-\alpha} ||\Delta\mathrm{x}^{(0)}||_1.
\end{aligned}$$

Since $\mathrm{x}^{(0)} = \hat{x}^{(k+1)}$, $\Delta\mathrm{x}^{(0)} = \Delta\hat{x}^{(k+1)}$, we have $||x^{(*)} - \hat{x}^{(k+1)}||_1 \leq \frac{||\Delta\hat{x}^{(k+1)}||_1}{1-\alpha}$, which concludes the proof. □

Without loss of generality, assume that current time is t_k and that during interval $[t_k, t_{k+1}]$ we only update vertex j. When updating vertex j, we accumulate $\Delta\hat{x}_j^{(k)}$ to $\hat{x}_j$, send $f_{(j,l)}(\Delta\hat{x}_j^{(k)})$ to a vertex l (and the total sending out value is no larger than $\alpha|\Delta\hat{x}_j^{(k)}|$), and reset $\Delta\hat{x}_j^{(k)}$ to 0. Therefore, we have the following theorem.

Theorem 5.3.2. $||\Delta\hat{x}^{(k+1)}||_1 \leq ||\Delta\hat{x}^{(k)}||_1 - (1-\alpha)|\Delta\hat{x}_j^{(k)}|$.

Theorem 5.3.2 implies that the upper bound monotonically decreases as updates continue. When updating vertex j, we have $\frac{||\Delta\hat{x}^{(k+1)}||_1}{1-\alpha} \leq \frac{||\Delta\hat{x}^{(k)}||_1}{1-\alpha} - |\Delta\hat{x}_j^{(k)}|$. It shows that the reduction in the upper bound is at least $|\Delta\hat{x}_j^{(k)}|$. Given a graph, α is a constant. Hence, when the operator '$\star$' is '+', we define the importance of the vertex j to be $|\Delta\hat{x}_j^{(k)}|$, i.e., $\eta_j = |\Delta\hat{x}_j^{(k)}|$.

5.3.3 Convergence

Our asynchronous incremental computation approach yields the same result as recomputation from scratch. In order to prove it, we first show that if synchronous updates (i.e., $x^{(k)} = f(x^{(k-1)})$) converge (and synchronous updates converge for all the graph algorithms discussed in Section 5.2.3), any asynchronous update scheme

that can guarantee that every vertex is updated infinitely often (until its state is fixed) will yield the same result as synchronous updates, as stated in Theorem 5.3.3.

Theorem 5.3.3. *If updates $x^{(k)} = f(x^{(k-1)})$ converge to $x^{(*)}$, any asynchronous update scheme that guarantees every vertex is updated infinitely often will converge to $x^{(*)}$ as well, i.e., $\hat{x}^{(\infty)} = x^{(*)}$.*

Since synchronous updates converge to $x^{(*)}$, i.e., $x^{(\infty)} = x^{(*)}$, we are going to show that asynchronous updates (no matter the order of updates) yield the same results as synchronous updates in the following. We first illustrate how the state vector is computed under different updates.

By synchronous updates, x_j after k iterations is:

$$x_j^{(k)} = x_j^{(0)} \star \Delta x_j^{(1)} \star \sum_{l=2}^{k} \star \Big(\sum_{\{i \twoheadrightarrow j\} \in P(j,l)} \star f_{\{i \twoheadrightarrow j\}}(\Delta x_i^{(1)}) \Big), \tag{5.3}$$

where

$$f_{\{i \twoheadrightarrow j\}}(\Delta x_i^{(1)}) = f_{\{i_{l-1},j\}}(\cdots f_{\{i_1,i_2\}}(f_{\{i_0,i_1\}}(\Delta x_{i_0}^{(1)})))$$

and $P(j,l)$ is the set of all l-hop paths to reach vertex j.

We define $S = \{S_0, S_1, \ldots, S_\infty\}$ as the series of vertex subsets, where S_k is a vertex subset, and the propagated values of all vertices in S_k have been received by their direct neighboring vertices during time interval $[t_k, t_{k+1}]$.

By asynchronous updates, following an activation sequence S, $\hat{x}_j$ at time t_k is:

$$\hat{x}_j^{(k)} = x_j^{(0)} \star \Delta x_j^{(1)} \star \sum_{l=2}^{k} \star \Big(\sum_{\{i \twoheadrightarrow j\} \in P'(j,l)} \star f_{\{i \twoheadrightarrow j\}}(\Delta x_i^{(1)}) \Big), \tag{5.4}$$

where $P'(j,l)$ is the set of l-hop paths that satisfy the following conditions. First, $i \in S_l$. Second, if $l > 1$, $i_1, \ldots, i_{l-1}$ respectively belongs to the sequence S. That is, there is $0 < m_1 < m_2 < \ldots < m_{l-1} < k$ such that $i_h \in S_{m_{l-h}}$.

We first consider the iterative graph algorithm with the operator '$\star$' as '+'. We know that the elements of $x^{(0)}$ can be nonnegative or negative. Therefore, we can divide it into two parts, nonnegative part $y^{(0)}$ and negative part $z^{(0)}$. Let $y^{(k)}$ and $z^{(k)}$ represent the values generated from $y^{(0)}$ and $z^{(0)}$, respectively, and $x_j^{(k)} = y_j^{(k)} + z_j^{(k)}$ for any k. Then, we have

$$y_j^{(k)} = y_j^{(0)} + \Delta y_j^{(1)} + \sum_{l=2}^{k} \star (\sum_{\{i \twoheadrightarrow j\} \in P(j,l)} \star f_{\{i \twoheadrightarrow j\}}(\Delta y_i^{(1)})),$$

$$z_j^{(k)} = z_j^{(0)} + \Delta z_j^{(1)} + \sum_{l=2}^{k} \star (\sum_{\{i \twoheadrightarrow j\} \in P(j,l)} \star f_{\{i \twoheadrightarrow j\}}(\Delta z_i^{(1)})).$$

Next, we show for asynchronous updates that $\hat{y}^{(k)}$ converges to $y^{(*)}$ (i.e., $y^{(\infty)}$) and that $\hat{z}^{(k)}$ also converges to $z^{(*)}$ (i.e., $z^{(\infty)}$). To this end, we introduce the following two lemmas.

Lemma 5.3.4. *For any infinite sequence S (i.e., each vertex can have an infinite number of updates in the sequence), given any iteration number k, we can find a subset index k' in S such that $|y_j^{(\infty)} - \hat{y}_j^{(k')}| \geq |y_j^{(\infty)} - y_j^{(k)}|$ for any j.*

Proof. From Eq. (5.3), we can see that, after k iterations of synchronous updates, each vertex receives the values from its direct/indirect neighbors as far as k hops away, and it receives the values originated from each direct/indirect neighbor once for each path. In other words, each vertex j propagates its own initial value $x_j^{(1)}$ and receives the values from its direct/indirect neighbors through a path once.

From Eq. (5.4), we can see that, for asynchronous updates, after time t_k, each vertex receives values from its direct/indirect neighbors as far as k hops away, and it receives values originated from each direct/indirect neighbor through a path at most once. During time period $[t_{k-1}, t_k]$, a value is received from a neighbor only if the neighbor is in S_k. If the neighbor is not in S_k, the value is stored at the neighbor or

is on the way to other vertices. The vertex will eventually receive the value as long as every vertex has an infinite number of updates.

As a result, $\hat{x}_j^{(k)}$ receives values via a subset of the paths from j's direct/indirect incoming neighbors within k hops. In contrast, $x_j^{(k)}$ receives values through all paths from j's direct/indirect incoming neighbors within k hops. Considering only the nonnegative part of the value, we can see that $\hat{x}_j^{(k)}$ receives less or equal nonnegative parts compared to $x_j^{(k)}$. Correspondingly, $\hat{y}_j^{(k)}$ is farther (or at least equal) to the converged point $y_j^{(\infty)}$ than $y_j^{(k)}$. Therefore, we can set $k' = k$ and complete the proof. □

Lemma 5.3.5. *For any infinite sequence S, given any iteration number k, we can find a subset index k'' in S such that $|y_j^{(\infty)} - \hat{y}_j^{(k'')}| \leq |y_j^{(\infty)} - y_j^{(k)}|$ for any j.*

Proof. From the proof of Lemma 5.3.4, we know that $y_j^{(k)}$ receives values from all paths from direct/indirect neighbors of j within k hops away. In order to allow $\hat{y}_j^{(k)}$ to receive all those values, we have to make sure that all paths from direct/indirect neighbors of j within k hops away are activated and their values are received. Since in sequence S each vertex can have an infinite number of updates, we can always find k'' such that $\{S_1, S_2, \ldots, S_{k''}\}$ contains all paths from direct and indirect neighbors of j within k hops away. Considering only the nonnegative part of the value, we can see that $\hat{y}_j^{(k)}$ receives more or equal nonnegative parts compared to $y_j^{(k)}$. Correspondingly, $\hat{y}_j^{(k'')}$ can be closer (or at least equal) to the converged point $y_j^{(\infty)}$ than $y_j^{(k)}$. Therefore, we complete the proof. □

From Lemma 5.3.4 and Lemma 5.3.5, it is easy to see that $\hat{y}_j^{(k)}$ and $y_j^{(k)}$ converge to the same result, i.e., $\hat{y}_j^{(\infty)} = y_j^{(\infty)}$. Similarly, we can show that $\hat{z}_j^{(k)}$ and $z_j^{(k)}$ converge to the same result, i.e., $\hat{z}_j^{(\infty)} = z_j^{(\infty)}$. Also, we know $\hat{x}_j^{(\infty)} = \hat{y}_j^{(\infty)} + \hat{z}_j^{(\infty)}$, $x_j^{(\infty)} = y_j^{(\infty)} + z_j^{(\infty)}$, and $x_j^{(\infty)} = x_j^{(*)}$. Thus, we have $\hat{x}_j^{(\infty)} = x_j^{(\infty)} = x_j^{(*)}$. For the iterative graph algorithm with the operator '$\star$' as 'min/max', we know that all of

the elements of $x^{(0)}$ are nonnegative (or negative). Hence, we can use Lemma 5.3.4 and Lemma 5.3.5 to show $\hat{x}_j^{(\infty)} = x_j^{(\infty)} = x_j^{(*)}$. As a result, we complete the proof of Theorem 5.3.3.

We then show that our asynchronous incremental computation approach fulfills the requirement of Theorem 5.3.3, as stated in Lemma 5.3.6.

Lemma 5.3.6. *Our asynchronous incremental computation approach can guarantee that every vertex is updated infinitely often (until its state is fixed).*

Proof. We prove this lemma by contradiction. Assume there are a number of vertices that belong to a set, W, which are updated only before a time point t. For the iterative graph algorithm with the operator '$\star$' as '+', its update function is a contraction mapping. For the iterative graph algorithm with the operator '$\star$' as 'min/max', we know that any element of its state vector monotonically decreases (or increases). Therefore, no matter what kind of iterative graph algorithm, the L1-norm of the delta states of the vertices in $V - W$ (where V is the whole set of vertices), $||\Delta\hat{x}_{V-W}||_1$ approaches 0 as updates continue (and thus $|\Delta\hat{x}_i|$ approaches 0 for any $i \in (V-W)$). Consequently, at some time point after t, for any vertex that belongs to $V - W$, it can reach the fixed state since it is always being updated. At that time, for any $i \in (V - W)$, we have $|\Delta\hat{x}_i| = 0$; if we also have $|\Delta\hat{x}_i| = 0$ for any $i \in W$, then $||\Delta\hat{x}||_1 = 0$, and thus the graph algorithm has converged; otherwise, a vertex in W (e.g., the one with the highest importance) must be selected to update, which contradicts with the assumption that any vertex in W is updated only before time point t. We complete the proof. □

Furthermore, we can also prove that recomputation from scratch converges to $x^{(*)}$ (no matter what type of updates it uses). As a result, we have the following theorem.

Theorem 5.3.7. *Our asynchronous incremental computation approach converges and yields the same result as recomputation from scratch.*

5.4 Distributed Framework

Oftentimes, iterative graph algorithms in real-world applications need to process massive graphs. Hence, it is desirable to leverage the parallelism of a cluster of machines to run these algorithms. Furthermore, it is troublesome to implement asynchronous incremental computation for each individual algorithm that can operate efficiently on dynamic graph data in a distributed environment. Therefore, we propose GraphIn, an in-memory asynchronous distributed framework for supporting iterative graph algorithms with incremental computation. GraphIn provides several high-level APIs to users for implementing asynchronous incremental computation and meanwhile hides the complexity of distributed computation. It leverages the proposed selective execution scheduling scheme to accelerate convergence.

GraphIn consists of a number of workers and one master. Workers perform vertex updates, and the master controls the flow of computation. The new graph and the previous computed result are taken as the input of GraphIn. The input graph is split into partitions and each worker is responsible for one partition. Each worker leverages an in-memory table to store the vertices assigned to it. A worker has two main operations for its stored vertices: the accumulate operation and the update operation, as illustrated in Section 5.3.1. The accumulate operation utilizes a user-defined function to aggregate incoming messages for a vertex. There is another user-defined function triggered by the accumulate operation, which is used to calculate the vertex's importance. The update operation uses a user-defined function to update the states of scheduled vertices and compute outgoing messages.

The prototype of GraphIn is built upon Maiter [107], an open-source distributed graph processing framework. Maiter is designed for processing static graphs, and thus has inherent impediments to the execution of graph algorithms with incremental computation. First, it relies on the specific initial state to guarantee the convergence of a graph algorithm. However, incremental computation leverages the previous result as

the initial state, which can be arbitrary. Second, although Maiter supports prioritized updates, its scheduling scheme assumes that Δx_i is always positive for any vertex i. Last, the termination check mechanism of Maiter assumes that $||x||_1$ varies monotonically, which can be not true as well under incremental computation. GraphIn removes all these impediments to efficiently support incremental computation.

5.4.1 Distributed Selective Execution

GraphIn uses the proposed selective execution scheduling as its default scheduling scheme. Since a centralized approach of finding the top-m elements is inefficient in a distributed environment, GraphIn allows each worker to build its own selective execution scheduling. Round by round (except the first round in which all vertices are selected to derive $\hat{x}^{(0)}$ and $\Delta\hat{x}^{(0)}$), each worker selects its local top-m vertices in terms of the importance. The number m is crucial to the effect of selective execution.

For the iterative graph algorithm with the operator '$\star$' as '+', GraphIn learns m online. We use $\mu \cdot n$ to quantify the overhead of selecting such m vertices (where μ represents the amortized overhead), which is proportional to the total number (n) of vertices with an efficient selection algorithm (e.g., quick-select). Also, we assume that the average cost of updating one vertex is ν, and then the cost of updating those m vertices is $\nu \cdot m$. Let $c(m)$ be the total cost of updating those m vertices (including both selection and update), then $c(m) = \mu \cdot n + \nu \cdot m$. Let $g(m) = \sum_{j \in S} |\Delta\hat{x}_j|$ (recall that $|\Delta\hat{x}_j|$ represents the importance of vertex i), where S denotes the set of the top-m selected vertices. For each round, we aim to find the m that can achieve the largest efficiency, i.e., $m = \arg\max_m \frac{g(m)}{c(m)}$. It is computationally impossible to try every value (from 1 to n) to figure out the best m. Therefore, our practical approach chooses several values ($0.05n$, $0.1n$, $0.25n$, $0.5n$, n), which cover the entire range of possible m, as the candidates. For each candidate m, we leverage quick-select to find the m-th $|\Delta\hat{x}_j|$, which is used as a threshold, and all $|\Delta\hat{x}_i|$ no less than the threshold

are counted into $g(m)$. By testing each candidate (we set ν/μ as 4 by default), we can figure out the best m and the set S. The practical approach leverages quick-select to avoid the time-consuming sorting, and thus takes $O(n)$ time on extracting the top-m vertices instead of $O(n \log n)$ time. For the iterative graph algorithm with the operator '$\star$' as 'min/max', the importance of a vertex might be close to ∞. If we still use the above idea, $g(m)$ might easily be overflown. Therefore, in this case, we simply set m as $0.1n$, which shows good performance in experiments. Note that if there are only m' ($m' < m$) vertices with the importance being larger than 0, we only select these m' vertices to update.

5.4.2 Distributed Termination Check

We design different termination check mechanisms for the iterative graph algorithm with the operator '$\star$' as 'min/max' and the iterative graph algorithm with the operator '$\star$' as '+'. When '$\star$' is 'min/max', $||\hat{x}^{(k)}||_1$ monotonically decreases or increases. Therefore, we can utilize $||\hat{x}^{(k)}||_1$ to perform the termination check. If and only if $||\hat{x}^{(k)}||_1 - ||\hat{x}^{(k-1)}||_1 = 0$, the algorithm has converged, and thus the computation can be terminated. When '$\star$' is '+', $||x^{(*)} - \hat{x}^{(k)}||_1$ is the choice for measuring convergence. However, it is difficult to directly quantify $||x^{(*)} - \hat{x}^{(k)}||_1$, since the fixed point $x^{(*)}$ is always unknown during the computation. Fortunately, we know $||x^{(*)} - \hat{x}^{(k)}||_1 \leq ||\Delta\hat{x}^{(k)}||_1/(1-\alpha)$ from Theorem 5.3.1, and thus can leverage $||\Delta\hat{x}^{(k)}||_1$ to measure convergence. We use the convergence criterion, $||\Delta\hat{x}^{(k)}||_1 \leq \epsilon$, where the convergence tolerance ϵ is a pre-defined constant.

GraphIn adopts a passively monitoring model to perform the termination check, which works by periodically (and the period is configurable) measuring $||\hat{x}^{(k)}||_1$ if the operator '$\star$' is '+' (or $||\Delta\hat{x}^{(k)}||_1$ if '$\star$' is 'min/max'). To complete the measure, each worker computes the sum of $|\hat{x}_j^{(k)}|$ (or $|\Delta\hat{x}_j^{(k)}|$) of its local vertices and sends the local sum to the master. The master aggregates the local sums into a global sum. The

challenge of performing such a distributed termination is to make sure that the local sum at each worker are calculated from the snapshot of the values at the same time (especially for $|\Delta\hat{x}_j^{(k)}|$). To address the challenge, GraphIn asks all the workers to pause vertex updates before starting to calculate the local sums. The procedure of the distributed termination check is as follows.

1. When it is the time to perform the termination check, the master broadcasts a chk_{pre} message to all the workers.
2. Upon receiving the chk_{pre} message, every worker pauses vertex updates and then replies a chk_{ready} message to the master.
3. The master gathers those chk_{ready} messages from all the workers, and then broadcasts a chk_{begin} message to them.
4. Upon receiving the chk_{begin} message, every worker calculates the local sum, $\sum_j |\hat{x}_j^{(k)}|$ (or $\sum_j |\Delta\hat{x}_j^{(k)}|$), and reports it to the master.
5. The master aggregates the local sums to the global sum $||\hat{x}^{(k)}||_1$ (or $||\Delta\hat{x}^{(k)}||_1$). If $||\hat{x}^{(k)}||_1 - ||\hat{x}^{(k-1)}||_1 \neq 0$ (or $||\Delta\hat{x}^{(k)}||_1 > \epsilon$), the master broadcasts a chk_{fin} message to all the workers. Otherwise, it broadcasts a *term* message.
6. When a worker receives the chk_{fin} message, it resumes vertex updates. When a worker receives the *term* message, it dumps the result to a local disk and then terminates the computation.

It is important to note that since calculating the local sums is inexpensive and it is done periodically, the overhead of the termination check is ignorable.

5.5 Evaluation

In this section, we evaluate the performance of our asynchronous incremental computation approach. We compare it with re-computation from scratch. Both approaches are supported by GraphIn. To show the performance of the selective execution scheduling, we compare it with the round-robin scheduling. The performance

of other distributed systems that can support synchronous incremental computation are also evaluated.

5.5.1 Experiment Setup

The experiments are performed on both a local cluster and a large-scale cluster on Amazon EC2 [1]. The local cluster consists of 4 machines, which are connected through a switch with a bandwidth of 1Gbps. The large-scale cluster consists of 50 EC2 medium instances.

Table 5.1. Graph Dataset Summary

Dataset	Vertices	Edges
Amazon co-purchasing graph (Amz) [52]	403K	3.4M
Web graph from Google (Gog) [52]	876K	5.1M
LiveJournal social network (LJ) [52]	4.8M	69M
Web graph from UK (UK) [14]	39M	936M
Web graph from IT (IT) [14]	41M	1.2B

Two graph algorithms are implemented on GraphIn, PageRank and the shortest paths algorithm. For PageRank, the damping factor is set to 0.8, and if not stated otherwise, the convergence tolerance ϵ (which is discussed in Section 5.4.2) is set to $10^{-2}/n$ (n is the number of vertices of the corresponding graph). The shortest paths algorithm stops running only when the convergence point is reached (i.e., all the vertices reach their shortest paths to the source vertex). The measurement of each experiment is averaged over 10 runs. Real-world graphs of various sizes are used in the experiments and are summarized in Table 5.1.

5.5.2 Overall Performance

We first show the convergence time of PageRank on the local cluster. The convergence time is measured as the wall-clock time that PageRank uses to reach the convergence criterion. We consider both the edge change case and the vertex change case. Under the edge change case, we randomly pick a number of vertices to change

their edges. In the graph evolving process, there are usually more added edges than deleted edges. Therefore, for 80% of the picked vertices, we add one random outgoing edge to it with a randomly picked neighbor. For the rest 20% vertices, we remove one randomly picked edge from it. Under the vertex change case, we pick a number (e.g., p, some percentage of the number of vertices) for each experiment. We add $0.8p$ new vertices to the graph and delete $0.2p$ vertices. For each added vertex, we put two edges (one incoming edge and one outgoing edge) with randomly picked neighbors. For each deleted vertex, we also delete all its edges.

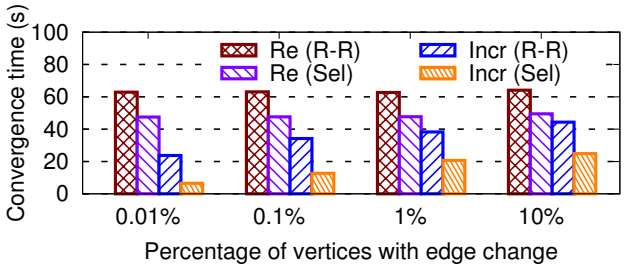


Figure 5.1. PageRank on Amz graph (edge change).

Figure 5.1 shows the performance on the Amz graph under the edge change case. We can see that incremental computation (denoted as "Incr") is much faster than re-computation from scratch (denoted as "Re") for different percentages of vertices with edge change. The selective execution scheduling (denoted as "Sel") is faster than the round-robin scheduling (denoted as "R-R") with either approach. The efficiency of the incremental computation is more prominent when the change is smaller. For example, when the percentage of vertices with edge change is 0.01%, incremental computation with the selective execution scheduling is about 10x faster than recomputation from scratch with the round-robin scheduling and 7x faster than recomputation from scratch with the selective execution scheduling. Not surprisingly, the incremental computation takes longer time as the percentage of vertices with edge change becomes larger, and the convergence time of the re-computation is almost the same since the change to the graph is relatively small. Similar trends are observed for the vertex change case, as plotted in Figure 5.2.

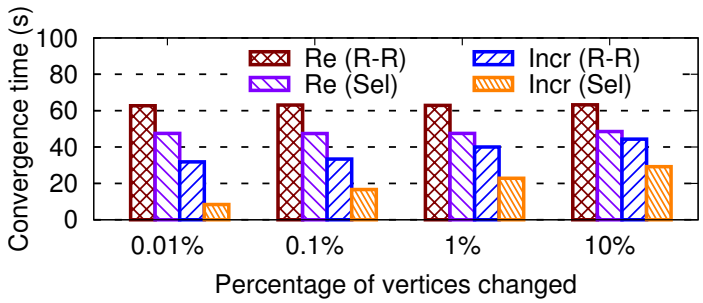


Figure 5.2. Pagrank on Amz graph (vertex change).

We then present the result of the shortest paths algorithm, which runs on weighted graphs. All the graphs summarized in Table 5.1 are unweighed. We generate a weighted graph by assigning weights to the Amz graph. The weight of each edge is an integer, which is randomly drawn from the rang $[1, 100]$. Figure 5.3 plots the performance comparison under the vertex adding case. The percentage means the ratio between the number of added vertices to the number of original vertices. For each added vertex, we put two weighted edges (one incoming edge and one outgoing edge) with randomly picked neighbors. From the figure, we can see that incremental computation with the selective execution scheduling is about 14x faster than recomputation from scratch with the round-robin scheduling when the percentage of added vertices is 0.01% and still 9x faster even when the percentage of added vertices is 10%. Similar results are observed for the edge adding case.

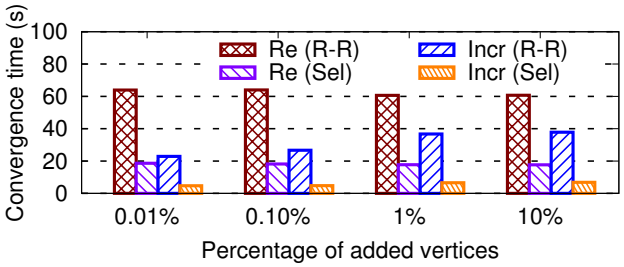
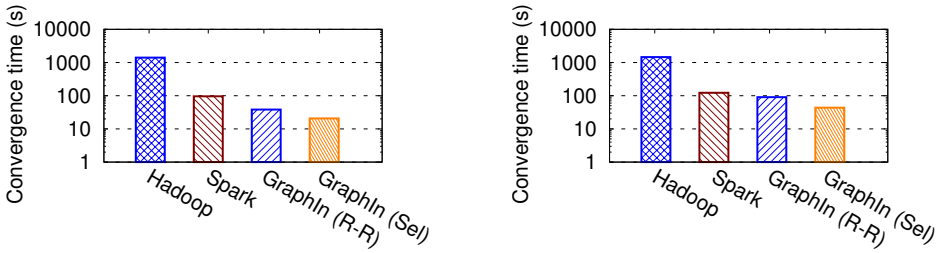


Figure 5.3. Shortest paths on weighted Amz graph.

5.5.3 Comparison with Synchronous Incremental Computation

It is also possible to build a framework to support incremental computation upon other systems, such as Hadoop and Spark. To demonstrate the efficiency of GraphIn, we compare it with both Hadoop and Spark for the 1% of vertices with edge change scenario. We restrict our performance comparison to PageRank, since it is a representative graph algorithm. For fair comparison, we instruct both systems to use the prior result as the starting point. For Hadoop, if there is no change in the input of some Map/Reduce tasks, we proportionally discount the running time. In this way, we can simulate task-level reusing, which is the key of MapReduce-based incremental processing frameworks. For Spark, we choose its Graphx [31] component to implement PageRank.



(a) Amz (b) Gog

Figure 5.4. PageRank on different frameworks.

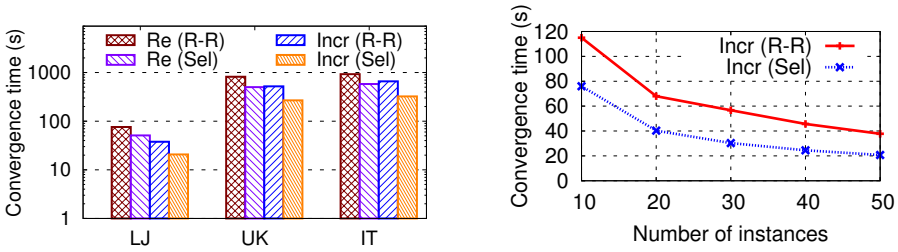
Figure 5.4 shows that GraphIn (especially with selective execution) is much faster than Hadoop and Spark. Hadoop is a disk-based system and uses synchronous updates. Even though Spark is a memory-based system, it also utilizes synchronous updates. Therefore, it is still slower than GraphIn.

5.5.4 Scaling Performance

We further evaluate incremental computation on the large-scale Amazon cluster to test its scalability. We consider the 1% of vertices with edge change scenario, and concentrate on PageRank (and set the convergence tolerance ϵ to 10^{-4}). We first use the three large real-world graphs, LJ, UK, and IT (both UK and IT have tens of

millions of vertices and a billion of edges), as input graphs when all the 50 instances are used. As shown in Figure 5.5(a) (note that the y-axis is in log scale), on the large-scale cluster incremental computation is still much faster than re-computation from scratch, and both approaches can benefit from the selective execution scheduling.

We then show the performance of incremental computation when different numbers of instances are used. Figure 5.5(b) shows the convergence time on the LJ graph as we increase the number of instances from 10 to 50. It can be seen that by increasing the number of instances, the convergence time is reduced, and that the selective execution scheduling is always faster than the round-robin scheduling.



(a) 50 instances (b) Varying instance number

Figure 5.5. Performance on Amazon cluster.

5.6 Related Work

Due to the dynamic nature of graphs in real-world applications, incremental computation has been studied extensively. In terms of iterative graph algorithms, most of the studies [7,46,47] focus on PageRank. The basic idea behind approaches in [46,47] is that when a change happens in the graph, the effect of the change on the PageRank scores is mostly local. These approaches start with the exact PageRank scores of the original graph but provide approximate scores for the graph after the change, and the estimations may drift away from the exact scores. On the contrary, our approach can provide exact scores. The work in [7] utilizes the Monte Carlo method to approximate PageRank scores on evolving graphs. It precomputes a number of random walk

segments for each vertex and stores them in distributed shared memory. Besides of the approximate result, it also incurs high memory overhead.

In recent years, the growing scale and importance of graph data have driven the development of a number of distributed graph systems. Pregel [64] employs a vertex-centric programming model and follows the Bulk Synchronous Parallel (BSP) computation model. Graphx [31] is a graph system built on top of Spark [101]. It stores graphs as tabular data and implements graph operations using distributed joins. PrIter [106], Maiter [107], and Prom [93], introduce prioritized updates to accelerate convergence. PrIter is a MapReduce-based framework, which requires synchronous iterations. Maiter and Prom utilize asynchronous accumulative iterative computation, which accumulates the intermediate iterative update results to accelerate convergence. All these graph systems aim at supporting graph computation on static graph structure.

There are several systems for supporting incremental parallel processing on massive datasets. Incoop [11] extends the MapReduce programming model to support incremental processing. It saves and reuses states at the granularity of individual Map or Reduce tasks. Continuous bulk processing (CBP) [60] provides a groupwise processing operator to reuse prior state for incremental analysis. Similarly, other systems like DryadInc [73] support incremental processing by allowing their applications to reuse prior computation results. However, most of the studies focus on one-pass applications rather than iterative applications. Several recent studies address the need of incremental processing for iterative applications. Kineograph [17] constructs incremental snapshots of the evolving graph and supports reusing prior states in processing later snapshots. Naiad [69] presents a timely dataflow computational mode, which allows stateful computation and nested iterations. Spark Streaming [102] extends the cyclic batch dataflow of original Spark to allow dynamic modification of the dataflow and thus supports iterative and incremental processing. However, most

of these systems apply synchronous updates to incremental computation. Our work illustrates how to efficiently apply asynchronous updates to incremental computation.

5.7 Conclusion

In this chapter, we propose an approach to efficiently apply asynchronous updates to incremental computation on evolving graphs. Our approach works for a family of iterative graph algorithms. We also present a scheduling scheme, selective execution, to coordinate asynchronous updates so as to accelerate convergence. Furthermore, to facilitate the implementation of iterative graph algorithms with incremental computation in a distributed environment, we design and implement an asynchronous distributed framework, GraphIn. The evaluation results show that our asynchronous incremental computation approach can significantly boost the performance.

CHAPTER 6

CONCLUSION

This dissertation explores new forms of iterative computations that reduce unnecessary computations so as to accelerate large-scale data processing in a distributed environment. We propose the iterative computation transformation for well-known data mining and machine learning algorithms such as expectation-maximization, nonnegative matrix factorization, belief propagation, and graph algorithms.

First, we apply frequent updates on Expectation-Maximization (EM) algorithms in a distributed environment. Because of the popularity of EM algorithms, many approaches for accelerating EM algorithms have been proposed. In particular, many EM algorithms that frequently update the parameters have been shown to be much more efficient than their concurrent counterparts. Accordingly, we propose two approaches to parallelize such EM algorithms in a distributed environment so as to scale to massive datasets. Based on the approaches, we design and implement a distributed framework, FreEM, to support the implementation of frequent updates for the EM algorithms. We show its efficiency through two categories of EM algorithms, clustering and topic modeling. These algorithms includes k-means clustering, fuzzy c-means clustering, parameter estimation for the Gaussian Mixture Model, and variational inference for Latent Dirichlet Allocation. Our evaluation shows that the EM algorithms with frequent updates implemented on FreEM can converge much faster than those implementations with traditional concurrent updates.

Second, block-wise updates are proposed for nonnegative matrix factorization (NMF) algorithms. As NMF is increasingly applied to massive datasets such as

web-scale dyadic data, it is desirable to leverage a cluster of machines to store those datasets and to speed up the factorization process. However, it is challenging to efficiently implement NMF in a distributed environment. We show that by leveraging a new form of update functions, we can perform local aggregation and fully explore parallelism. Therefore, the new form is much more efficient than the traditional form in distributed implementations. Furthermore, we propose frequent block-wise updates, which aim to use the most recently updated data whenever possible. As a result, frequent block-wise updates can further improve the performance, compared with their traditional concurrent counterparts. Through a series of experiments on a local cluster as well as the Amazon EC2 cloud, we demonstrate that our implementation with frequent updates is up to two orders of magnitude faster than the existing implementation with the traditional form of update functions.

Third, we introduce an efficient dynamic scheduling scheme, the prioritized block scheduling, for belief propagation (BP) algorithms. The proposed scheduling scheme selects a set of messages to update at a time and leverages a novel priority to determine which messages are selected. In order to efficiently compute the priority and update messages, we introduce an incremental-update approach, which is much more efficient than the traditional basic-update approach. As the size of the model grows, it is desirable to leverage the parallelism of a cluster of machines to reduce the inference time. Therefore, we design a distributed framework, Prom, to facilitate the implementation of BP algorithms. We implement two BP algorithms, the sum-product algorithm and the max-product algorithm, on Prom. The evaluation results show that the prioritized block scheduling outperforms the state-of-the-art dynamic scheduling scheme, and that the incremental-update approach can further accelerate the prioritized block scheduling.

Lastly, we present an approach to efficiently apply asynchronous updates to incremental computation on evolving graphs. Asynchronous incremental computation

can bypass synchronization barriers and always utilize the most recent values, and thus it is more efficient than its synchronous counterpart. Our approach works for a broad family of iterative graph algorithms. Furthermore, we develop a distributed framework, GraphIn, to facilitate implementations of incremental computation on massive evolving graphs. We evaluate our asynchronous incremental computation approach via extensive experiments on a local cluster as well as the Amazon EC2 cloud. The evaluation results show that our asynchronous incremental computation approach can significantly boost the performance.

The work presented in this dissertation also open several possible directions for future work. We discuss these possible directions in the following.

For applying frequent updates on EM algorithms, we have discussed the size of the block/subrange plays an important role on the efficiency. Currently, the size is fixed in all workers across iterations. A more thorough study could be done to derive algorithms that dynamically adjust the block/subrange size across iterations in order to achieve better performance. Moreover, the derived algorithms should also allow each worker to have its own block/subrange size based on its capacity. For example, a more powerful worker could have a larger block/subrange size.

The frequent block-wise updates scheme proposed for NMF algorithms in this dissertation takes the advantage of skipping unnecessary matrix computations. However, only adjusting frequency of updates (by changing the block size) might not fully take this advantage, since all of the blocks are still updated in a round-robin manner. A possible future direction is to study how to dynamically choose blocks to further improve efficiency. To this end, one might need to compute the loss value associated with each block and only update the blocks with larger loss values.

The work on BP algorithms (and Prom) also open new directions for future research. Although GraphLab as a representative asynchronous graph processing framework is discussed, a more in-depth discussion and comparison to other graph

processing frameworks can be included. It is interesting to see whether the proposed scheduling approach could be implemented as part of other frameworks. Moreover, it is also interesting to include results from a larger distributed deployment in order to show the scalability limits of the proposed approach and the framework (i.e., Prom).

Although the asynchronous incremental computation approach studied in this dissertation cover a range of graph algorithms, we still lack a systematic and practical way to accommodate an exhausted list of graph algorithms. Taking into account that graph mining algorithms, such as subgraph mining, dense subgraph discovery, community detection, and graph clustering, are also very useful. It is challenging and important to elaborate on how to apply asynchronous incremental computation on these algorithms.

While the proposed techniques are in the context of specific algorithm domains, they may also address the challenges faced in many other algorithm domains. The core ideas of the techniques to leverage iterative computation transformations to accelerate large-scale data processing in a distributed environment. Iterative computations are common in many algorithm domains (even beyond data mining and machine learning). We believe that the ideas presented in this dissertation can be applied to other algorithm domains as well.

BIBLIOGRAPHY

[1] Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[2] Hadoop. `http://hadoop.apache.org/`.

[3] UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`.

[4] UW-CSE MLN. `http://alchemy.cs.washington.edu/mlns/uw-cse/`.

[5] Akoglu, Leman, Khandekar, Rohit, Kumar, Vibhore, Parthasarathy, Srinivasan, Rajan, Deepak, and Wu, Kun-Lung. Fast nearest neighbor search on large time-evolving graphs. In *ECML/PKDD' 14* (2014), pp. 17–33.

[6] Avrachenkov, Konstantin, and Lebedev, Dmitri. Pagerank of scale-free growing networks. *Internet Math. 3*, 2 (2006), 207–232.

[7] Bahmani, Bahman, Chowdhury, Abdur, and Goel, Ashish. Fast incremental and personalized pagerank. *Proc. VLDB Endow. 4*, 3 (Dec. 2010), 173–184.

[8] Baluja, Shumeet, Seth, Rohan, Sivakumar, D., Jing, Yushi, Yagnik, Jay, Kumar, Shankar, Ravichandran, Deepak, and Aly, Mohamed. Video suggestion and discovery for youtube: taking random walks through the view graph. In *WWW '08* (2008), pp. 895–904.

[9] Bertsekas, DimitriP. Distributed asynchronous computation of fixed points. *Mathematical Programming 27*, 1 (1983), 107–120.

[10] Bezdek, James C. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Kluwer Academic Publishers, 1981.

[11] Bhatotia, Pramod, Wieder, Alexander, Rodrigues, Rodrigo, Acar, Umut A., and Pasquin, Rafael. Incoop: Mapreduce for incremental computations. In *SoCC '11* (2011).

[12] Blei, David M., Ng, Andrew Y., and Jordan, Michael I. Latent dirichlet allocation. *J. Mach. Learn. Res. 3* (Mar. 2003), 993–1022.

[13] Bogdanov, Petko, and Singh, Ambuj. Accurate and scalable nearest neighbors in large networks based on effective importance. In *CIKM '13* (2013), pp. 1009–1018.

[14] Boldi, Paolo, and Vigna, Sebastiano. The WebGraph framework I: Compression techniques. In *WWW '04* (2004), pp. 595–601.

[15] Brin, Sergey, and Page, Lawrence. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems 30*, 1–7 (1998), 107–117.

[16] Brunet, Jean-Philippe, Tamayo, Pablo, Golub, Todd R., and Mesirov, Jill P. Metagenes and molecular pattern discovery using matrix factorization. *PNAS 101*, 12 (Mar. 2004), 4164–4169.

[17] Cheng, Raymond, Hong, Ji, Kyrola, Aapo, Miao, Youshan, Weng, Xuetian, Wu, Ming, Yang, Fan, Zhou, Lidong, Zhao, Feng, and Chen, Enhong. Kineograph: Taking the pulse of a fast-changing and connected world. In *EuroSys '12* (2012), pp. 85–98.

[18] Cheng, Xiang, Su, Sen, Gao, Lixin, and Yin, Jiangtao. Co-ClusterD: A distributed framework for data co-clustering with sequential updates. *IEEE Trans. Knowl. Data Eng. 27*, 12 (2015), 3231–3244.

[19] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[20] Crick, Christopher, and Pfeffer, Avi. Loopy belief propagation as a basis for communication in sensor networks. In *UAI '03* (2003), pp. 159–166.

[21] Darwiche, Adnan. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[22] Dean, Jeffrey, and Ghemawat, Sanjay. MapReduce: Simplified data processing on large clusters. In *OSDI '04* (2004), pp. 107–113.

[23] Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B 39*, 1 (1977).

[24] Ding, Chris, Li, Tao, and Jordan, Michael I. Convex and semi-nonnegative matrix factorizations. *TPAMI 32*, 1 (2010), 45–55.

[25] Dunn, J. C. A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. *Journal of Cybernetics 3*, 3 (1973), 32–57.

[26] Elidan, G., McGraw, I., and Koller, D. Residual belief propagation: Informed scheduling for asynchronous message passing. In *UAI '06* (2006), pp. 165–173.

[27] Gabielkov, Maksym, Rao, Ashwin, and Legout, Arnaud. Studying social networks at scale: Macroscopic anatomy of the twitter social graph. In *SIGMETRICS '14* (2014), pp. 277–288.

[28] Gemulla, Rainer, Nijkamp, Erik, Haas, Peter J., and Sismanis, Yannis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD '11* (2011), pp. 69–77.

[29] Gonzalez, Joseph E., Low, Yucheng, and Guestrin, Carlos. Residual splash for optimally parallelizing belief propagation. In *AISTATS '09* (2009), pp. 177–184.

[30] Gonzalez, Joseph E., Low, Yucheng, Guestrin, Carlos, and O'Hallaron, David. Distributed parallel inference on large factor graphs. In *UAI '09* (2009), pp. 203–212.

[31] Gonzalez, Joseph E., Xin, Reynold S., Dave, Ankur, Crankshaw, Daniel, Franklin, Michael J., and Stoica, Ion. Graphx: Graph processing in a distributed dataflow framework. In *OSDI'14* (2014), pp. 599–613.

[32] Griffiths, Thomas L., and Steyvers, Mark. Finding scientific topics. *PNAS 101*, suppl. 1 (2004), 5228–5235.

[33] Guan, Naiyang, Tao, Dacheng, Luo, Zhigang, and Yuan, Bo. Online nonnegative matrix factorization with robust stochastic approximation. *IEEE Trans. Neural Netw. Learning Syst. 23*, 7 (2012), 1087–1099.

[34] Guan, Ziyu, Wu, Jian, Zhang, Qing, Singh, Ambuj, and Yan, Xifeng. Assessing and ranking structural correlations in graphs. In *SIGMOD '11* (2011), pp. 937–948.

[35] Ha, Jiwoon, Kwon, Soon-Hyoung, Kim, Sang-Wook, Faloutsos, Christos, and Park, Sunju. Top-N recommendation through belief propagation. In *CIKM '12* (2012), pp. 2343–2346.

[36] Hoyer, Patrik O. Non-negative matrix factorization with sparseness constraints. *J. Mach. Learn. Res. 5* (Dec. 2004), 1457–1469.

[37] Jeh, Glen, and Widom, Jennifer. Scaling personalized web search. In *WWW '03* (2003), pp. 271–279.

[38] Karypis, G., and Kumar, V. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing 48* (1998), 96–129.

[39] Kersting, Kristian, Ahmadi, Babak, and Natarajan, Sriraam. Counting belief propagation. In *UAI '09* (2009), pp. 277–284.

[40] Kim, Hyunsoo, and Park, Haesun. Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method. *SIAM Journal on Matrix Analysis and Applications 30*, 2 (2008), 713–730.

[41] Kim, Jingu, and Park, Haesun. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM J. Sci. Comput. 33*, 6 (Nov. 2011), 3261–3281.

[42] Kleinberg, Jon M. Authoritative sources in a hyperlinked environment. *J. ACM 46* (September 1999), 604–632.

[43] Koller, D., and Friedman, N. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[44] Koren, Yehuda, Bell, Robert, and Volinsky, Chris. Matrix factorization techniques for recommender systems. *Computer 42*, 8 (Aug. 2009), 30–37.

[45] Kowalczyk, Wojtek, and Vlassis, Nikos. Newscast EM. In *NIPS '04* (2005), pp. 713–720.

[46] Langville, Amy N., and Meyer, Carl D. Updating PageRank with iterative aggregation. In *WWW '04* (2004), pp. 392–393.

[47] Langville, Amy N., and Meyer, Carl D. Updating markov chains with an eye on Google's PageRank. *SIAM J. Matrix Anal. Appl. 27*, 4 (2006), 968–987.

[48] Lauritzen, S. L., and Spiegelhalter, D. J. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological) 50*, 2 (1988), pp. 157–224.

[49] Lee, Daniel D., and Seung, H. Sebastian. Learning the parts of objects by non-negative matrix factorization. *Nature 401* (1999), 788–791.

[50] Lee, Daniel D., and Seung, H. Sebastian. Algorithms for non-negative matrix factorization. In *NIPS* (2000), MIT Press, pp. 556–562.

[51] Lempel, R., and Moran, S. Salsa: The stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst. 19*, 2 (Apr. 2001), 131–160.

[52] Leskovec, Jure, and Krevl, Andrej. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[53] Li, Boduo, Tata, Sandeep, and Sismanis, Yannis. Sparkler: Supporting large-scale matrix factorization. In *EDBT '13* (2013), pp. 625–636.

[54] Li, Li-Xin, Wu, Lin, Zhang, Hui-Sheng, and Wu, Fang-Xiang. A fast algorithm for nonnegative matrix factorization and its convergence. *IEEE Trans. Neural Netw. Learning Syst. 25*, 10 (2014), 1855–1863.

[55] Lin, Chih-Jen. Projected gradient methods for nonnegative matrix factorization. *Neural Comput. 19*, 10 (Oct. 2007), 2756–2779.

[56] Liu, Benben, Yu, ChiWai, Wang, D. Z., Cheung, R. C. C., and Yan, Hong. Design exploration of geometric biclustering for microarray data analysis in data mining. *IEEE Transactions on Parallel and Distributed Systems 25*, 10 (Oct 2014), 2540–2550.

[57] Liu, Chao, Yang, Hung-chih, Fan, Jinliang, He, Li-Wei, and Wang, Yi-Min. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW '10* (2010), pp. 681–690.

[58] Lloyd, S. Least squares quantization in PCM. *Information Theory, IEEE Transactions on 28*, 2 (mar 1982), 129 – 137.

[59] Lofgren, Peter A., Banerjee, Siddhartha, Goel, Ashish, and Seshadhri, C. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *KDD '14* (2014), pp. 1436–1445.

[60] Logothetis, Dionysios, Olston, Christopher, Reed, Benjamin, Webb, Kevin C., and Yocum, Ken. Stateful bulk processing for incremental analytics. In *SoCC '10* (2010), pp. 51–62.

[61] Low, Yucheng, Bickson, Danny, Gonzalez, Joseph, Guestrin, Carlos, Kyrola, Aapo, and Hellerstein, Joseph M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[62] Low, Yucheng, Gonzalez, Joseph, Kyrola, Aapo, Bickson, Danny, Guestrin, Carlos, and Hellerstein, Joseph M. GraphLab: A new parallel framework for machine learning. In *UAI '10* (2010).

[63] Macqueen, J. B. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability* (1967), pp. 281–297.

[64] Malewicz, Grzegorz, Austern, Matthew H., Bik, Aart J.C, Dehnert, James C., Horn, Ilan, Leiser, Naty, and Czajkowski, Grzegorz. Pregel: A system for large-scale graph processing. In *SIGMOD '10* (2010), pp. 135–146.

[65] McEliece, R.J., MacKay, D. J C, and Cheng, Jung-Fu. Turbo decoding as an instance of pearl's "belief propagation" algorithm. *Selected Areas in Communications, IEEE Journal on 16*, 2 (Sept. 2006), 140–152.

[66] Meng, Xiao L., and Rubin, Donald B. Maximum Likelihood Estimation via the ECM Algorithm: A General Framework. *Biometrika 80*, 2 (1993), 267–278.

[67] Menon, Aditya Krishna, and Elkan, Charles. Link prediction via matrix factorization. In *ECML/PKDD '11* (2011), pp. 437–452.

[68] Mooij, Joris, and Kappen, Hilbert. Sufficient conditions for convergence of loopy belief propagation. In *UAI '05* (2005), pp. 396–403.

[69] Murray, Derek G., McSherry, Frank, Isaacs, Rebecca, Isard, Michael, Barham, Paul, and Abadi, Martín. Naiad: A timely dataflow system. In *SOSP '13* (2013), pp. 439–455.

[70] Neal, Radford, and Hinton, Geoffrey E. A view of the EM algorithm that justifies incremental, sparse, and other variants. In *Learning in Graphical Models* (1998), pp. 355–368.

[71] Pauca, V. Paul, Shahnaz, Farial, Berry, Michael W., and Plemmons, Robert J. Text mining using non-negative matrix factorizations. In *SDM* (2004), pp. 452–456.

[72] Pearl, Judea. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., 1988.

[73] Popa, Lucian, Budiu, Mihai, Yu, Yuan, and Isard, Michael. Dryadinc: Reusing work in large-scale computations. In *HotCloud'09* (2009).

[74] Revelle, Matt, Domeniconi, Carlotta, Sweeney, Mack, and Johri, Aditya. Finding community topics and membership in graphs. In *ECML/PKDD' 15* (2015), pp. 625–640.

[75] Richardson, Matthew, and Domingos, Pedro. Markov logic networks. *Mach. Learn. 62*, 1-2 (Feb. 2006), 107–136.

[76] Robert, Christian P., and Casella, George. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag New York, Inc., 2005.

[77] Sarkar, Purnamrita, and Moore, Andrew W. Fast nearest-neighbor search in disk-resident graphs. In *KDD '10* (2010), pp. 513–522.

[78] Schelter, Sebastian, Boden, Christoph, Schenck, Martin, Alexandrov, Alexander, and Markl, Volker. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *RecSys '13* (2013), pp. 281–284.

[79] Singh, Ajit P., and Gordon, Geoffrey J. A unified view of matrix factorization models. In *ECML/PKDD '08* (2008), pp. 358–373.

[80] Song, Han Hee, Cho, Tae Won, Dave, Vacha, Zhang, Yin, and Qiu, Lili. Scalable proximity estimation and link prediction in online social networks. In *IMC '09* (2009), pp. 322–335.

[81] Song, Han Hee, Savas, Berkant, Cho, Tae Won, Dave, Vacha, Lu, Zhengdong, Dhillon, Inderjit S., Zhang, Yin, and Qiu, Lili. Clustered embedding of massive social networks. In *SIGMETRICS '12* (2012), pp. 331–342.

[82] Song, Le, Gretton, Arthur, Bickson, Danny, Low, Yucheng, and Guestrin, Carlos. Kernel belief propagation. In *AISTATS '11* (2011), pp. 333–341.

[83] Sutton, Charles, and McCallum, Andrew. Improved dynamic schedules for belief propagation. In *UAI '07* (2007), pp. 376–383.

[84] Teflioudi, Christina, Makari, Faraz, and Gemulla, Rainer. Distributed matrix completion. In *ICDM '12* (2012), pp. 655–664.

[85] Thiesson, Bo, Meek, Christopher, and Heckerman, David. Accelerating EM for large databases. *Mach. Learn. 45*, 3 (Dec. 2001), 279–299.

[86] Wang, Daisy Zhe, Michelakis, Eirinaios, Garofalakis, Minos N., and Hellerstein, Joseph M. BAYESSTORE: Managing large, uncertain data repositories with probabilistic graphical models. *PVLDB 1*, 1 (2008), 340–351.

[87] Wang, Fei, Li, Tao, Wang, Xin, Zhu, Shenghuo, and Ding, Chris. Community discovery using nonnegative matrix factorization. *Data Min. Knowl. Discov.* (May 2011).

[88] Wang, Guozhang, Xie, Wenlei, Demers, Alan, and Gehrke, Johannes. Asynchronous large-scale graph processing made easy. In *CIDR '13* (2013).

[89] Wang, Y., Xiang, Y., Zhang, J., Zhou, W., Wei, G., and Yang, L. T. Internet traffic classification using constrained clustering. *IEEE Transactions on Parallel and Distributed Systems 25*, 11 (Nov 2014), 2932–2943.

[90] Wolfe, Jason, Haghighi, Aria, and Klein, Dan. Fully distributed EM for very large datasets. In *ICML '08* (2008).

[91] Xiang, Chaocan, Yang, Panlong, Tian, Chang, Cai, Haibin, and Liu, Yunhao. Calibrate without calibrating: An iterative approach in participatory sensing network. *IEEE Transactions on Parallel and Distributed Systems 26*, 2 (Feb 2015), 351–361.

[92] Yedidia, Jonathan S., Freeman, William T., and Weiss, Yair. Exploring artificial intelligence in the new millennium. Morgan Kaufmann Publishers Inc., 2003, ch. Understanding Belief Propagation and Its Generalizations, pp. 239–269.

[93] Yin, Jiangtao, and Gao, Lixin. Scalable distributed belief propagation with prioritized block updates. In *CIKM '14* (2014), pp. 1209–1218.

[94] Yin, Jiangtao, and Gao, Lixin. Asynchronous distributed incremental computation on evolving graphs. In *ECML/PKDD '16* (2016).

[95] Yin, Jiangtao, Gao, Lixin, and Zhang, Zhongfei (Mark). Scalable nonnegative matrix factorization with block-wise updates. In *ECML/PKDD '14* (2014), pp. 337–352.

[96] Yin, Jiangtao, Zhang, Yanfeng, and Gao, Lixin. Accelerating expectation-maximization algorithms with frequent updates. In *CLUSTER '12* (2012), pp. 275–283.

[97] Yu, Hsiang-Fu, Hsieh, Cho-Jui, Si, Si, and Dhillon, Inderjit. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM '12* (2012).

[98] Yu, Kai, Dang, Xin, Bart, H., and Chen, Yixin. Robust model-based learning via spatial-EM algorithm. *Knowledge and Data Engineering, IEEE Transactions on 27*, 6 (June 2015), 1670–1682.

[99] Yu, Xiaofeng, Lam, Wai, and Chen, Bo. An integrated discriminative probabilistic approach to information extraction. In *CIKM '09* (2009), pp. 325–334.

[100] Yun, Hyokun, Yu, Hsiang-Fu, Hsieh, Cho-Jui, Vishwanathan, S. V. N., and Dhillon, Inderjit. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proc. VLDB Endow. 7*, 11 (July 2014), 975–986.

[101] Zaharia, Matei, Chowdhury, Mosharaf, Das, Tathagata, Dave, Ankur, Ma, Justin, McCauley, Murphy, Franklin, Michael J., Shenker, Scott, and Stoica, Ion. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI'12* (2012).

[102] Zaharia, Matei, Das, Tathagata, Li, Haoyuan, Hunter, Timothy, Shenker, Scott, and Stoica, Ion. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP '13* (2013), pp. 423–438.

[103] Zhai, Ke, Boyd-Graber, Jordan, Asadi, Nima, and Alkhouja, Mohamad L. Mr. LDA: A flexible large scale topic modeling package using variational inference in mapreduce. In *WWW '12* (2012), pp. 879–888.

[104] Zhang, Chao, Jiang, Shan, Chen, Yucheng, Sun, Yidan, and Han, Jiawei. Fast inbound top-k query for random walk with restart. In *ECML/PKDD' 15* (2015), pp. 608–624.

[105] Zhang, Yanfeng, Gao, Qixin, Gao, Lixin, and Wang, Cuirong. iMapReduce: A distributed computing framework for iterative computation. In *DataCloud '11* (2011), pp. 1112 –1121.

[106] Zhang, Yanfeng, Gao, Qixin, Gao, Lixin, and Wang, Cuirong. PrIter: A distributed framework for prioritized iterative computations. In *SoCC '11* (2011), pp. 13:1–13:14.

[107] Zhang, Yanfeng, Gao, Qixin, Gao, Lixin, and Wang, Cuirong. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *ScienceCloud '12* (2012), pp. 13–22.

[108] Zhao, H., Liu, X., and Li, X. DLBEM: Dynamic load balancing using expectation-maximization. In *IPDPS '08* (2008), pp. 1–7.

[109] Zhao, X., Li, X., Zhang, Z., Shen, C., Zhuang, Y., Gao, L., and Li, X. Scalable linear visual feature learning via online parallel nonnegative matrix factorization. *IEEE Trans. Neural Netw. Learning Syst. PP*, 99 (2015), 1–15.

[110] Zhou, Yunhong, Wilkinson, Dennis, Schreiber, Robert, and Pan, Rong. Large-scale parallel collaborative filtering for the netflix prize. In *AAIM '08* (2008), pp. 337–348.

[111] Zoidi, Olga, Tefas, Anastasios, and Pitas, Ioannis. Multiplicative update rules for concurrent nonnegative matrix factorization and maximum margin classification. *IEEE Trans. Neural Netw. Learning Syst. 24*, 3 (2013), 422–434.

[112] Zou, Jun, and Fekri, Faramarz. A belief propagation approach for detecting shilling attacks in collaborative filtering. In *CIKM '13* (2013), pp. 1837–1840.