Social Network Analysis and Mining manuscript No.
(will be inserted by the editor)

Parallel Collective Factorization for Modeling Large Heterogeneous Networks

Ryan A. Rossi · Rong Zhou

Received: June 13, 2015 / Accepted: June 15, 2016

Abstract Relational learning methods for heterogeneous network data are becoming increasingly important for many real-world applications. However, existing relational learning approaches are sequential, inefficient, unable to scale to large heterogeneous networks, as well as many other limitations related to convergence, parameter tuning, etc. In this paper, we propose Parallel Collective Matrix Factorization (PCMF) that serves as a fast and flexible framework for joint modeling of a variety of heterogeneous network data. The PCMF learning algorithm solves for a single parameter given the others, leading to a parallel scheme that is fast, flexible, and general for a variety of relational learning tasks and heterogeneous data types. The proposed approach is carefully designed to be (a) efficient for large heterogeneous networks (linear in the total number of observations from the set of input matrices), (b) flexible as many components are interchangeable and easily adaptable, and (c) effective for a variety of applications as well as for different types of data. The experiments demonstrate the scalability, flexibility, and effectiveness of PCMF for a variety of relational modeling tasks. In particular, PCMF outperforms a recent state-of-the-art approach in runtime, scalability, and prediction quality. Finally, we also investigate variants of PCMF for serving predictions in a real-time streaming fashion.

Keywords Recommender systems · missing value estimation · matrix completion · relational learning · low rank approximation · parallelization · scalable graph models · matrix factorization · collective factorization · coupled matrix-tensor factorization · cyclic coordinate descent · heterogeneous networks · prediction · social networks · link prediction · role discovery · network analysis

Ryan A. Rossi and Rong Zhou
Palo Alto Research Center (PARC, a Xerox Company)
3333 Coyote Hill Rd, Palo Alto, CA 94304 USA
E-mail: {rrossi, rzhou}@parc.edu

1 Introduction

In many real-world settings, there are often many large networks of different types that are important to model. Modeling these networks in a joint fashion is often critical for quality and accuracy. This work aims to develop a fast, flexible and scalable approach for jointly factorizing these heterogeneous data sources into a set of low-rank factors that approximate the original data. Besides flexibility and scalability, our approach is general and serves as a basis for use in a variety of predictive and descriptive modeling tasks.

Low-rank matrix factorization is a key component of machine learning and lies at the heart of many regression, factor analysis, dimensionality reduction, and clustering algorithms (with applications in signal and image processing, recommender systems, bioinformatics, among others). However, the majority of work has focused on techniques for factorizing a single matrix and thus are very limited in their ability to exploit the numerous heterogeneous data sources available in most real-world applications. Recently matrix factorization methods for recommendation have gained significant attention. More specifically, there has recently been numerous efforts on scaling up traditional matrix factorization methods for use on large data such as a single user-item rating matrix [41, 23, 39]. Matrix factorization based recommendation systems learn a model to predict the preferences of users [14]. Due to the significant practical importance of recommendation (*e.g.*, suggesting movies, products, friends), there have been a number of recent proposals to

speedup matrix factorization and the optimization methods that lie at the heart of these techniques. In particular, the main optimization schemes for factorizing a single user-item for recommendation have recently been parallelized including stochastic gradient descent (SGD) [41, 23,35,11,24], cyclic coordinate descent (CCD) [39], and alternating least squares (ALS) is easily parallelized via rows [40]. Importantly, Yu *et al.* [39] introduced a parallel CCD approach called CCD++ that factorized a single user-item matrix for the task of recommendation and demonstrated a significant improvement over the other state-of-the-art parallel schemes.

However, these approaches are limited to only a single matrix, while in reality there are often multiple heterogeneous data sources available that are of importance [32,34,13]. Furthermore, not only is multiple data sources common in practice, but is useful and may significantly improve model accuracy when leveraged. For instance, in the case of recommendation, it is known that personal preferences such as which product to buy (*e.g.*, the brand and type of laptop that a user will ultimately purchase) are directly influenced by our social connections and close contacts [30]. This assumption has been confirmed and used in many domains including sociology (homophily) [21], web/recommendation [30], and relational learning (autocorrelation) [27]. In contrast, this article proposes a fast parallel method for factorizing and ultimately fusing multiple network data sources.

In the context of recommendation, some work has combined multiple data sources in order to provide better recommendations to users [33,37,7]. More recently, Hao *et al.* [19] developed a social recommender system (SoRec) based on probabilistic matrix factorization (PMF) [28] using SGD to combine the social network with the user item matrix. Other work used the social network as a form of regularization [37,33,36, 12]. Instead, we focus on *parallel collective factorization* methods that are efficient, flexible, and general for use in a variety of predictive and descriptive modeling tasks in large heterogeneous data.

While recent work has scaled up traditional matrix factorization techniques for recommendation, these approaches use only a single matrix despite that multiple data sources are common in practice and improve performance when leveraged. Conversely, recent methods for combining the social network and user-item matrix are inefficient, sequential, do not scale, and have many other issues with convergence, etc. Instead, this paper proposes a general framework for **P**arallel **C**ollective **M**atrix **F**actorization (PCMF) that simultaneously factorizes multiple heterogeneous data sources. We also point out that PCMF naturally handles sparse and dense matrices, heterogeneous and homogeneous networks consisting of multiple node and edge types as well as dense feature matrices. Thus, PCMF is extremely general for including additional information in the factorization such as textual data represented as a word-document matrix for modeling topics or for computing feature-based roles from a set of graphs and feature matrices for each node type, among many other possibilities.

This paper proposes a general learning algorithm for PCMF that analytically solves for one parameter at a time, given the others. The learning algorithm of PCMF is generalized for jointly modeling an arbitrary number of matrices (network data or feature matrices). In addition, we propose a fast parallel learning algorithm that enables PCMF to model extremely large heterogeneous network data. Furthermore, the parallel learning algorithm of PCMF is extremely *flexible* as many components are interchangeable and can be customized for specific relational learning tasks. One important advantage of PCMF lies in the flexibility of choosing when and how parameters are selected and optimized. Our approach also has other benefits such as its ability to handle data that is extremely sparse. Despite the difficulty of this problem, PCMF leverages additional information such as the social network or other known information to improve prediction quality.

The experiments demonstrate the effectiveness of PCMF for jointly modeling heterogeneous network data. In particular, PCMF as well as our single matrix variant PCMF-BASIC outperforms the recent state-of-the-art in terms of the following: (1) runtime, (2) scalability and parallel speedup, and (3) prediction quality. Furthermore, even the most basic PCMF variant called PCMF-BASIC (the single matrix variant) is significantly faster and more scalable than CCD++—the recent state-of-the-art parallel approach for recommender systems. While this approach is for recommendation only and does not handle more than a single matrix, it is nevertheless important as it helps us understand the importance of the PCMF learning algorithm, and the key factors that lead to PCMF-BASIC's significant improvement, both in terms of runtime as well as quality of the model learned for various relational prediction tasks.

The main contributions of this work are as follows:

- *Novel algorithm.* We propose PCMF—a fast, parallel relational model that jointly models a variety of heterogeneous network data sources. At the heart of PCMF lies a fast parallel optimization scheme that updates a single parameter at a time, given all others.
- *Effectiveness*: The experiments demonstrate the scalability, flexibility, and effectiveness of PCMF for a variety of predictive modeling tasks. In addition, our

collective factorization framework is especially useful for sparse data with a limited number of observations as PCMF naturally incorporates the additional data sources into the factorization and leverages them to improve the quality of inference. We also demonstrate the effectiveness of PCMF for serving recommendations in real-time streaming environment.

- *Non-parametric and data-driven.* For descriptive modeling tasks, we propose a fast relaxation method for automatic and effective search over the space of models, selecting the appropriate model. The approach is data-driven and completely automatic as it doesn't require any user-defined parameters, making it suitable for many real-world applications.
- *Scalability*: The runtime is linear with respect to the number of nonzero elements in all matrices. Our parallel method is also shown to scale extremely well for a wide variety of data with different underlying characteristics. Notably, even PCMF's single-matrix variant PCMF-BASIC scales better than CCD++–the recent state-of-the-art parallel approach for recommender systems.
- *Generality*: We demonstrate the generality of PCMF by applying it for a variety of data types and recommendation tasks as well as using it for serving recommendations in a streaming fashion.

1.1 Scope and Organization of this Article

This article primarily focuses on designing efficient and flexible methods for parallel collective factorization with a special emphasis on fast coordinate descent methods. Prior work has not exploited fast coordinate descent for these types of problems, nor do they propose scalable parallel schemes. Furthermore this article also focuses on the generality of our approach for use on a variety of *applications* (*e.g.*, heterogeneous link prediction, topic modeling) and *input data* (*e.g.*, node/edge attributes, similarity matrices, sparse and dense single/multi-typed graphs). In contrast, recent work proposes specialized techniques specifically for the recommendation problem and are severely limited as a result of numerous assumptions used in learning as well as input data. As a result, the class of PCMF models are quite general for a number of predictive and descriptive modeling tasks as demonstrated later in Section 4.

We do not focus on stochastic gradient descent (SGD) or other similar optimization schemes since they have been extensively investigated and the problems and issues that arise from these methods are well known. Moreover, we also do not focus on sequential methods for collective factorization. As an aside, many of the ideas discussed in this article may be helpful in proposing collective optimization schemes using SGD. Nevertheless, SGD based methods are outside the scope of this work. Instead, this article investigates fast parallel collective coordinate descent based methods for jointly factorizing additional contextual and side information. We systematically investigate a large number of such parallel coordinate descent based variants for parallel collective factorization. Furthermore, the variants are compared empirically on a number of predictive and descriptive modeling tasks and across a variety of input data and characteristics. This article gives intuition for when such variants should be used and the impact of parameters, dimensionality of the model, among many others. Since PCMF was designed for use in real-time, we propose a data-driven non-parametric model that is fully automatic and does not require user input. We also propose an extremely fast relaxation method that essentially searches the space of models using rough approximations. Results are shown to be strikingly accurate with significant speedups compared to the general approach from the PCMF framework.

Many other optimizations are detailed throughout the manuscript including memory and thread layouts, careful memory access patterns to fully utilize available cache lines, the importance of selecting "good" initial matrices, ordering strategies, and other techniques to minimize dynamic load balancing issues, among many others. Although distributed memory architectures are outside the scope of this work, we nevertheless discuss a relatively simple distributed parallelization scheme that avoids the problems and issues that arise in many of the recent approaches for the simple one-matrix factorization problem. A key observation allows us to exploit the dependencies of the latent feature matrices to ensure both the CPU and network are fully utilized while also avoiding the synchronization issues that are arise in many other approaches (*e.g.*, workers remain idle waiting for the slowest worker to finish, known as the "curse of the last reducer").

The remainder of this article is organized as follows. Section 2 gives preliminaries and provides a running example. Section 3 introduces PCMF — a fast parallel approach for factorizing an arbitrary number of matrices and/or attributes (PCMF). This approach is then evaluated in Section 4. Section 5 concludes with future directions.

2 Background

Let $\mathbf{A} = \left[a_{ij}\right] \in \mathbb{R}^{m \times n}$ be a matrix with m rows (*e.g.*, users) and n columns (*e.g.*, items). In the context of

recommendation, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is typically a weighted matrix (*e.g.*, ratings for items, movies, etc.) with $m \gg n$ and represents a user-item bipartite graph where A_{ij} is the rating given by user i for item j. For other applications the matrix $\mathbf{A}$ may also be unweighted and/or symmetric where $A_{ij} = 1$ may indicate that a user i is a friend of user j (*e.g.*, social networks). Nevertheless, the algorithms developed in this manuscript are suitable for graphs that are weighted/unweighted, homogeneous/heterogeneous, and for sparse/dense graphs. We also denote $\Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ as the observed entries of $\mathbf{A}$ (nonzeros) where $(i, j) \in \Omega$ indicates that user i gave item j a rating of A_{ij}. Furthermore, let $\Omega_i := \{j : (i, j) \in \Omega\}$ be the set of items rated by the ith user whereas $\bar{\Omega}_j := \{i : (i, j) \in \Omega\}$ denotes the set of users who rated the jth item. Hence, $|\Omega_i|$ and $|\bar{\Omega}_j|$ are the number of nonzeros in row i and column j, respectively.

The goal of the traditional (single) matrix completion problem is to approximate the incomplete matrix $\mathbf{A}$ by $\mathbf{U}\mathbf{V}^\top$. More formally, given $\mathbf{A} \in \mathbb{R}^{m \times n}$, find $\mathbf{U} \in \mathbb{R}^{m \times d}$ and $\mathbf{V} \in \mathbb{R}^{n \times d}$ where $d \ll \min(m, n)$ such that $\mathbf{A} \approx \mathbf{U}\mathbf{V}^\top$. Intuitively, $\mathbf{U} \in \mathbb{R}^{m \times d}$ represents the low dimensional latent *user feature space* whereas $\mathbf{V} \in \mathbb{R}^{n \times d}$ represents the *item feature space*, respectively. Each row $\mathbf{u}_i^\top \in \mathbb{R}^d$ of $\mathbf{U}$ can be interpreted as a low dimensional rank-d embedding of the ith row in $\mathbf{A}$ (i.e., user if $\mathbf{A}$ is a user-item ratings matrix). Alternatively, each row $\mathbf{v}_j^\top \in \mathbb{R}^d$ of $\mathbf{V}$ represents a d-dimensional embedding of the jth column in $\mathbf{A}$ (i.e., item) using the same low rank-d dimensional space. Also, $\mathbf{u}_k \in \mathbb{R}^m$ is the kth column of $\mathbf{U}$ and similarly $\mathbf{v}_k \in \mathbb{R}^n$ is the kth column of $\mathbf{V}$. Further, let U_{ik} be a scalar (k^{th} element of $\mathbf{u}_i^\top$ or the ith element of $\mathbf{u}_k$). Similarly, let u_{ik} and v_{jk} for $1 < k < d$ denote the kth coordinate of the column vectors $\mathbf{u}_i$ and $\mathbf{v}_j$ (and thus interchangeable with U_{ik} and V_{jk}). For clarity, we also use $\mathbf{U}_{:k}$ to denote the kth column of $\mathbf{U}$ (and $\mathbf{U}_{i:}$ for the ith row of $\mathbf{U}$). Similar notation is used for $\mathbf{V}$ and $\mathbf{Z}$.

2.1 Problem Formulation

To measure the quality of our model, we use a nonzero squared loss: $(A_{ij} - \mathbf{u}_i^\top \mathbf{v}_j)^2$. Though, our optimization method may use any arbitrary separable loss (as an objective function). Regularization terms are also introduced in order to prevent overfitting and predict well on unknown ratings. Let us note that PCMF is easily capable of handling a number of regularizers. In this work, we use square-norm regularization. For the traditional single matrix factorization problem (i.e., a special case of PCMF), we have the following objective function:

$$\min_{\substack{\mathbf{U} \in \mathbb{R}^{m \times d} \\ \mathbf{V} \in \mathbb{R}^{n \times d}}} \sum_{(i,j) \in \Omega} \Big\{ (A_{ij} - \mathbf{u}_i^\top \mathbf{v}_j)^2 + \lambda_u \|\mathbf{u}_i\|^2 + \lambda_v \|\mathbf{v}_j\|^2 \Big\}$$

where the *regularization parameters* $\lambda_u > 0$ and $\lambda_v > 0$ are scalars that trade off the loss function with the regularizer. The above problem formulation has a variety of limitations in practice. For instance, the above problem uses only a *single* data source in the factorization and has prediction quality problems for rows (users) with very few or no observations (ratings). To help solve the data sparsity issues that arise in practice, we take advantage of correlations between different data sets and simultaneously factorize multiple matrices. Given two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{m \times m}$, we formulate the collective factorization problem as:

$$\min_{\mathbf{U},\mathbf{V},\mathbf{Z}} \Big\{ \sum_{(i,j) \in \Omega_{\mathbf{A}}} (A_{ij} - \mathbf{u}_i^\top \mathbf{v}_j)^2 + \lambda_u \|\mathbf{U}\|_F + \lambda_v \|\mathbf{V}\|_F + \sum_{(i,j) \in \Omega_{\mathbf{B}}} (B_{ij} - \mathbf{u}_i^\top \mathbf{z}_j)^2 + \alpha \|\mathbf{Z}\|_F \Big\} \quad (1)$$

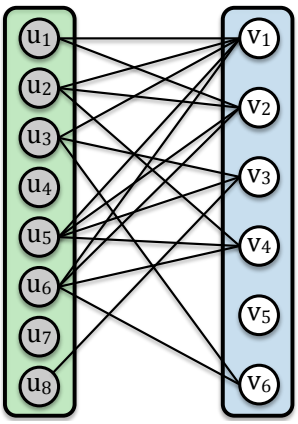
where $\mathbf{U} \in \mathbb{R}^{m \times d}$, $\mathbf{V} \in \mathbb{R}^{n \times d}$, and $\mathbf{Z} \in \mathbb{R}^{m \times d}$ are low-rank factor matrices.

2.2 Motivating Example

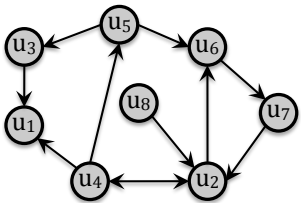
For illustration purposes, we consider PCMF with two types of input data shown in Fig. 1. More specifically, we are given a weighted bipartite graph (bigraph) with two types of nodes representing 8 users and 6 items along with 17 weighted edges representing the fact that a user rated an item. We also have a *directed* social network with 8 users and 12 edges representing friendships[1]. Furthermore, the social network has only a single node and edge type (i.e., homogeneous) whereas the user-item bigraph has two types of nodes representing users and items. Apart from the node types, this example also includes two edge types, i.e., ratings in $\mathbf{A}$ and friendships in $\mathbf{B}$. Indeed, PCMF naturally generalizes for many other types of data and scenarios not covered by this simple example (*e.g.*, dense matrices such as feature or similarity matrices, etc).

Following the intuition that a user is more likely to be influenced by friends than random users, we jointly factorize the user-item matrix and social network (from Fig 1) using $\mathbf{U}\mathbf{V}^\top$ and $\mathbf{U}\mathbf{Z}^\top$ where the low-dimensional matrix $\mathbf{U} \in \mathbb{R}^{m \times d}$ represents the `shared` user latent feature space, and the low-dimensional matrix $\mathbf{V} \in \mathbb{R}^{n \times d}$ represents the *item latent feature space*, and similarly,

[1] Note that undirected homogeneous networks (symmetric matrices) are a special case of our framework



(a) User-item bipartite graph



(b) Social graph

	$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$	$\mathbf{v}_4$	$\mathbf{v}_5$	$\mathbf{v}_6$
$\mathbf{u}_1$	5	3				
$\mathbf{u}_2$	4	2		5		
$\mathbf{u}_3$	4		2			4
$\mathbf{u}_4$						
$\mathbf{u}_5$	4	1	2	4		
$\mathbf{u}_6$	5	3		4		5
$\mathbf{u}_7$						
$\mathbf{u}_8$			3			

(c) $\mathbf{A} \in \mathbb{R}^{m \times n}$

	$\mathbf{u}_1$	$\mathbf{u}_2$	$\mathbf{u}_3$	$\mathbf{u}_4$	$\mathbf{u}_5$	$\mathbf{u}_6$	$\mathbf{u}_7$	$\mathbf{u}_8$
$\mathbf{u}_1$								
$\mathbf{u}_2$				1		1		
$\mathbf{u}_3$	1							
$\mathbf{u}_4$	1	1			1			
$\mathbf{u}_5$			1			1		
$\mathbf{u}_6$							1	
$\mathbf{u}_7$		1						
$\mathbf{u}_8$		1				1		

(d) $\mathbf{B} \in \mathbb{R}^{m \times m}$

Fig. 1 An example of the input data leveraged by PCMF. In this example, we show a bipartite user-item graph (a) and its weighted adjacency matrix (c) as well as the directed social graph (b) represented by the adjacency matrix in (d).

$\mathbf{Z} \in \mathbb{R}^{m \times d}$ is the *social latent feature space*. Therefore, $\mathbf{A} \approx \mathbf{U}\mathbf{V}^\top$ and $\mathbf{B} \approx \mathbf{U}\mathbf{Z}^\top$. In this example, we set the rank $d = 2$, and use $\alpha = 2$ to control the influence of $\mathbf{B}$ on the factorization. For the regularization parameters, we set $\lambda = 0.1$. The low-dimensional matrices representing the latent feature space of the users $\mathbf{U}$, items $\mathbf{V}$, and social interactions $\mathbf{Z}$ are shown in Fig. 2 and colored by the coordinates magnitude. A key advantage of PCMF is its flexibility for collectively factorizing a wide variety of data (textual, features, similarity matrices, images, etc) that goes well beyond the simple example used here. However, the data sources must be correlated *w.r.t.* the objective (or more generally the application), otherwise the additional data may lead to worse performance (due to noise/bias). This fundamental assumption is not unique to PCMF but is required for any such method.

To motivate PCMF we demonstrate its use for prediction using our example. For evaluation, a fraction of the known observations are withheld from learning to use as a testing set. In the example, the testing set consists of $\Omega^{\text{test}} = \{(u_4, v_1, 5), (u_6, v_3, 3), (u_8, v_1, 5)\}$. Using the low-dimensional representation from Fig 2, one can predict these known ratings and measure the difference (model error) using a metric that quantifies the "prediction quality." We predict the missing value of A_{ij} s.t. $(i, j) \notin \Omega_{\mathbf{A}}$ using $\mathbf{u}_i^\top \mathbf{v}_j$. For instance, the missing value for user u_4 and item v_1 (first test example) is given by $\mathbf{u}_4^\top \mathbf{v}_1 = \left[\begin{smallmatrix} 3.2 & 0.99 \end{smallmatrix}\right] \left[\begin{smallmatrix} 1.7 \\ -0.52 \end{smallmatrix}\right] = 4.92$. Therefore, we use PCMF to predict each test example in Ω^{test} and use Root Mean Squared Error (RMSE) to measure the error (prediction quality). Using only the user-item matrix (PCMF-BASIC) we obtain an error of 2.55 compared to 0.19 when our collective factorization approach is used.

More generally, one may use PCMF for the matrix completion problem which seeks to estimate all (or a subset of) the missing nonzero values in a given matrix. This problem is at the heart of many ranking tasks (*e.g.*, search engines must display only the top-k most relevant pages). As such, we investigate using two PCMF variants for this problem that differ in the information used to learn the low-dimensional feature spaces. Results are shown and discussed in Fig 3.

Another important problem where PCMF may be used directly is the social network link prediction problem. The goal here is to predict the *existence* of a future or missing link. For this problem, one can use PCMF to find a low-dimensional embedding of the large heterogeneous data, then we can predict the likelihood of a link between i and j using $\mathbf{u}_i^\top \mathbf{z}_j$ directly. All such links may be weighted using $\mathbf{U}\mathbf{Z}^\top$, which provides a ranking of the most likely potential links for each user.

3 Parallel Collective Factorization

We now derive a scalar coordinate descent optimization scheme that leverages fast and efficient element-wise updates via the columns. The basic idea of coordinate descent (CD) is to optimize one element at a time while fixing other elements by decomposing the objective in (1) into several one-variable subproblems. As pre-

2.6	-0.79
2.8	0.78
2.1	-0.28
3.2	0.99
2.3	0.66
2.8	-0.39
3.9	0.94
2.9	-0.37

(a) $\mathbf{U} \in \mathbb{R}^{m \times d}$

1.7	-0.52
0.89	-0.83
0.94	-0.24
1.5	0.63
0.42	0.62
1.8	-0.09

(b) $\mathbf{V} \in \mathbb{R}^{n \times d}$

0.01	0.07
0.27	0.11
0.27	-0.04
0.3	0.05
0.3	-0.02
0.22	0.13
0.27	0.15
0.3	0.04

(c) $\mathbf{Z} \in \mathbb{R}^{m \times d}$

Fig. 2 Low-dimensional latent features of the users, items, and social interactions. The initial input data from Fig. 1 is represented in a low-dimensional space using 2 dimensions and can be viewed as a rank-2 approximation such that $\mathbf{A} \approx \mathbf{U}\mathbf{V}^\top$ and $\mathbf{B} \approx \mathbf{U}\mathbf{Z}^\top$.

	$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$	$\mathbf{v}_4$	$\mathbf{v}_5$	$\mathbf{v}_6$
$\mathbf{u}_1$	5	3	2.6	3.4	0.6	4.7
$\mathbf{u}_2$	4	2	2.4	5	1.7	5.0
$\mathbf{u}_3$	4	2.1	2	3.0	0.7	4
$\mathbf{u}_4$	4.9	2.0	2.8	5.0	2.0	5.0
$\mathbf{u}_5$	4	1	2	4	1.4	4.1
$\mathbf{u}_6$	5	3	2.7	4	0.9	5
$\mathbf{u}_7$	5.0	2.7	3.4	5.0	2.2	5.0
$\mathbf{u}_8$	5.0	2.9	3	4.1	1.0	5.0

(a) PCMF

	$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$	$\mathbf{v}_4$	$\mathbf{v}_5$	$\mathbf{v}_6$
$\mathbf{u}_1$	5	3	2.6	3.5	0.6	4.7
$\mathbf{u}_2$	4	2	2.4	5	1.6	4.8
$\mathbf{u}_3$	4	2.1	2	3.2	0.8	4
$\mathbf{u}_4$	0.6	-0	0.3	1.5	0.8	1.0
$\mathbf{u}_5$	4	1	2	4	1.4	4.1
$\mathbf{u}_6$	5	3	2.8	4	1.0	5
$\mathbf{u}_7$	0.6	0.2	0.4	0.9	0.4	0.8
$\mathbf{u}_8$	5.0	2.9	3	4.3	1.0	5.0

(b) PCMF-BASIC

Fig. 3 Matrix completion using PCMF and PCMF-BASIC. Observed instances used in training were removed (see Fig 1). Using PCMF to leverage both the social network and user-item matrix gives the most reasonable predictions as seen in (a) when compared to PCMF-BASIC in (b) which uses only the user-item matrix. Strikingly, for users with no ratings (u_4 and u_7), we find that PCMF is able to predict reasonable ratings by leveraging the social information, whereas predictions made using PCMF-BASIC are clearly less meaningful. For items with no ratings (v_5), PCMF gives slightly more reasonable ratings than PCMF-BASIC. Though, if we had another matrix for items such as the latent topics from the textual information or another form of item similarity, then PCMF is expected to provide significantly better predictions for items with no ratings. For presentation purposes, predicted values were rounded and limited to be no larger than the max rating possible.

viously mentioned, we randomly initialize $\mathbf{U} \in \mathbb{R}^{m\times d}$, $\mathbf{V} \in \mathbb{R}^{n\times d}$, and $\mathbf{Z} \in \mathbb{R}^{m\times d}$. Furthermore we apply a sparsification technique which effectively sets a few entries to zero. As shown later, this speeds up convergence while also improving the model accuracy by generalizing better (in fewer iterations). For each inner iteration denoted t, we alternatively update $\mathbf{V}_{:k}$ and $\mathbf{Z}_{:k}$ and then use this to update $\mathbf{U}_{:k}$ and repeat. For clarity, we note that $\mathbf{V}_{:k} \in \mathbb{R}^n$, $\mathbf{Z}_{:k} \in \mathbb{R}^m$, and $\mathbf{U}_{:k} \in \mathbb{R}^m$ are used here to denote columns of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{Z}$, respectively. In particular, a single inner iteration updates the kth latent feature of $\mathbf{V}$, $\mathbf{Z}$ and $\mathbf{U}$ in the following order:

$$\overbrace{\underbrace{V_{1k}, V_{2k}, ..., V_{nk}}_{\mathbf{V}_{:k}}, \underbrace{Z_{1k}, Z_{2k}, ..., Z_{mk}}_{\mathbf{Z}_{:k}}, \underbrace{U_{1k}, U_{2k}, ..., U_{mk}}_{\mathbf{U}_{:k}}}^{\text{inner iteration}}$$

and thus, each outer-iteration updates the latent features in the following order:

$$\overbrace{\underbrace{\mathbf{V}_{:1}, \mathbf{Z}_{:1}, \mathbf{U}_{:1}}_{\text{inner iteration}}, ..., \underbrace{\mathbf{V}_{:k}, \mathbf{Z}_{:k}, \mathbf{U}_{:k}}_{\forall t=1,2,...}, ..., \underbrace{\mathbf{V}_{:d}, \mathbf{Z}_{:d}, \mathbf{U}_{:d}}_{\forall t=1,2,...}}^{\tau \text{ outer iteration}} \tag{2}$$

Note that this is in contrast to the common update ordering:

$$\mathbf{V}_{:1}, \mathbf{V}_{:2}, ..., \mathbf{V}_{:d}, \mathbf{Z}_{:1}, \mathbf{Z}_{:2}, ..., \mathbf{Z}_{:d}, \mathbf{U}_{:1}, \mathbf{U}_{:2}, ..., \mathbf{U}_{:d} \tag{3}$$

We have also experimented with different update strategies such as: $\mathbf{Z}_{:1}, \mathbf{V}_{:1}, \mathbf{U}_{:1}$,..., $\mathbf{Z}_{:d}, \mathbf{V}_{:d}, \mathbf{U}_{:d}$, among more sophisticated adaptive ordering variants.

We now give the update rules for a single element. We allow V_{jk} to change with v and fix all other variables to solve the following one-variable subproblem:

$$\min_{\mathrm{v}} J(\mathrm{v}) = \sum_{i\in\Omega_j^{\mathbf{A}}} \Big(A_{ij} - (\mathbf{U}_{i:}\mathbf{V}_{j:}^\top - U_{ik}V_{jk}) - U_{ik}\mathrm{v}\Big)^2 + \lambda_v \mathrm{v}^2$$

Since $J(\mathrm{v})$ is a univariate quadratic function, the unique solution is simply:

$$\mathrm{v}^\star = \frac{\sum_{i\in\Omega_j^{\mathbf{A}}} \big(A_{ij} - \mathbf{U}_{i:}\mathbf{V}_{j:}^\top + U_{ik}V_{jk}\big)U_{ik}}{\lambda + \sum_{i\in\Omega_j} U_{ik}U_{ik}} \tag{4}$$

While a naive implementation takes $\mathrm{O}(|\Omega^{\mathbf{A}}|d)$, we efficiently compute the update above in $O(|\Omega_j^{\mathbf{A}}|)$ linear time by carefully maintaining the residual matrix $\mathbf{E}^{\mathrm{a}}$:

$$E_{ij}^a \equiv A_{ij} - \mathbf{U}_{i:}\mathbf{V}_{j:}^\top, \quad \forall (i,j) \in \Omega^{\mathbf{A}} \tag{5}$$

Therefore, we can rewrite the above equation in terms of E_{ij}^a, the optimal $\mathrm{v}^\star$ is simply:

$$\mathrm{v}^\star = \frac{\sum_{i\in\Omega_j} \big(E_{ij}^a + U_{ik}V_{jk}\big)U_{ik}}{\lambda + \sum_{i\in\Omega_j} U_{ik}U_{ik}} \tag{6}$$

Now, we update V_{jk} and E_{ij}^a in $O(|\Omega_j|)$ time:

$$E_{ij}^a \leftarrow E_{ij}^a - (\mathrm{v}^\star - V_{jk})U_{ik}, \quad \forall_i \in \Omega_j \tag{7}$$

$$V_{jk} \leftarrow \mathrm{v}^\star \tag{8}$$

Similar update rules for solving a single subproblem in $\mathbf{Z}$ and $\mathbf{U}$ are straightforward to derive and may be computed efficiently using the same trick. Thus, the update rules for the one-variable subproblems V_{jk}, Z_{ik}, and U_{ik} are as follows:

$$\mathrm{v}^\star = \frac{\sum_{i\in\Omega_j^{\mathbf{A}}} \big(E_{ij}^a + U_{ik}V_{jk}\big)U_{ik}}{\lambda_v + \sum_{i\in\Omega_j^{\mathbf{A}}} U_{ik}U_{ik}} \tag{9}$$

$$\mathrm{z}^\star = \frac{\sum_{j\in\Omega_i^{\mathbf{B}}} \big(E_{ij}^b + U_{jk}Z_{ik}\big)U_{jk}}{\alpha + \sum_{j\in\Omega_i^{\mathbf{B}}} U_{jk}U_{jk}} \tag{10}$$

$$\mathrm{u}^\star = \frac{\sum_{j\in\Omega_i^{\mathbf{A}}} \big(E_{ij}^a + U_{ik}V_{jk}\big)V_{jk}}{\lambda_u + \sum_{j\in\Omega_i^{\mathbf{A}}} V_{jk}V_{jk}} + \frac{\sum_{j\in\Omega_i^{\mathbf{B}}} \big(E_{ij}^b + U_{ik}Z_{jk}\big)Z_{jk}}{\alpha + \sum_{j\in\Omega_i^{\mathbf{B}}} Z_{jk}Z_{jk}} \tag{11}$$

These updates rules may be used regardless of the order in which the one-variable subproblems are updated. Consequently, this gives rise to the PCMF class of variants that leverage adaptive strategies. Although we update $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ via asynchronous column-wise updates, the order in which the updates are performed may also impact performance and convergence properties. The factorization using a column-wise update sequence corresponds to the summation of outer-products:

$$\mathbf{A} \approx \mathbf{U}\mathbf{V}^\top = \sum_{k=1}^{d} \mathbf{U}_{:k}\mathbf{V}_{:k}^\top \tag{12}$$

$$\mathbf{B} \approx \mathbf{U}\mathbf{Z}^\top = \sum_{k=1}^{d} \mathbf{U}_{:k}\mathbf{Z}_{:k}^\top \tag{13}$$

where $\mathbf{V}_{:k}$, $\mathbf{Z}_{:k}$ and $\mathbf{U}_{:k}$ denote the kth column (or latent feature) of $\mathbf{V}$, $\mathbf{Z}$ and $\mathbf{U}$, respectively. Let $\mathrm{T}_{\text{outer}}$ and $\mathrm{T}_{\text{inner}}$ be the number of inner and outer iterations, respectively. Note $\mathrm{T}_{\text{outer}}$ is the number of times the kth latent feature is updated. $\mathrm{T}_{\text{inner}}$ is the number of times the kth latent feature is updated before updating the $k+1$ latent feature. Furthermore if the update order for each outer iteration is:

$$\mathbf{V}_{:1}, \mathbf{Z}_{:1}, \mathbf{U}_{:1}, \ldots, \mathbf{V}_{:k}, \mathbf{Z}_{:k}, \mathbf{U}_{:k}, \ldots, \mathbf{V}_{:d}, \mathbf{Z}_{:d}, \mathbf{U}_{:d},$$

then each of the d latent features are updated at each outer-iteration (via $\mathrm{T}_{\text{inner}}$ inner-iterations). Since elements in $\mathbf{v}_k$ (or $\mathbf{z}_k$, $\mathbf{z}_k$) can be computed independently, we focus on scalar approximations for each individual element. This gives rise to a scalar coordinate descent algorithm. For now, let us consider column-wise updates where the kth latent feature of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ is selected and updated in arbitrary order. See Alg 1 for a detailed description. Thus, during each inner iteration, we perform the following updates:

$$\mathbf{V}_{:k} \leftarrow \mathbf{v}^\star$$
$$\mathbf{Z}_{:k} \leftarrow \mathbf{z}^\star$$
$$\mathbf{U}_{:k} \leftarrow \mathbf{u}^\star$$

where $\mathbf{v}^\star$, $\mathbf{z}^\star$, and $\mathbf{u}^\star$ are obtained via the inner iterations. As an aside, PCMF performs rank-1 updates in-place so that we always use the current estimate. To obtain the updates above (i.e., $\mathbf{v}^\star$, $\mathbf{z}^\star$, $\mathbf{u}^\star$) one must solve the subproblem:

$$\min_{\mathbf{u},\mathbf{v},\mathbf{z}} \Bigg\{ \sum_{(i,j)\in\Omega^{\mathbf{A}}} \big(E_{ij}^a + U_{ik}^{(\tau)}V_{jk}^{(\tau)} - u_{ik}^\star v_{jk}^\star\big)^2 + \sum_{(i,j)\in\Omega^{\mathbf{B}}} \big(E_{ij}^b + U_{ik}^{(\tau)}Z_{jk}^{(\tau)} - u_i z_j\big)^2 + \lambda_u\|\mathbf{u}\|^2 + \lambda_v\|\mathbf{v}\|^2 + \alpha\|\mathbf{z}\|^2 \Bigg\} \tag{14}$$

where $\mathbf{E}^{\mathrm{a}} = \mathbf{A} - \mathbf{u}_k\mathbf{v}_k^\top$ and $\mathbf{E}^{\mathrm{b}} = \mathbf{B} - \mathbf{u}_k\mathbf{z}_k^\top$ are the initial sparse residual matrices for $\mathbf{A}$ and $\mathbf{B}$, respectively. The residual term for $\mathbf{A}$ is denoted as $\mathbf{A} - \mathbf{U}\mathbf{V}^\top = \mathbf{A}^{(k)} - \mathbf{u}_k\mathbf{v}_k^\top$ where the k-residual $\mathbf{A}^{(k)}$ is defined as:

$$\begin{aligned}\mathbf{A}^{(k)} &= \mathbf{A} - \sum_{f\neq k} \mathbf{u}_f\mathbf{v}_f^\top \\ &= \mathbf{A} - \mathbf{U}\mathbf{V}^\top + \mathbf{u}_k\mathbf{v}_k^\top, \quad \text{for } k = 1, 2, ..., d\end{aligned} \tag{15}$$

Similarly, the residual term for $\mathbf{B}$ is $\mathbf{B} - \mathbf{U}\mathbf{Z}^\top = \mathbf{B}^{(k)} - \mathbf{u}_k\mathbf{z}_k^\top$ where the k-residual $\mathbf{B}^{(k)}$ is defined as:

$$\begin{aligned}\mathbf{B}^{(k)} &= \mathbf{B} - \sum_{f\neq k} \mathbf{u}_f\mathbf{z}_f^\top \\ &= \mathbf{B} - \mathbf{U}\mathbf{Z}^\top + \mathbf{u}_k\mathbf{z}_k^\top, \quad \text{for } k = 1, 2, ..., d\end{aligned} \tag{16}$$

For a single residual entry, let us define $A_{ij}^{(k)}$ and $B_{ij}^{(k)}$ as:

$$A_{ij}^{(k)} = E_{ij}^a + U_{ik}V_{jk},\ \forall(i,j) \in \Omega^{\mathbf{A}} \quad \text{and} \tag{17}$$

$$B_{ij}^{(k)} = E_{ij}^b + U_{ik}Z_{jk},\ \forall(i,j) \in \Omega^{\mathbf{B}} \tag{18}$$

Equivalently, let $\mathbf{A}^{(k)} = \mathbf{E}^{\mathrm{a}} + \mathbf{u}_k\mathbf{v}_k^\top$ and $\mathbf{B}^{(k)} = \mathbf{E}^{\mathrm{b}} + \mathbf{u}_k\mathbf{z}_k^\top$. Now a straightforward rewriting of Eq. 14 gives the exact rank-1 matrix factorization problem from Eq. 1,

$$\min_{\mathbf{u},\mathbf{v},\mathbf{z}} \Bigg\{ \sum_{(i,j)\in\Omega^{\mathbf{A}}} \big(A_{ij}^{(k)} - u_{ik}v_{jk}\big)^2 + \lambda_u\|\mathbf{u}\|^2 + \lambda_v\|\mathbf{v}\|^2 + \sum_{(i,j)\in\Omega^{\mathbf{B}}} \big(B_{ij}^{(k)} - u_{ik}z_{jk}\big)^2 + \alpha\|\mathbf{z}\|^2 \Bigg\} \tag{19}$$

Using Eq. 19 gives an approximation by alternating between updating $\mathbf{v}$, $\mathbf{z}$, and $\mathbf{u}$ via column-wise updates. Note that when performing rank-1 updates, a single subproblem can be solved without any further residual maintenance. Thus, each one-variable subproblem may benefit from $\mathrm{T}_{\text{inner}}$ iterations. The inner iterations are fast to compute since we avoid updating the residual matrices while iteratively updating a given k latent factor via a number of inner iterations. As previously mentioned, updates are performed in-place and thus the most recent estimates are leveraged. This also has other important consequences as it reduces storage requirements, memory locality, etc. Furthermore, after the $\mathrm{T}_{\text{inner}}$ inner iterations, we update $\mathbf{E}^{\mathrm{a}}$ and $\mathbf{E}^{\mathrm{b}}$,

$$E_{ij}^a \leftarrow A_{ij}^{(k)} - \mathrm{u}_i^\star\mathrm{v}_j^\star,\ \forall(i,j) \in \Omega^{\mathbf{A}} \quad \text{and} \tag{20}$$

$$E_{ij}^b \leftarrow B_{ij}^{(k)} - \mathrm{u}_i^\star\mathrm{z}_j^\star,\ \forall(i,j) \in \Omega^{\mathbf{B}} \tag{21}$$

For convenience, we also define $\mathbf{E}^{\mathrm{a}} = \mathbf{A}^{(k)} - \mathbf{u}_k\mathbf{v}_k^\top$ and $\mathbf{E}^{\mathrm{b}} = \mathbf{B}^{(k)} - \mathbf{u}_k\mathbf{z}_k^\top$. Finally, the scalar coordinate descent algorithm updates each element independently using the

following update rules:

$$V_{jk} = \frac{\sum_{i \in \Omega_j^{\mathbf{A}}} A_{ij}^{(k)} U_{ik}}{\lambda_v + \sum_{i \in \Omega_j^{\mathbf{A}}} U_{ik} U_{ik}}, \quad j = 1, 2, ..., n \tag{22}$$

$$Z_{ik} = \frac{\sum_{j \in \Omega_i^{\mathbf{B}}} B_{ij}^{(k)} U_{jk}}{\alpha + \sum_{j \in \Omega_i^{\mathbf{B}}} U_{jk} U_{jk}}, \quad i = 1, 2, ..., m \tag{23}$$

$$U_{ik} = \frac{\sum_{j \in \Omega_i^{\mathbf{A}}} A_{ij}^{(k)} V_{jk}}{\lambda_u + \sum_{j \in \Omega_i^{\mathbf{A}}} V_{jk} V_{jk}} + \tag{24}$$

$$\frac{\sum_{j \in \Omega_i^{\mathbf{B}}} B_{ij}^{(k)} Z_{jk}}{\alpha + \sum_{j \in \Omega_i^{\mathbf{B}}} Z_{jk} Z_{jk}}, \quad i = 1, ..., m \tag{25}$$

Thus, the above update rules perform n element-wise updates on $\mathbf{v}_k$, then we perform m updates on $\mathbf{z}_k$, and finally m updates are performed for $\mathbf{u}_k$. Furthermore that approach does not define an element-wise update strategy (assumes a natural ordering given as input) nor does it allow for partial updates. For instance, one may update a single element in V_{jk}, then Z_{ik}, and U_{ik} and continue rotating until all elements have been updated once. Nevertheless, this work leverages such flexibility by defining both column-wise and element-wise update ordering strategies. We also relax the strict update pattern used in previous work where a single column is updated in its entirety before updating elements in any of the other columns. Intuitively, finer grained control of the updates may have caching or other benefits due to memory access patterns.

Solving the problem in (1) that has many free parameters (i.e., $(2m + n)d$) from sparse graph data typically leads to overfitting. Thus, we use the following weighted regularization term that penalizes large parameters.

$$\lambda_u \sum_{i=1}^{m} |\Omega_i| \cdot \|\mathbf{u}_i\|^2 + \lambda_v \sum_{j=1}^{n} |\Omega_j| \cdot \|\mathbf{v}_j\|^2 + \alpha \sum_{i=1}^{m} |\Omega_i^{\mathbf{B}}| \cdot \|\mathbf{z}_i\|^2$$

where $|\cdot|$ is the cardinality of a set (i.e., number of nonzeros in a row or col of $\mathbf{A}$ or $\mathbf{B}$) and $\|\cdot\|^2$ is the L_2 vector norm. where each regularization term is essentially weighted by the degree (in/out). We use a simple yet effective graph sparsifier for speeding up the general method. Outside of coordinate descent, it has been used for many applications [31,29,17]. This technique works quite well with cyclic coordinate descent as seen in Section 4. Due to known issues, we also use a custom random number generator, see Section 3.2 for details. On many problems we have observed faster convergence and consequently better predictions in fewer iterations.

3.1 Sparsity and Non-negativity Constraints

A number of important problems may warrant other constraints. We now discuss a few of the most important alternatives and derive them for the PCMF collective factorization framework.

3.1.1 Sparsity Constraints

An important variation of PCMF is the use of sparsity constraints on one or more factors. Sparsity constraints may be enforced using the ℓ_1-norm to obtain a sparse model. For imposing sparsity constraints, the previous objective function is replaced with the following penalized objective:

$$\min_{\substack{\mathbf{U} \in \mathbb{R}^{m \times d}, \\ \mathbf{V} \in \mathbb{R}^{n \times d}, \\ \mathbf{Z} \in \mathbb{R}^{m \times d}}} \sum_{(i,j) \in \Omega^{\mathbf{A}}} \left(\mathbf{A}_{ij} - \mathbf{u}_i^\top \mathbf{v}_j\right)^2 + \beta \sum_{(i,j) \in \Omega^{\mathbf{B}}} \left(\mathbf{B}_{ij} - \mathbf{u}_i^\top \mathbf{z}_j\right)^2 + \lambda_u \sum_{i=1}^{m} \|\mathbf{u}_i\|_1 + \lambda_v \sum_{j}$$

Now let us define

$$h = -2 \sum_{i \in \Omega_j^{\mathbf{A}}} A_{ij}^{(k)} U_{ik}$$

then we have the following update rules:

$$V_{jk}^{(t)} = \frac{-\text{sgn}(h) \max(|h| - \lambda_v, 0)}{2 \sum_{i \in \Omega_j^{\mathbf{A}}} U_{ik} U_{ik}} \tag{26}$$

Similar element-wise update rules are easily derived for the other factors.

3.1.2 Non-negative Collective Factorization

It is straightforward to extend our approach for *non-negative collective factorization* with a simple projection to constrain the element-wise updates to be positive. For instance, one may simply check if the update is negative and if so then it would be set to zero, otherwise the update is carried out in one of the following ways previously mentioned. Thus, the corresponding update rules for ℓ_2-norm is,

$$V_{jk}^{(t)} = \max\left(0, \frac{\sum_{i \in \Omega_j^{\mathbf{A}}} A_{ij}^{(k)} U_{ik}}{\lambda_v + \sum_{i \in \Omega_j^{\mathbf{A}}} U_{ik} U_{ik}}\right) \tag{27}$$

Analogous update rules are easily derived for the other factors. In addition, one may easily adapt other types of update rules for non-negative collective factorization. For instance, the non-negativity constraint is trivially added to other update rules that employ other constraints such as the update rule for sparsity constraints shown in Section 3.1.1, among many others.

3.2 Parallel Algorithm

Now we introduce a general parallel approach for the PCMF framework that succinctly expresses many of the variants. As mentioned previously, PCMF optimizes the objective function in (19) one element at a time. A fundamental advantage of this approach is that all such one-variable subproblems of a k latent feature are independent and can be updated simultaneously. While our current investigation focuses on shared-memory and thus suitable for parallelization on both CPU and GPU, we describe the parallelization procedure such that it could be used with a distributed memory architecture using MPI and others. Details to optimize the procedure for distributed memory architectures are given later in the discussion. Further, the parallelization procedure easily handles any arbitrary number of matrices (data sources) and is a straightforward extension.

We denote a parallel computing unit as a *worker*[2] and we let p denote the number of workers. A *job* (or task) is an independent unit of work and a *block* is defined as a ordered set of b jobs. Note that a job and task are interchangeable as well as the terms rows, row indices, and vertices. Further, all sets are assumed to be ordered and can be thought of as queues.

3.2.1 Simple Parallel Approaches and Challenges

Notice that the coordinates in each of the rows of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ are independent and thus can be updated simultaneously. Thus, a simple naive parallel approach is to partition the rows of $\mathbf{V}$ (vertices) into p sets of approximately equal size such that $\mathcal{I}^v = \{\mathcal{I}^v_1, ..., \mathcal{I}^v_p\}$ where $|\mathcal{I}^v_w| = \lceil n/p \rceil$. Now each worker $w \in \{1, ..., p\}$ updates the set of row indices $\mathcal{I}^v_w$ assigned to them. The fundamental assumption of the above static scheduling scheme is that each update takes approximately the same time to update. However, the time to perform updates may vary significantly and depends on many factors. For instance, it may depend on the underlying characteristics of each node $v_i \in \mathcal{I}^v_w$ (i.e., memory access pattern and number of memory accesses). Therefore, performing an update on a node with only a few neighbors is significantly faster than a node with many such neighbors. For a worker w, this leads to $\sum_{j \in \mathcal{I}^v_w} 4|\Omega^{\mathbf{A}}_j|$ time and likely differs for each worker. Furthermore many of the graphs found in the real-world have skewed degree distributions and thus only a handful of vertices are likely to have extremely large degrees whereas all other vertices are likely to have only a few connections. That finding essentially guarantees load balancing issues if not handled properly.

To further understand the fundamental importance of ordering, suppose all vertices of a given type (i.e., row indices of $\mathbf{V}$, $\mathbf{U}$, etc.) are ordered from smallest to largest by a graph property $f(\cdot)$ such as degree. Further, suppose b jobs are initially assigned in a round-robin fashion such that the first worker w is assigned the initial b vertices from the ordering π, whereas the second worker is assigned the next b vertices to update, and so on. Therefore, this gives us,

$$\overbrace{\underbrace{v_1, \ldots, v_{b-1}, v_b}_{w}, \; v_{b+1}, \ldots, v_{pb-1}, v_{pb}}^{\text{initial jobs assigned}}, \; v_{pb+1}, \ldots, v_{n-1}, v_n$$

where worker w is assigned the b vertices with largest degree and therefore this ordering initially ensures that the jobs assigned to worker w take at least as much time as the $w+1$ worker and so on. Moreover, if b is large enough the load is likely to be extremely unbalanced and therefore the jobs are fundamentally uneven as their cost depends on the number of nonzeros (i.e., observations/training instances). This clearly illustrates the fundamental importance of selecting an appropriate ordering strategy that distributes the work load evenly among the p workers.

We note that previous single matrix factorization approaches (such as CCD++) use explicit barriers between each rank-one updates. Hence, after performing a rank-one update on $\mathbf{V}$, all workers must wait before performing a rank-one update on $\mathbf{Z}$, $\mathbf{U}$, and any others. These approaches are slow due to the synchronization and static assignment of jobs to the workers.

3.2.2 Locking and Blocking Problems

For parallel algorithms to scale, it is important that all workers remain busy. Unfortunately, many parallel matrix factorization methods including CCD++ suffer from locking problems (also known as "the curse of the last reduced") that are typically due to ineffective load balancing strategies. Intuitively, this issue arises when workers must wait for the slowest worker to finish. While other problems may also lead to these locking problems, they are usually an effect and not the fundamental cause. For instance,as we shall see later, requiring explicit synchronization after each rank-one update may not significantly impact runtime and scalability as well as work load is appropriately balanced across the p workers such that all workers remain busy.

To gain intuition of our approach, we first present an oversimplification of our approach in Alg 1 that serves to highlight a few of the fundamental differences with our

[2] A worker is a *thread* in shared memory setting and *machine* in distributed memory architecture.

fast improved approach presented later in Section 3.2.3. Most importantly, that approach does not leverage asynchronous updates and does not allow an arbitrary order for performing the element-wise updates. In particular, the updates are performed in natural ordering (*e.g.*, $j = 1, ..., n$) and the kth latent factor of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ are updated in full. Instead, since single elements in $\mathbf{v}_k$, $\mathbf{z}_k$, and $\mathbf{u}_k$ may be computed independently, then one can easily imagine more complex orderings for updating the variables. For instance, one can update a single element in $\mathbf{v}_k$ then update an element in $\mathbf{z}_k$, $\mathbf{u}_k$, then continue alternating in this manner. Besides the two fundamental limitations of a fixed and simple ordering and synchronization between each rank-one update, that approach is also constrained by the specific update performed, despite there being many other updates that could just as easily be used (*e.g.*, non-negative updates, ℓ_1-norm updates, among others).

Algorithm 1 Simplified Parallel Collective Factorization

1 **Input:** $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{m \times m}$, d, λ_u, λ_v, α
2 Initialize $\mathbf{U} \in \mathbb{R}^{m \times d}$, $\mathbf{V} \in \mathbb{R}^{n \times d}$, and $\mathbf{Z} \in \mathbb{R}^{m \times d}$ uniformly at random
3 Set $\mathbf{E}^{\text{a}} = \mathbf{A}$ and $\mathbf{E}^{\text{b}} = \mathbf{B}$
4 **repeat** ▷ outer iterations $\tau = 1, 2, ...,$
5 **for** $k = 1, 2, ..., d$ **do**
6 $\mathbf{A}^{(k)} = \mathbf{E}^{\text{a}} + \mathbf{u}_k \mathbf{v}_k^\top$ **in parallel**
7 $\mathbf{B}^{(k)} = \mathbf{E}^{\text{b}} + \mathbf{u}_k \mathbf{z}_k^\top$ **in parallel**
8 **for** $t = 1, ..., \text{T}_{\text{inner}}$ **do**
9 **parallel for** $j = 1, 2, ..., n$ **do**
10 $V_{jk} = \frac{\sum_{i \in \Omega_j^{\mathbf{A}}} A_{ij}^{(k)} U_{ik}}{\lambda_v + \sum_{i \in \Omega_j^{\mathbf{A}}} U_{ik} U_{ik}}$
11 **end parallel**
12 **parallel for** $i = 1, 2, ..., m$ **do**
13 $Z_{ik} = \frac{\sum_{j \in \Omega_i^{\mathbf{B}}} B_{ij}^{(k)} U_{jk}}{\alpha + \sum_{j \in \Omega_i^{\mathbf{B}}} U_{jk} U_{jk}}$
14 **end parallel**
15 **parallel for** $i = 1, 2, ..., m$ **do**
16 $U_{ik} = \frac{\sum_{j \in \Omega_i^{\mathbf{A}}} A_{ij}^{(k)} V_{jk}}{\lambda_u + \sum_{j \in \Omega_i^{\mathbf{A}}} V_{jk} V_{jk}} + \frac{\sum_{j \in \Omega_i^{\mathbf{B}}} B_{ij}^{(k)} Z_{jk}}{\alpha + \sum_{j \in \Omega_i^{\mathbf{B}}} Z_{jk} Z_{jk}}$
17 **end parallel**
18 **end for**
19 $\mathbf{E}^{\text{a}} = \mathbf{A}^{(k)} - \mathbf{u}_k \mathbf{v}_k^\top$ **in parallel** ▷ $O(|\Omega^{\mathbf{A}}|)$ time
20 $\mathbf{E}^{\text{b}} = \mathbf{B}^{(k)} - \mathbf{u}_k \mathbf{z}_k^\top$ **in parallel** ▷ $O(|\Omega^{\mathbf{B}}|)$ time
21 **end for**
22 **until** stopping criterion is reached

3.2.3 Fast Asynchronous Approach

In this section, we introduce a fast generalized parallel collective factorization approach shown in Alg. 2. Most importantly, that approach serves as a fundamental basis for studying different objective functions/regularizers, ordering strategies, asynchronous updates, among many other variations. Alg. 2 starts by initializing the latent factor matrices $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ using some strategy (Line 1), *e.g.*, one may use $v_{jk} \sim \text{UniformReal}\left(0, 1/\sqrt{d}\right)$ for $1 \leq j \leq n, 1 \leq k \leq d$, and similarly for $\mathbf{U}$ and $\mathbf{Z}$. Line 2 applies graph sparsifiers to reduce the computational complexity. Both have been shown to be effective in reducing the runtime. Next, Line 3 obtains an initial ordering Π using a predefined metric or graph property f, *e.g.*, from largest to smallest in/out-degree. Now one may compute the initial residual matrices (Line 4). The algorithm proceeds iteratively using a maximum of T_{outer} iterations (Line 5) or until convergence or stopping criterion is reached (Line 19). Every outer iteration τ consists of updating the k latent factors via rank-one updates (Line 6). For the case that the observations in $\mathbf{A}$ and $\mathbf{B}$ are sparse (and/or dense but small enough to fit in memory), then we maintain a sparse k-residual matrix (Line 7 and Line 17). Now we perform T_{inner} rank-one updates for each of the kth latent factors.

Previous work assumed the one-variable subproblems are updated in the order given as input (i.e., natural ordering). This approach typically leads to load balancing issues and may significantly decrease the performance due to long wait times between updates. For instance, suppose vertices/rows are given as input to CCD++ in order of degree, thus if b is large enough, then the first worker is assigned the first b rows/vertices with the largest degree which corresponds to the fundamentally difficult and time consuming jobs. This leads to load balancing issues since the other $p - 1$ workers are likely to finish very rapidly and must wait until all such workers are finished. Using a smaller block b of jobs may reduce load balancing issues at the expense of increasing communication and overhead costs.

Instead, PCMF generalizes the previous work to allow vertices to be updated in any arbitrary order. This additional flexibility results in a significantly faster parallelization with shorter wait times between respective updates. Let $\pi_v = \{v_1, ..., v_n\}$, $\pi_z = \{z_1, ..., z_m\}$, $\pi_u = \{u_1, ..., u_m\}$ denote an ordering of the rows in $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$, respectively. For now, π_v, π_z, and π_u are assumed to be independently permuted in an arbitrary manner (largest degree, k-core, max error, etc.). From

Algorithm 2 A Generalized Algorithmic Template for Parallel Collective Factorization

1 Initialize latent factor matrices ▷ Select values uniformly at random or use Gaussian distribution, etc.
2 Use hybrid graph sampling and/or graph sparsification
3 Obtain a ordering Π of all unique nodes from every node type
4 Compute initial residual matrices in `parallel` asynchronously (if appropriate), otherwise one may simply set $\mathbf{E}^{\mathrm{a}} = \mathbf{A}$ and $\mathbf{E}^{\mathrm{b}} = \mathbf{B}$
5 **for** $\tau = 1, 2, ..., \mathrm{T}_{\mathrm{outer}}$ **do**
6 **for** $k = 1, 2, ..., d$ **do**
7 Compute k-residual matrices in `parallel` asynchronously via Eq. (17), Eq. (18), etc.
8 **for** $t = 1, 2, ..., \mathrm{T}_{\mathrm{inner}}$ **do**
9 **parallel for** next b jobs in Π in order **do**
10 **while** worker w has updates to perform in local queue **do**
11 Obtain next element to update from local ordering (dequeue job from w's queue)
12 Perform element-wise update using appropriate rule: Eq. (22), Eq. (23), Eq. (25)
13 **end while**
14 **end parallel**
15 Recompute ordering Π using bucket sort (if adaptive ordering strategy used)
16 **end for**
17 Update residual matrices in `parallel` asynchronously via Eq. (20), Eq. (21), etc.
18 **end for**
19 **if** stopping criterion is reached **then** terminate
20 **end for**
21 **return** factor matrices, i.e., $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$

this, let us denote Π as:

$$\Pi = \{\underbrace{v_1, \cdots, v_n}_{\pi_v^{(t)}}, \underbrace{z_1, \cdots, z_m}_{\pi_z^{(t)}}, \underbrace{u_1, \cdots, u_m}_{\pi_u^{(t)}}\} \tag{28}$$

where t denotes the inner iteration. This implies that one-variable updates are performed in the order given by π_v, then π_z, and so on. Hence, the parallel method selects the next b row indices (i.e., vertices) to update in a dynamic fashion from the Π ordering above.

More generally, our approach is flexible for updating individual elements in any order and is not restricted to updating all elements in π_v first (or the kth factor of $\mathbf{v}$) before updating π_z, and thus one can select the elements to update at a finer granularity. This is possible since we are focused on the approximation between a single element $a_{ij}^{(k)}$ in the k-residual matrix and the multiplication of U_{ik} and V_{jk} and similarly we are interested in the approximation between $b_{ij}^{(k)}$ and the multiplication of U_{ik} and Z_{jk}. Therefore, let us redefine Π as an arbitrary permutation of the set $\{v_1, ..., v_n, z_1, ..., z_m, u_1, ..., u_m\}$. The ordering may also change at each inner iteration and is not required to be static (Line 15). As an example, one may adapt the ordering at each inner iteration based on the error from the previous inner iteration. Further, we may choose to update only the top-x elements with largest error in the previous iteration. This ensures we focus on the variables that are most likely to improve the objective function and also ensures work is not wasted on fruitless updates that are unlikely to lead to a significant decrease in error.

In PCMF, each of the p workers are initially assigned a disjoint set of b vertices (i.e., row indices) to update. After a worker w completes its jobs (*e.g.*, updating all vertices in its assigned queue/vertex set), Line 9 in Alg. 2 dynamically assigns the next set of b rows to update (in order of Π). Thus, we assign jobs dynamically based on the availability of a worker. More formally,

each worker $w \in \{1, ..., p\}$ has a local concurrent `queue` which contains a set of $(j, \mathbf{v}_j)$ pairs to process where $\mathbf{v}_j \in \mathbb{R}^k$. Further, every such pair $(j, \mathbf{v}_j)$ is known as a job and corresponds to a one-variable update (from Section 3). Thus a worker w pops the next job off its local `queue` (Line 11), performs one or more updates (Line 12), and repeats. At the start, each worker $w \in \{1, ..., p\}$ initially pushes b job pairs onto its own `queue` using the ordering Π. If a worker w's local `queue` becomes empty, an additional set of b jobs are dynamically pushed into the `queue` of that worker. The specific set of jobs assigned to the w worker is exactly the next b jobs from the ordering Π (i.e., starting from the job last assigned to the p workers). Note that these jobs correspond to the rows that have yet to be updated in the current (inner) iteration (which are given by the ordering defined above). For instance, consider the case where $p = 2$ and $b = 3$, then $\{v_1, v_2, v_3\}$ and $\{v_4, v_5, v_6\}$ are initially assigned to each of the two workers. Now, once a workers queue is empty, it requests additional work, and the next batch of one-variable subproblems from the ordering Π are assigned. Thus, the first worker to complete the set of one-variable updates is assigned the next set of b jobs in the ordering, i.e., $\{v_7, v_8, v_9\}$, and so on until all such rows have been updated. This dynamic scheduling strategy effectively balances the load as it significantly reduces wait times between updates. Furthermore it has been shown to be significantly faster than the previous state-of-the-art for real-world networks with skewed degree and triangle distributions. The number b of such $(j, \mathbf{v}_j)$ job pairs assigned to a worker w is parameterized in PCMF so that the number of jobs assigned at a given time may be adapted automatically or set by the user.

To store the result from a single update, we index directly into the proper vertex/row position in the array and store the updated value. This has two implications. First we avoid additional storage requirements needed to store the intermediate results as done with previous approaches. Second the result from the update is stored using the same array and thus the current estimate may be used immediately. Let us note that CCD++ requires additional arrays of length m and n to store the intermediate results computed at each iteration. The results are then copied back afterwards.

A key advantage of the proposed PCMF framework is the ability to perform updates asynchronously without synchronization barriers between each rank-one update. Previous work in traditional matrix factorization usually requires that updates are completed in full for $\mathbf{U}_{:k}$ before moving on to $\mathbf{V}_{:k}$ and is explicitly implemented using barriers to ensure synchronization between rank-1 updates. For instance, CCD++ has multiple synchronization barriers at each inner iteration. However, this may cause most workers to wait for long periods while waiting for another worker that is either slow or assigned an extremely skewed work load. We relax such a requirement to allow for the inner iterates to be completely asynchronous. The rank one updates for $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ are completely asynchronously, and when there is no more jobs to be assigned to a worker w, that worker immediately grabs the next set of b jobs from the ordering Π. Our approach avoids the inner synchronization all together by dynamically assigning the next b jobs in the ordering Π to the next available worker w regardless of the rank-one update. Thus, workers do not have to remain idle while waiting for other workers to finish. We note that while there is a slight chance of a conflict, in practice it is hardly ever a problem (and even if a vertex gets an old update, then it is fixed within the next iteration).

We found that the job size used by CCD++ significantly impacts the parallel performance and leads to strikingly worse runtimes. As an aside, these issues may not have been noticed in the initial evaluation of CCD++ as it is proposed for the single matrix recommendation problem and evaluated on a relatively small number of recommender data sets. The shared-memory implementation significantly benefits from an effective *work-stealing* strategy. To avoid these problems, we use a work-stealing strategy so that workers do not remain idle waiting for the slowest to finish. Instead, jobs are pushed into each of the local worker queues (in some order), and once a worker completes the assigned jobs, we pop the last k jobs from the queue of the slowest worker and assign them to an available worker. Clearly this parallel approach significantly improves CCD++ as workers are never idle and always making progress by partitioning the jobs assigned to the slowest worker. For implementation, one may use the TBB library[3]. Note that this offers a far better dynamic load balancing since it keeps all workers fully utilized while not requiring us to optimize the number of jobs dynamically assigned upon completion. However, we still must select the worker to steal jobs from, the number of jobs to steal, and what specific jobs to steal. Here, one may efficiently estimate the number of optimal jobs to steal by using the total number of nonzero entries as a proxy.

Importantly, PCMF is naturally amendable for streaming, multi-core, and MapReduce architectures. This is in part due to the efficient, yet independent rank-1 updates and the limited number of passes over the matrices (compared to the state-of-the-art). In the PCMF framework, the number of updates b that is dynamically assigned to the workers is automatically optimized based on the size of the data (number of rows/cols), nonzero density, and

[3] `https://www.threadingbuildingblocks.org/`

number of latent dimensions. Though, the experiments in Section 4 do not use this technique for comparison. The other key advantages of this approach are summarized below:

- *Computations performed asynchronously whenever appropriate.* The inner-iteration is completely asynchronous. We also update the residual matrices in a non-blocking asynchronous fashion and carry out many other computations in a similar manner whenever possible.
- *Flexible ordering strategy allows finer granularity.* Flexible ordering strategy for performing updates and improving load balancing. Additionally, the previous CCD++ method and various others assume a natural ordering of the vertices (one-variable subproblems), which is the ordering given by the input graph data. In contrast, we propose using various other ordering strategies based on graph parameters such as degree, k-core, triangle counts, among others. We also propose other adaptive techniques based on maximum error in the previous iteration.
- *Updates performed in-place and current estimates utilized.* Furthermore updates are performed in-place and thus the current estimates are immediately utilized. This also reduces the storage requirements.
- *Memory and thread layout optimized for NUMA architecture.* In addition, we also have optimized PCMF for NUMA architecture [38] using interleaved memory allocations in a round robin fashion between processors. Results are discussed in Section 4.3.
- *Column-wise memory optimization.* The latent factor matrices are stored and accessed as a $k \times m$ contiguous block of memory. Thus, this memory scheme is optimized for column-wise updates where the k latent feature of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ are updated via $\mathrm{T}_{\text{inner}}$ iterations before moving on to the $k+1$ latent feature. Using this scheme allows us to exploit memory locality while avoiding caching ping pong and other issues by utilizing memory assignments that carefully align with cache lines.

We also note that the latent feature matrices $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ are initialized to be uniformly distributed random matrices where each entry is set by independently sampling a uniformly random variable. In contrast, the state-of-the-art matrix factorization method CCD++ initializes each entry of $\mathbf{V}$ to be zero whereas $\mathbf{U}$ is initialized using the standard C++ random number generator. Other strategies such as placing zero-mean spherical Gaussian priors on the latent feature matrices were also investigated. Additionally, we also investigated a variety of ranges such as $(0, \frac{1}{\sqrt{d}})$ for independently sampling a uniformly random variable.

3.2.4 Remarks

Now we briefly discuss an approach for distributed architectures. We divide the rows of $\mathbf{V}$, $\mathbf{Z}$, and $\mathbf{U}$ as evenly as possible across p workers. Thus each worker only stores $1/p$ rows of each. Additionally, each worker stores only the nonzero entries of the residual matrices that correspond to the partitioned rows assigned to the worker. Now for communication, we only require each worker $w \in \{1, ..., p\}$ to broadcast the rows of $\mathbf{V}^w_{:k}$, $\mathbf{Z}^w_{:k}$, and $\mathbf{U}^w_{:k}$ that were updated locally such that each worker has the latest $\mathbf{V}$, $\mathbf{Z}$ and $\mathbf{U}$ to update the residual matrices after the inner iterations. Thus, we avoid broadcasting the residual matrices. However, we send each rank-one update to all workers upon completion. Thus for each inner iteration, worker w must broadcast the latest $\mathbf{V}^w_{:k}$ to the other workers before updating $\mathbf{U}^w_{:k}$. However, observe that unlike the prior work where a communication step must be performed followed by a computation step (sequentially), we can instead transfer the $\mathbf{V}^w_{:k}$ rows thereby keeping the network fully utilized while updating $\mathbf{Z}^w_{:k}$ in parallel and thus keeping the CPU busy as well. This is in contrast to other strategies that utilize bulk synchronization between updates and suffers from the fact that when the CPU is busy, the network remains idle, and vice-versa. Notably this arises due to the dependencies of the latent feature matrices. Similarly, we may transfer $\mathbf{Z}^w_{:k}$ utilizing the network while (partially) updating $\mathbf{U}^w_{:k}$ by fixing $\mathbf{V}^w_{:k}$. Finally, the partial updates of $\mathbf{U}^w_{:k}$ may be transferred while finishing the updates to $\mathbf{U}^w_{:k}$ by fixing $\mathbf{Z}^w_{:k}$. This ensures both the network and CPU are busy simultaneously. Nevertheless, a worker w must wait until all $\mathbf{U}^w_{:k}$ entries that are required to update their respective row indices of $\mathbf{V}^w_{:k}$ and $\mathbf{Z}^w_{:k}$ are received. However, once a worker w receives the updates of $\mathbf{U}^w_{:k}$ required for even a single row index of $\mathbf{V}^w_{:k}$ and $\mathbf{Z}^w_{:k}$, we may immediately perform the update, and continue in this manner. This fast distributed parallelization scheme arises from the dependencies between the latent feature matrices. Clearly additional matrices may give rise to other types of dependencies which may lead to better parallelization and/or further improvements. For the updates where both the CPU and network are fully utilized, each machine reserves a few workers for communication (sending the updates) whereas the remaining workers compute the rank-one the updates in parallel.

The proposed distributed parallelization scheme ensures both the CPU and network are fully utilized by exploiting dependencies between the latent feature matrices. In comparison, CCD++ suffers from the fundamental problems with bulk synchronization since CCD++ alternates between communicating the rank-one updates and computing them. Therefore, the CPU and network

are never used simultaneously while also requiring both steps to be synchronized including computing the rank-one update and communicating it. More importantly, the synchronization between the sequential steps forces all machines to wait for the slowest machine to finish (i.e., "curse of the last reduced" problem). Thus, majority of machines in CCD++ remain idle between the each of the synchronized communication and computation steps.

Clearly the PCMF distributed parallelization scheme is significantly superior in scalability since the drawbacks of bulk synchronization are avoided as we the CPU and network remain simultaneously busy while also providing effective load balancing via an appropriate ordering strategy (that evenly distributes the work). To further enhance the scalability of our distributed scheme, we leverage asynchronous dynamic load balancing. Using this approach significantly improves scalability allowing us to scale up our approach for use on 1000+ processors. Notably, we avoid issues that arise from the naive "manager-worker" algorithm by eliminating the single manager as a bottleneck using a distributed work queue that is both asynchronous and performs dynamic load balancing via work stealing to ensure machines are always utilized and never idle. In particular, we leverage a distributed shared queue maintained across a number of servers in a hierarchical fashion (all servers maintain a consistent view of the global queue by communicating with one another directly, whereas each worker maintaining the queue has a fixed number of workers, and thus as the number of machines increase, additional servers are allocated).

Asynchronous dynamic load balancing has two key advantages: First, it is extremely scalable as the number of servers forming the distributed shared queue may be increased dynamically (as the number of workers increase) to handle the extra demand and thus avoids communication bottlenecks as the number of machines assigned to a server are fixed. Second, it provides instantaneous load balancing via *work-stealing* to ensure all machines remain busy. Intuitively, work is stolen from machines with unbalanced workloads and assigned to machines with significantly fewer jobs that are nearly complete and thus would otherwise be idle. This strategy effectively balances the load automatically. A number of asynchronous dynamic load-balancing libraries exist for this purpose. For instance, one may use the MPI-based ADLB library [18] and adapt it for the distributed PCMF.

3.3 Complexity Analysis

Let $|\Omega^{\mathbf{A}}|$ and $|\Omega^{\mathbf{B}}|$ denote the number of nonzeros in $\mathbf{A}$ and $\mathbf{B}$, respectively (*e.g.*, the user-by-item matrix $\mathbf{A}$ and the user-interaction matrix $\mathbf{B}$). As previously mentioned, d is the number of latent features in the factorization. The time complexity of PCMF is $\mathrm{O}\left(d\left(|\Omega^{\mathbf{A}}| + |\Omega^{\mathbf{B}}|\right)\right)$ time for a single iteration and therefore linear in the number of nonzeros in $\mathbf{A}$ and $\mathbf{B}$. Note that in the case that $\mathbf{A}$ and $\mathbf{B}$ represent user-item ratings and social network information, then $|\Omega^{\mathbf{A}}|$ usually dominates $|\Omega^{\mathbf{B}}|$ as it is usually more dense than the social network matrix $\mathbf{B}$ (i.e., in epinions approx. 13M ratings compared to 607k trust/friendship relations). Observe that each iteration that updates $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{Z}$ takes $O\big(d(|\Omega^{\mathbf{A}}| + |\Omega^{\mathbf{B}}|)\big)$, $O(|\Omega^{\mathbf{A}}|d)$, and $O(|\Omega^{\mathbf{B}}|d)$ time, respectively. Hence, the total computational complexity in one iteration is $O\big(d(|\Omega^{\mathbf{A}}| + |\Omega^{\mathbf{B}}|)\big)$. Therefore the runtime of PCMF is linear with respect to the number of nonzero elements in the collection of matrices given as input. Clearly this approach is efficient for large complex heterogeneous networks.

3.4 A Fast Nonparametric Model

Nonparametric parallel collective factorization methods (NPCMF) that are data-driven and completely automatic requiring no input from a user are clearly important for many real-world scenarios. In this work, we develop a non-parametric data-driven PCMF that discovers the best model completely automatic without user intervention while also being efficient and scalable for large graph data. Furthermore we also introduce an extremely fast relaxation of PCMF which can be used to quickly search over the space of PCMF models. Despite the practical importance of automatically determining the best model from the given data, investigations into such approaches are extremely rare, even for the well-studied single matrix factorization problem. Yet, the number of dimensions as well as regularization parameters and other model parameters may significantly impact accuracy and efficiency (as demonstrated later in Section 4). In addition, these parameters are application and data-dependent and require significant interaction by the user to tune them appropriately. Our goal is to automatically find the best model parameters given an arbitrary set of graphs and attributes. The other objective is to develop a fast relaxation method to search over the space of models. In other words, a data-driven non-parametric PCMF that is completely automatic with no user-defined parameters while also extremely fast and efficient for big graph data sets. These two requirements are critical

for real-time systems where (a) the data may be significantly changing over time and thus require constant tuning by the user or (b) simply inefficient and unable to handle the large continuous streaming graph data.

To search over the space of models from PCMF, one must first choose a model selection criterion. In this work, we primarily used information criterion of Akaike (AIC) [5], though other model selection criterion may also be used in PCMF such as Minimum Description Length (MDL) and many others. The AIC value is $C_{\texttt{AIC}} = 2x - 2\ln(\mathcal{L})$ where x is the number of parameters in the model, that is $x = d(m+n)$, and $\mathcal{L}$ is the maximized likelihood. Thus for a given $\mathbf{U} \in \mathbb{R}^{m\times d}$ and $\mathbf{V} \in \mathbb{R}^{n\times d}$ computed using PCMF with d-dimensions gives the following:

$$\ln(\mathcal{L}) = -\frac{1}{2\sigma^2}\|\mathbf{A} - \mathbf{U}\mathbf{V}^\top\|_F^2 \tag{29}$$

where σ^2 is the variance of the error. This criterion balances the trade-off between goodness of fit and the complexity of the model (or number of free parameters). Hence, the goodness of fit of a model is penalized by the number of estimated parameters and thus discourages overfitting. The model selected is the one that minimizes the information criterion. Since efficiency is of fundamental importance, we avoid precomputing a set of models for selection via AIC and instead use a greedy approach to effectively search over the space of reasonable or likely models from the bottom-up. Hence, we begin computing models using low parameter estimates and gradually increase each parameter until we discover a model that leads to a larger $C_{\texttt{AIC}}$ than found thus far. At this point, one may terminate or continue searching the next few models and terminate if a better model is not found in those few attempts. However, if such a model is found, then we reset the counter indicating the number of failed trials in a row. This last step is to provide more certainty that a global optimum was reached. To avoid storing all such models, we only need to store the best model found thus far. Thus, the method is space-efficient as well. In addition, we avoid computing $\|\mathbf{A} - \mathbf{U}\mathbf{V}^\top\|_F^2$ since we directly use the residual matrix $\mathbf{E}^{\text{a}}$ from PCMF.

Searching over the space of models is expensive and in certain settings may be impractical, even using the relatively fast search technique above. For this reason, we develop a fast relaxation variant of PCMF. Intuitively, the relaxation method performs a few iterations to obtain a fast rough approximation. The method reuses data structures and leverages intermediate computations to improve efficiency when searching the space of models. Furthermore the relaxation method serves as a basis for exploring the space of models defined by the PCMF framework (from Section 3). Most importantly, it is shown to be strikingly fast and scalable for large complex heterogeneous networks, yet effective, obtaining near-optimal estimates on the parameters.

Models are learned using the fast relaxation method and our model search technique is used to find the "best performing model" from the infinite model space. We also compared to a naive method that is essentially the vanilla PCMF from Section 3 using the early termination strategies. Overall, the fast relaxation method is typically 20+ times faster than the naive method, yet is only marginally more accurate than our fast relaxation. For instance, using the eachmovie data set from Table 1 we automatically found a nearly optimal model in only 9.5 seconds compared to 258 seconds using the naive approach. The fast non-parametric relaxation automatically identified a model with $d = 65$ whereas the slower but more accurate approach found $d = 75$. Nevertheless, once the parameters were learned automatically, we then used the corresponding models to predict the unknown test instances and used RMSE to measure the quality of the models. The difference was insignificant as the fast relaxation had 1.108 whereas the much slower approach gave 1.107 and thus the difference in RMSE between the two is insignificant. In a few instances, the model learned from the fast relaxation using AIC was of better quality (lower testing error).

Let us note that in previous work the model is typically selected arbitrarily and typically varies for each data set [39]. Instead, we perform model selection automatically using AIC and perform a fast relaxation method for computing a rough approximation. As previously mentioned, this data-driven non-parametric PCMF arose from necessity as it is essential for many practical situations and real-time systems (*i.e.*, tuning by users are not possible or expensive and efficiency is critical due to the sheer size of the streaming graph data). Note that if external attributes are available, then another approach is to search the space of models and select the one that gives rise to the best performance (*i.e.*, accuracy or application-specific metric). Finally, it is also straightforward to adapt the information criterion above for an arbitrary number of matrices and alternate objective functions[4].

3.5 Collective Graph Sparsifiers

Many real-world graphs are often sparse with skewed degree distributions. This implies that a small set of nodes give rise to the vast majority of links (*i.e.*, nonzeros in the matrix representing ratings, etc). We have

[4] The likelihood expression assumes noise in the data is Gaussian

also observed through numerous experiments that removing links (ratings, etc) from high degree nodes does not significantly impact accuracy/quality of the model, but in some instances may reduce the time for learning quite significantly. On the other hand, removing observations from nodes with only few observed values may significantly impact our ability to predict that node's preferences, etc. From the above observations, we propose a strikingly effective hybrid graph sampling approach that has two main steps. First, we add the set of nodes and their edges that satisfy $|N(u_i)| \leq \delta$ where $|N(u_i)|$ is the number of neighbors/edges of u_i and δ is a threshold. For each of the nodes not included previously, we sample uniformly at random δ edges from each of the remaining nodes 1-hop local neighborhood. Here we set $\delta = \lceil 1/m \sum_{i=1}^{m} |\Omega_i^{\mathbf{A}}| \rceil$. However, unlike the previous node sampling where all edges are included, we instead bias the sampling towards low degree vertices (*i.e.*, those with few ratings or friendships in social networks). Intuitively, if a node with large degree is sampled, then a fraction of its edges are sampled uniformly at random[5], otherwise all edges are sampled. This sampling method may be repeated for each additional matrix or alternatively weighted by the previously sampled nodes and edges of multiple types.

Learning time is significantly reduced using hybrid graph sampling while maintaining the quality of the solutions. Furthermore the hybrid graph sampling also improves load balancing as the resulting degree distribution is far less skewed than the original. Additionally, this strategy forces many nodes to have approximately the same edges and thus the individual jobs take about the same time to compute. Finally, this approach may be directly leveraged for ensemble learning as well as our nonparametric data-driven approach discussed in Section 3.4.

3.6 Further Optimization Details and Improvements

3.6.1 Adaptive Coordinate Updates

There are a number of interesting PCMF variants based on the strategy used to select the next coordinate to update. One of the proposed adaptive variants selects the coordinate that gave rise to the largest decrease in error in the previous iteration. Thus, the coordinate with the largest decrease in error is chosen to update first, and so on. Consider solving the following one-variable subproblem:

$$\mathrm{u}_i^\star \leftarrow \min_{\mathrm{u}_i} J(\mathrm{u}_i) = \sum_{j \in \Omega_i} \left(A_{ij}^{(k)} - \mathrm{u}_i \mathrm{v}_j\right)^2 + \lambda \mathrm{u}_i^2$$

where u_i is updated to the optimal $\mathrm{u}_i^\star$. Thus, this leads to the following decrease in the objective function:

$$J(\mathrm{u}_i) - J(\mathrm{u}_i^\star) = (\mathrm{u}_i^\star - \mathrm{u}_i)^2 \Big(\lambda + \sum_{j \in \Omega_i} \mathrm{v}_j^2\Big)$$

Therefore, we select the ith variable with largest decrease:

$$i^\star = \mathrm{argmax}_i \, J(\mathrm{u}_i) - J(\mathrm{u}_i^\star), \forall 1 \leq i \leq m$$

We also proposed more efficient variants that use graph parameters to determine an effective update ordering such as vertex degrees, k-core numbers, triangle counts, triangle-core numbers, and number of colors used to color a vertex neighborhood. These update orderings only need to be computed once and are most are straightforward to parallelize. Furthermore we also proposed techniques to efficiently compute the top-k variables with largest error and perform a few additional updates on this set of variables.

3.6.2 Early stopping criterion

Given as input is a small positive ratio ϵ (usually $\epsilon = 10^{-3}$), then we terminate an inner-iteration early: For each outer-iteration τ, we have the maximum function reduction Δ_{max} given by the past iterations (for τ). If the current inner-iteration reduces the localized objective function to be less than $\epsilon \Delta_{max}$, then we terminate the inner-iterations early, and update the residuals. For this, we need to sum up all the function value reductions from the single variable updates:

$$\sum_i (\mathrm{u}_i^\star - \mathrm{u}_i)^2 \cdot (\lambda + \sum_{j \in \Omega_i^{\mathbf{A}}} \mathrm{v}_j)$$

This value represents the total objective function reduction from all i and represents a complete update of $\mathbf{u}$ for a single iteration.

4 Experimental Results

The experiments are designed to answer the following questions:

- *Parallel Scaling.* Does PCMF and its variants scale well for large data and how does it scale for different types of data?

[5] Edges were also sampled inversely proportional to the degree of each neighborhood node.

Table 1 Data description and statistics.

graph data sets	(semantics) node & edge types	m	n	$\|\Omega\|$	max d_{out}	max d_{in}	$\bar{d}_{\text{out}}$	$\bar{d}_{\text{in}}$	weight range	μ_{out}	μ_{in}	ρ	$\|\Omega^{\text{test}}\|$
amazon	users-rate-items	2.1M	1.2M	4.7M	9.8k	2.5k	2.18	3.79	1-5	3.57	3.77	10^{-6}	1.2M
dating	users-rate-users	135k	169k	16M	24k	32k	121.83	97.7	1-10	5.93	6.03	<0.01	868k
eachmovie	users-rate-movies	1.6k	74k	2.2M	26k	1.2k	1.4k	30.22	1-6	3.99	3.45	0.02	562k
epinions	users-rate-items	120k	756k	13M	159k	1.2k	111.18	17.73	1-5	4.64	4.31	<0.01	137k
	users-trust-users	120k	120k	607k	1.8k	3.2k	5.03	5.03	1	0.47	0.35	10^{-5}	
flickr	users-friend-users	1.9M	1.9M	18M	21k	13k	9.72	9.72	1	0.68	0.8	10^{-6}	
	users-join-group	1.7M	104k	8.5M	2.2k	35k	4.94	82.45	1	0.23	1	10^{-5}	4.5M
lastfm	users-listento-songs	992	1.1M	15M	146k	14k	15k	14.13	1	1	0.93	0.01	
	users-listento-band	992	174k	19M	183k	115k	19k	110.01	1	1	1	0.11	3.8M
livejournal	users-friend-users	5.2M	5.2M	39M	12k	928	7.56	7.56	1	0.55	0.95	10^{-6}	
	users-join-groups	3.2M	7.5M	112M	300	1.1M	35.08	15	1	1	1	10^{-6}	9.8M
movielens10M	users-rate-movies	72k	65k	9.3M	7.3k	29k	129.97	142.8	0.5-5	3.42	0.52	<0.01	699k
stackoverflow	users-favorite-posts	545k	97k	1.0M	4.0k	4.9k	1.91	10.78	1	0.86	0.92	10^{-5}	260k
yelp	users-rate-businesses	230k	12k	225k	575	824	0.98	19.53	1-5	0.73	3.66	10^{-5}	4.6k
youtube	users-friend-users	1.2M	1.2M	4.0M	23k	20k	3.42	3.42	1	0.46	0.86	10^{-6}	
	users-join-groups	664k	30k	293k	1.0k	7.6k	0.44	9.75	1	0.14	1	10^{-5}	989k

- *Effectiveness.* Is PCMF useful for a variety of data and applications? How does it compare to recent state-of-the-art coordinate descent approaches? Does incorporating additional information (*e.g.*, social, group, attributes) improve the quality of the learned model compared to using only a single data source such as the user-item ratings matrix?
- *Generality.* Is PCMF effective for a variety of heterogeneous data sources and prediction tasks?

To demonstrate the generality of our approach we apply PCMF for a number of predictive tasks and data sets.

4.1 Data and Experimental Setup

4.1.1 Platform

For our experiments, we use a 2-processor Intel Xeon X5680 3.33GHz CPU. Each processor has 6 cores (12 hardware threads) with 12MB of L3 cache. Finally, each core has 2 threads and 256KB of L2 cache. The server also has 96GB of memory in a NUMA architecture. The PCMF framework was implemented entirely in C++ and we also deployed it in our fast high-performance RECommendation PACKage called RECPACK. Note that the case where we have only a single matrix to factorize is a special case of PCMF which we denote as PCMF-BASIC.

4.1.2 Data

For evaluation we used a variety of data types with different characteristics. The collection of data used in the evaluation is summarized in Table 1. Whenever possible we used cross-validation with the original training/test splits for reproducibility. For the others, test sets were generated by randomly splitting the instances repeatedly. Test sets consisted of $\{1\%, 20\%, 50\%, 80\%\}$ of the instances. Many results were removed due to brevity, but are consistent with the reported findings. Unless otherwise specified, we used only the bipartite graph, social network, or other additional data if it exists (see Table 1). To evaluate PCMF we focused primarily on the large Epinions data [20] as it has both user-product ratings (1-5 rating) and a user-user social network. This data has also been extensively investigated and is significantly different than the traditional single matrix problems used in recommendation. Note that some results/plots were removed for brevity as well as other

data sets. Let us also note that some plots compare only PCMF-BASIC due to data containing only a single graph (see Table 1). The data used in our experiments is accessible online at Network Repository[6] (NR) [26].

4.1.3 Evaluation Metrics & Comparison

For evaluation we hold out a set of observations for testing, then learn a model with the remaining data and use it to predict the held-out known observations. Let Ω^{test} denote the instances in the test set (*e.g.*, ratings held out in the learning phase). Given a user i and item j from the test set $(i,j) \in \Omega^{\text{test}}$, we predict the rating for the $(i,j)^{\text{th}}$ entry in $\mathbf{A}$ as $\langle \mathbf{u}_i, \mathbf{v}_j \rangle$ where $\langle \cdot, \cdot \rangle$ is the inner Euclidean product of the user and item vectors, respectively. To measure the prediction quality (error) of PCMF and its variants against other approaches, we use root mean squared error (RMSE) [37,33,36]. It is simply,

$$\sqrt{\frac{\sum_{(i,j)\in\Omega^{\text{test}}} \left(A_{ij} - \langle \mathbf{u}_i, \mathbf{v}_j \rangle\right)^2}{|\Omega^{\text{test}}|}} \tag{30}$$

The mean absolute error (MAE) is also preferred in some instances. It is defined as:

$$\frac{1}{|\Omega^{\text{test}}|} \sum_{(i,j)\in\Omega^{\text{test}}} \text{abs}\left(A_{ij} - \langle \mathbf{u}_i, \mathbf{v}_j \rangle\right) \tag{31}$$

where abs$(\cdot)$ is used for absolute value to avoid ambiguity with set cardinality $|\Omega^{\text{test}}|$ and $\text{abs}\left(A_{ij} - \langle \mathbf{u}_i, \mathbf{v}_j \rangle\right)$ denotes the absolute value of the difference. Results for Normalized RMSE are reported in a few instances. This metric is normalized using the min and max values predicted by the model and is defined as:

$$\frac{RMSE\ (Eq.\ 30)}{\max\limits_{(i,j)\in\Omega^{\text{test}}} \langle \mathbf{u}_i, \mathbf{v}_j \rangle - \min\limits_{(i,j)\in\Omega^{\text{test}}} \langle \mathbf{u}_i, \mathbf{v}_j \rangle} \tag{32}$$

where the error is bounded between 0 and 1 (*i.e.*, a NRMSE of 0 indicates perfection) and can be easily interpreted. Additional evaluation metrics such as Symmetric MAPE (SMAPE) were also used, however, results were removed due to brevity. We also report time in seconds for training.

We set $\lambda_v = \lambda_u = 0.1$ and $\text{T}_{\text{outer}} = \text{T}_{\text{inner}} = 5$, unless otherwise noted. For comparison, we used the exact code from [39].

[6] `http://networkrepository.com`

4.2 Parallel Scaling

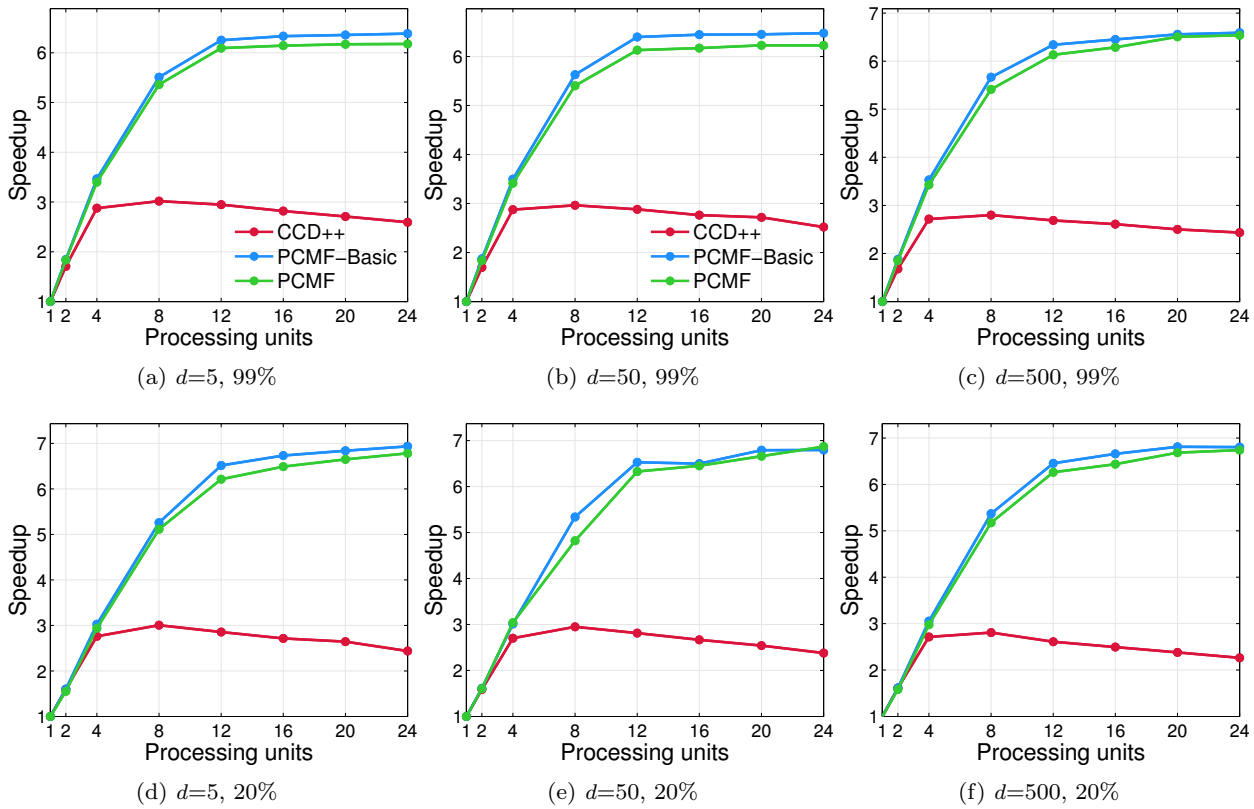
This section investigates the speed and scalability of PCMF for single factorization and collective factorization. Speedup (and efficiency) is used to evaluate the effectiveness of a parallel algorithm since it measures the reduction in time when more than a single worker is used. We vary the number of workers and measure the running time of each method. Speedup is simply $S_p = \frac{T_1}{T_p}$ where T_1 is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm with p workers (cores). Further, let efficiency be defined as $E_p = \frac{T_1}{pT_p}$ where p is the number of workers. Note that the machine has a total of 12 cores and thus each worker typically represents a unique core. However, we also investigate using up to 24 threads with the use of hyper-threading (and its potential advantages).

4.2.1 Speedup and Efficiency

How does PCMF and PCMF-BASIC (single matrix) scale compared to other coordinate descent methods? To answer this question, we systematically compare the speedup of PCMF, PCMF-BASIC, and CCD++[7] on the epinions data. In particular, models are learned using different amounts of training data and for each we also vary the number of latent dimensions. As shown in Figure 4, both PCMF and PCMF-BASIC are found to significantly outperform CCD++ across all models learned using different amounts of training data and dimensions. This finding is also consistent using various other parameter settings as well. See Section 4.2.3 for a detailed discussion into the factors that contribute to the significant improvement. We also find that PCMF-BASIC scales slightly better than PCMF, though this is expected since PCMF factorizes significantly more data and thus more likely to have performance issues due to caching or other effects. Similar results are found for other data sets, see Figure 6.

To further understand the effectiveness of the strategies leveraged by PCMF such as the asynchronous updates and appropriate ordering of updates, a simple PCMF variant that factorizes only a single matrix is evaluated. While PCMF collectively factorizes an arbitrary number of matrices, we can nevertheless investigate the special case of PCMF that factorizes a single matrix. Figure 5 evaluates the scalability of our approach on the MovieLens where we vary the number of latent factors learned. For each model, we measure the time in seconds vs. number of processing units, speedup, and efficiency. Just as before, PCMF-BASIC outperforms the others in

[7] A recently proposed parallel coordinate descent method for recommendation



(a) d=5, 99% (b) d=50, 99% (c) d=500, 99%

(d) d=5, 20% (e) d=50, 20% (f) d=500, 20%

Fig. 4 Varying amount of training data and dimensionality of models. The above plots demonstrate the scalability of PCMF and PCMF-BASIC on the epinions benchmark data set. Similar results arise for $d = \{10, 20, 40\}$.

all cases. Notably, we find the single matrix variant of PCMF outperforms CCD++ (and consequently the other optimization schemes using ALS, SGD, etc. as shown in [39]). This indicates the effectiveness of our approach that uses asynchronous updates, ordering, and other strategies that help avoid the locking problem.

Figure 7 compares the efficiency of the methods across a variety of data sets that have significantly different characteristics. Just as before, we find the PCMF variants outperform the other method. In particular, the individual processing units of the PCMF variants are shown to be more effectively utilized in parallel. A few of the interesting speedup results are also shown in Figure 8.

We have also used many other benchmark data sets in our comparison and in all cases found that PCMF converged faster to the desired error. For instance, in the dating agency data (users-rate-users) it took us 95 seconds compared to 138 given by the best state-of-the-art method, whereas for yelp (users-rate-businesses) it took us 2 seconds compared to 5, and similar times were observed for eachmovie (users-rate-movies). We also experimented with numerous other variants with different update strategies. Regardless of the variant and parameters, the best scalability arises from using one of the PCMF variants. We also compare the methods using performance profiles across a number of data sets. Performance profile plots allow us compare algorithms on a range of problems [9]. One example is shown in Figure 9. Just like ROC curves, the best results lie towards the upper left. Suppose there are N problems and an algorithm solves M of them within x times the speed of the best, then we have the point $(\tau, p) = (\log_2 x, M/N)$ where the horizontal axes reflects a speed difference factor of 2^τ.

4.2.2 Hyper-threading

For each processor core (physically present), hyper-threading addresses two logical cores and shares the workload between them whenever possible. Performance increases of up to 30% have been observed, though performance improvement is highly dependent on the application and may vary depending on the nature of the computation as well as access patterns. Thus an ideal speedup is about 15.6 on a 12-core system. However, in some cases overall performance may decrease with the use of hyper-threading.

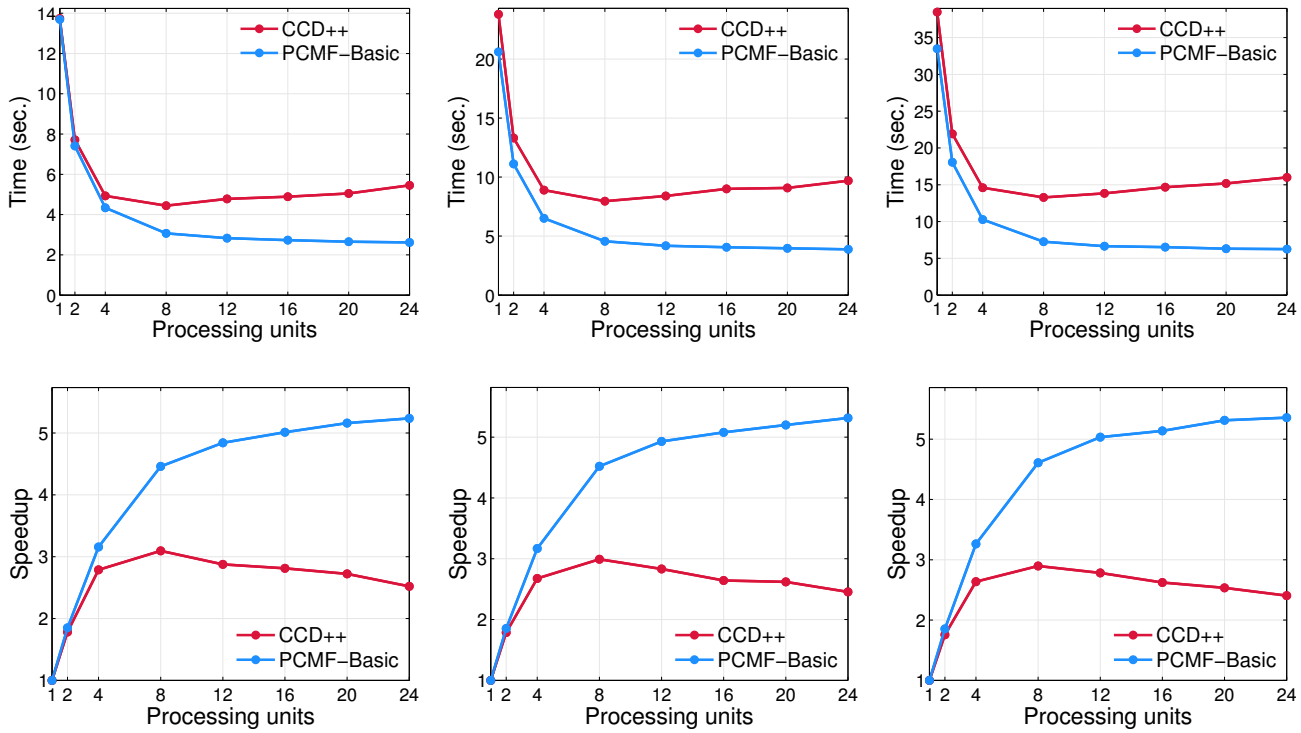
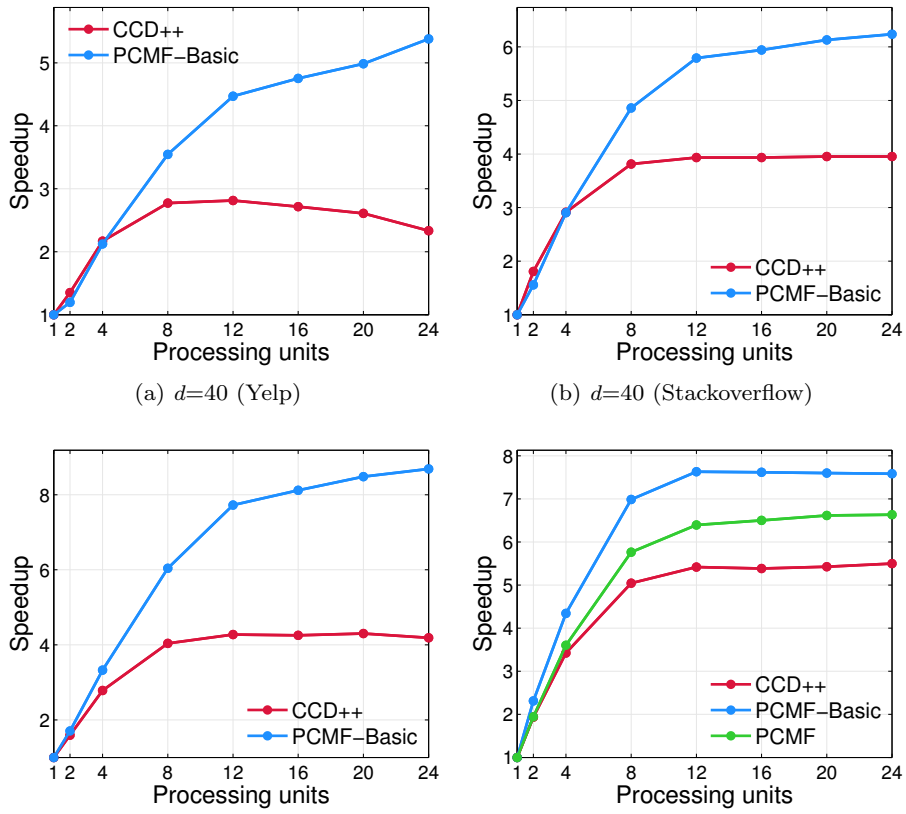


Fig. 5 Scalability. We use the MovieLens 10M data and vary the number of latent factors. For each model we measure the time in seconds, speedup, and efficiency. From these results, we compare PCMF-BASIC (*i.e.*, PCMF using only single data source) compared to CCD++. In all cases, PCMF-BASIC outperforms CCD++. This is due to the asynchronous nature of PCMF and its variants whereas CCD++ has a variety of locking and synchronization problems and thus more likely to result in significant slowdowns due to the fact that all workers must wait for the slowest worker to finish between each inner iteration.

To evaluate the utility of hyper-threading, methods are compared when the number of processing units is greater than the number of cores. In our case, this corresponds to the case where the number of processing units is greater than 12. As the number of processing units are increased from 12 to 24 shown in Figure 4, hyper-threading consistently improves performance for both PCMF and PCMF-BASIC whereas the performance of CCD++ is found to decrease with additional processing units via hyper-threading. Similar results are found for other data and parameter settings, see Figure 8, Figure 6, and Figure 5 among others. Hyper-threading typically decreases the runtime of the PCMF variants and thus in most cases there is an advantage. However, PCMF-BASIC and other methods result in longer runtimes due to synchronization issues and the fact that updates are not computed in-place as done by PCMF and thus require additional storage.

4.2.3 Discussion

Furthermore, many factors contribute to the significant improvement in scalability observed using the PCMF framework and its variants. First, the dynamic load balancing of CCD++ may lead to large wait times and is largely due to the fact that jobs are dynamically assigned for each rank-1 update and workers must wait until each update is entirely completed. Since the jobs assigned to each processor are defined at the level of individual updates and each update is likely to require a different number of computations, then it is straightforward to see that the jobs assigned to the workers in CCD++ are fundamentally uneven. As a result, CCD++ may have a number of workers idle between the rank-1 updates. Instead PCMF performs updates asynchronously and therefore does not suffer from the CCD++ problems (*e.g.*, frequent synchronization). We also note that CCD++ does not perform the updates in any special order. However, we observed that significant speedups may arise by ordering the updates appropriately. In particular, the updates may be ordered by degree or by the residual difference, among others. Furthermore the scalability of CCD++ is fundamentally dependent on the chunk-size used in their implementation.



(a) d=40 (Yelp) (b) d=40 (Stackoverflow)

(c) d=40 (Amazon) (d) d=40 (Flickr)

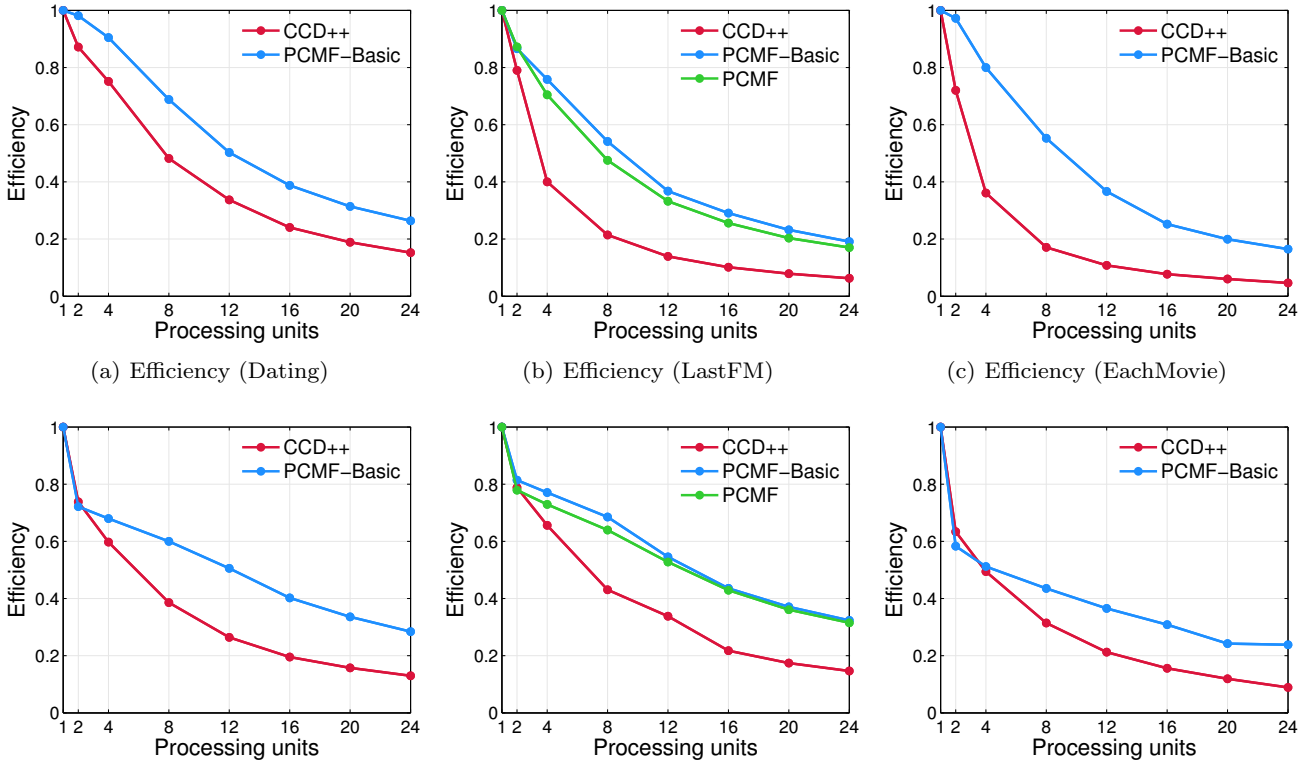
Fig. 6 Comparing speedup across different types of data and characteristics. Models are learned from each data source using $d = 40$ for comparison across the various data sources. See Table 1 for data set statistics.

4.3 Memory and Thread Layout

In addition, we also have optimized PCMF for NUMA architecture [38]. Since the local memory of a processor can be accessed faster than other memory accesses, we explored two main memory layouts including bounded where memory allocated to the local processor (socket) and interleaving memory allocations in a round robin fashion between processors. Overall, we found the layout of memory to processors had a large effect on scalability and in particular, the best scalability arises from interleaved memory (due to caching benefits). Note that we also experimented with thread layout and found no significant difference. Speedups were computed relative to the fastest run with a single thread, which always occurred using memory bound to a single processor. A representative sample of the results across a variety of data sets using different number of latent dimensions are provided. Speedup of the various memory and thread layout combinations are shown in Figure 10. Interleaved memory typically outperforms the bounded memory layout, regardless of the thread layout strategy used. Additionally, this finding is consistent across the various data sets and dimensionality settings. Therefore, PCMF leverages the interleaved memory layout, unless specified otherwise.

4.4 Predictive Quality

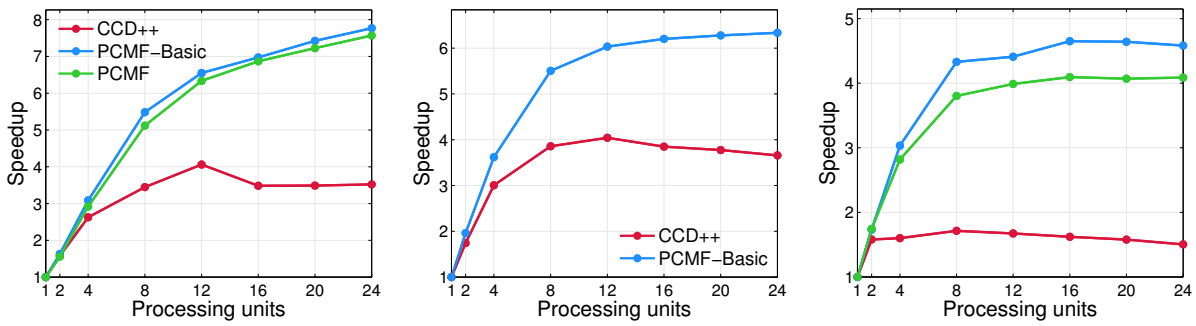
How effective is the proposed parallel collective factorization approach compared to the current state-of-the-art coordinate descent method? Recall the hypothesis was that incorporating an arbitrary number of potentially heterogeneous networks represented as matrices will improve the quality of the learned model. Therefore, we first use PCMF to jointly factorize the target matrix used for prediction as well as any number of additional data (*e.g.*, social network, group memberships, user-item matrix). Using the learned model from PCMF, we measure the prediction quality using an evaluation criterion (*e.g.*, RMSE, NRMSE) that quantifies the quality of the model for prediction of the held-out set of observations from the target matrix (not used in learning). The prediction quality of PCMF is compared to the state-of-the-art parallel approach (CCD++) that uses only the



(a) Efficiency (Dating) (b) Efficiency (LastFM) (c) Efficiency (EachMovie)

(d) Efficiency (StackOverflow) (e) Efficiency (YouTube) (f) Efficiency (Yelp)

Fig. 7 Comparing efficiency of various methods across a variety of different types of data. For each data set, we learn a model with $d = 100$ dimensions and measure efficiency. The efficiency measure captures the fraction of time a processing unit is used in a useful manner. As mentioned previously, it is defined as the ratio of speedup to the number of processing units and thus in an ideal system where speedup of p processing units is p also has an efficiency of one. Though in practice, efficiency is between 0 and 1 depending on the effectiveness of each processing unit and how well they are utilized.



(a) d=100 (YouTube) (b) d=100 (Dating) (c) d=100 (LastFM)

Fig. 8 Comparing the methods using speedup. Models are learned using $d = 100$.

target matrix. Results are summarized in Figure 11 for a variety of different data sets. This experiment uses NRMSE since the error is bounded between 0 and 1 and allows us to easily compare the relative error of the models across different data sets. In all cases, we find that PCMF outperforms the baseline as expected. Intuitively, this indicates that modeling the additional matrix collectively is useful and includes useful information that is leveraged in PCMF. The lowest NRMSE is found for epinions whereas the largest difference in NRMSE is from the youtube and flickr data sets. Most importantly, both flickr and youtube are also the most similar as they both contain social information as well as group membership information. Similar results are found when α and λ vary as shown later in Section 4.5. Overall, we find that PCMF improves the accuracy of

predictions (*e.g.*, recommendations) and the improvement is statistically significant. We also compare the single-matrix variant of PCMF called PCMF-BASIC and find essentially very similar results. To understand the accuracy gain with PCMF, we also evaluated variations of PCMF that performed the rank-1 updates in a different order, used different regularization, and a variant that updated entries in a biased manner (top-k entries with largest error). Similar results were observed in majority of cases (plots removed for brevity).

Figure 12 compares the PCMF and PCMF-BASIC variants that arise from specifying the various combinations of inner and outer iterations. For this experiment, MAE is used to quantify the quality of the learned models. The PCMF model with the best quality is found when 2 inner and 5 outer iterations are used. In contrast, the best PCMF-BASIC model arises from 2 inner and 3 outer iterations. For both PCMF and PCMF-BASIC, the fastest variants arise when a single outer iteration is used, regardless of the inner iterations. Thus, the runtime depends more strongly on the number of outer iterations than the inner iterations as each outer iteration requires updating the residual matrices.

4.5 Impact of α

PCMF seamlessly allows for additional information to be used in the factorization. In these experiments, we vary the α parameter in the collective factorization framework and measure the effects. The α parameter controls the amount of influence given to the interaction matrix in the model. In particular, if $\alpha = 0$ then the additional information (*e.g.*, social interactions) are ignored and only the initial target matrix of interest is used in the factorization (*e.g.*, the user-item matrix). Conversely, if $\alpha = \infty$ (or becomes large enough), then only the social interactions are used (as these dominate). Figure 13 evaluates the impact of α used in our PCMF and includes PCMF-BASIC along with CCD++ as a baseline. We find that α significantly impacts performance. This indicates that incorporating additional information (*e.g.*, social network) improves performance over what is obtainable using only the user-item matrix. In particular, the prediction accuracy increases as α increases, but then begins to decrease as α becomes too large. Hence, the optimal accuracy is achieved when all data sources are used in the factorization.

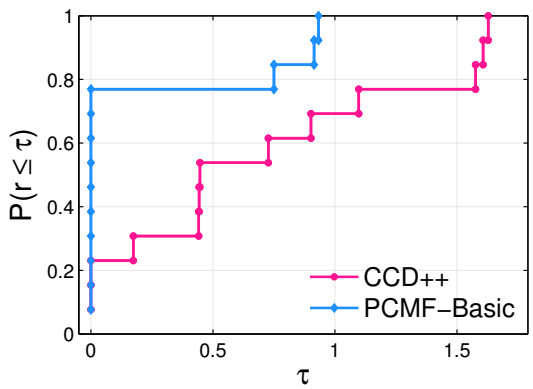
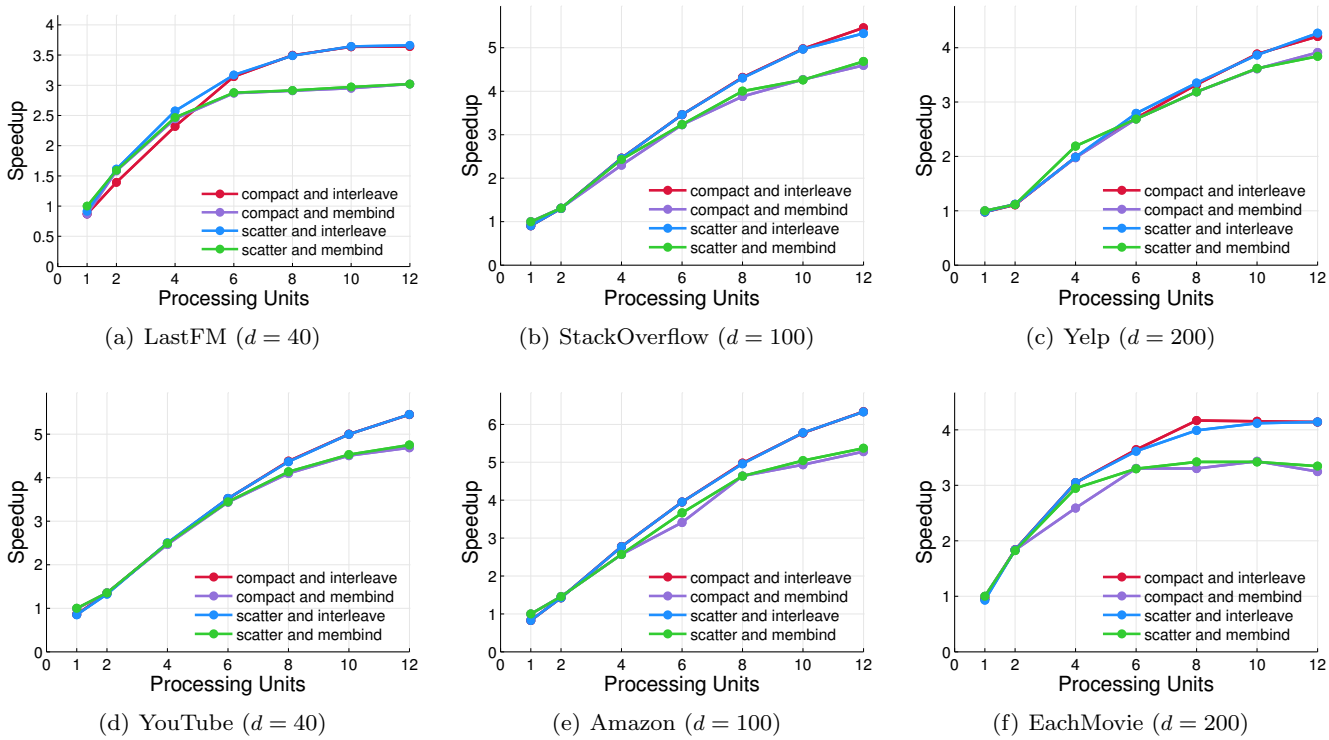


Fig. 9 Performance profile comparing methods on a number of problem instances.

To understand the impact of α when λ changes and vice-versa, we learn models for $\alpha \in \{0, 0.1, 0.5, 1, 5, 100\}$ and $\lambda \in \{0, 0.1, 0.5, 1, 5, 100\}$ and use RMSE to evaluate the quality of the model learned from each combination. Figure 14 searches over the parameter space defined by α and λ. Models learned using $\alpha = 0$ indicates the additional information is essentially not used and similarly for $\lambda = 0$. For this particular data set, the best model as quantified by RMSE is found when $\alpha = 1$ with $\lambda = 0.1$. Nevertheless, other types of data sets give rise to different optimal parameter settings, though in general we find similar behavior when using extremely small or large parameter settings.

4.6 Impact of the Latent Dimension

For this experiment, we investigate the impact on accuracy and scalability of PCMF when learning models with different dimensions. Using a smaller number of dimensions d leads to faster convergence as updates are linear to d as noted in Section 3. Alternatively, as the dimensionality parameter d increases, the model parameter space expands capturing weaker signals in the data at the risk of overfitting. The scalability of both PCMF and PCMF-BASIC does not significantly change as we vary the number of latent dimensions learned. One example of this behavior is shown in Figure 4 for $d \in \{5, 50, 500\}$ and for each of the different amounts of training data available for learning. Similar results are found for other parameter settings as well. Another example is shown in Figure 5. In most cases, models learned using a relatively small number of dimensions are likely to be more useful in practice. Since learning models with additional dimensions are unlikely to significantly increase quality, whereas increasing the number of latent factors increases the storage requirements as well as the time taken for learning and prediction (at a constant rate as d increases). Furthermore models with a large number of dimensions also impact the ability to make predictions in real-time. Thus the additional quality gained using a significantly larger number of dimensions are likely not worth the additional cost in terms of time and space.



(a) LastFM ($d = 40$) (b) StackOverflow ($d = 100$) (c) Yelp ($d = 200$)

(d) YouTube ($d = 40$) (e) Amazon ($d = 100$) (f) EachMovie ($d = 200$)

Fig. 10 Comparing the speedup of different memory and thread layouts. Speedup of PCMF and its variants for the four methods that utilize different memory and thread layouts. For the above experiments, we set $\lambda = 0.1$ and $\alpha = 0.1$. We use various data sets and use different number of latent dimensions for representativeness. Others were removed for brevity.

4.7 Serving Predictions in Real-time Streams

We also developed fast and effective methods for serving recommendations in a streaming fashion [1,2,3,10]. In this section, we investigate PCMF-based methods in a streaming setting where user requests are unbounded and continuously arriving over time. However, the rate of requests may be fluctuating over time with temporal patterns such as bursts, periodicity, and seasonality at different time scales. The problem we focus on is as follows: Given a continuous stream $\mathcal{S} = \{\mathcal{S}_1, ..., \mathcal{S}_k, \mathcal{S}_{k+1}, ...\}$ of user requests, the objective is to predict a set of personalized recommendations for each active user in the system (*i.e.*, begins browsing a retail site such as Amazon or Ebay). The stream of requests are generated synthetically using resampling to select users uniformly at random. Resampling sequentially induces an implicit ordering on the requests which is a fundamental requirement of streams. In the experiments, users are assumed to be arriving faster than can be served, allowing for a more principled evaluation. Moreover, this also allows us to compare the parallel properties of the methods without worrying about the stream rate. As an aside the streaming methods are also tunable[8], *e.g.*, as d decreases, the number of requests per second increases at the expense of accuracy.

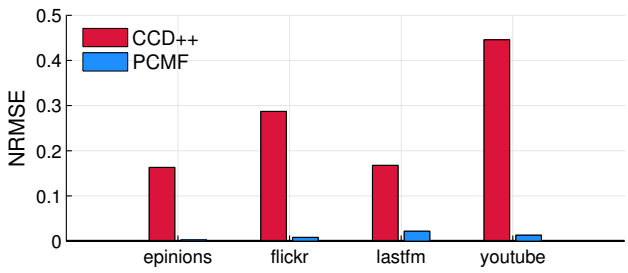


Fig. 11 Comparing quality of the models learned.

The first method called PCMF-NAÏVE uses PCMF to compute weights for each item using $\mathbf{V}$, and keeps track of the top-k items with the largest likelihood. As a baseline for comparison, we use the basic PCMF to compute weights for each item using $\mathbf{V}$, and keep track of the top-k items with the largest likelihood. The computational cost for a single user is $O(nd)$ (worst case) since we must compute $\mathbf{u}_i\mathbf{V}^\top$ where $\mathbf{u}_i \in \mathbb{R}^d$ and $\mathbf{V}^\top \in \mathbb{R}^{d\times n}$ and therefore dependent on the number of items n and

[8] Speed may be fundamentally more important than accuracy

inner iters. \ outer iterations	1	2	3	4	5
1	0.506	0.505	0.502	0.498	0.495
2	0.489	0.473	0.459	0.450	0.445
3	0.484	0.474	0.465	0.462	0.464
4	0.480	0.470	0.466	0.467	0.472
5	0.476	0.467	0.468	0.474	0.483

(a) PCMF MAE

inner iters. \ outer iterations	1	2	3	4	5
1	0.31	0.72	1.14	1.56	1.98
2	0.42	0.94	1.47	1.98	2.50
3	0.53	1.15	1.78	2.42	3.06
4	0.63	1.36	2.14	2.84	3.59
5	0.73	1.58	2.42	3.29	4.11

(b) PCMF time

inner iters. \ outer iterations	1	2	3	4	5
1	0.575	0.557	0.549	0.542	0.540
2	0.532	0.520	0.516	0.519	0.520
3	0.522	0.517	0.518	0.519	0.522
4	0.519	0.517	0.519	0.522	0.526
5	0.518	0.517	0.520	0.524	0.529

(c) PCMF-BASIC MAE

inner iters. \ outer iterations	1	2	3	4	5
1	0.24	0.54	0.77	1.12	1.37
2	0.34	0.59	0.82	1.31	1.60
3	0.35	0.60	0.99	1.30	1.77
4	0.36	0.64	1.08	1.50	1.96
5	0.37	0.67	1.14	1.62	2.11

(d) PCMF-BASIC time

Fig. 12 Comparing the effectiveness of inner and outer iterations. Maximum absolute error is used to evaluate the space of models learned when varying the outer and inner iterations. We also measure the runtime for each of the learned models. This demonstrates the effectiveness of relaxation methods that require only a few iterations while being sufficiently accurate. Note results above are from the epinions data using $d = 20$, $\alpha = 1$, and $\lambda = 0.1$.

latent dimensions d. Note items already purchased by a user are ignored.

The key intuition of the second method PCMF-NN is to leverage the users set of purchased items to limit the computations necessary while improving accuracy by examining a smaller refined set of items (finds a better candidate set). Given the next active user i from the stream $\mathcal{S}$ and the set of items purchased by that user denoted $\mathcal{P} \leftarrow N(i)$, we compute the set of users $N(\mathcal{P})$ who also purchased the items in $\mathcal{P}$. For each of the users in $N(\mathcal{P})$, we find the purchased items that were *not* purchased previously by user i (*i.e.*, not in $\mathcal{P}$) and use this set of items for prediction in $\mathbf{V}$ (similar to previous). Many variants of the above can also be exploited without increasing computational costs. For instance, instead of treating the items uniformly, we also

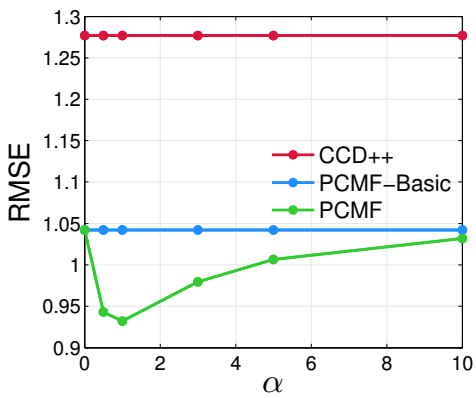


Fig. 13 Varying α (epinions 20% with $d = 10$).

α \ λ	0	0.1	0.5	1	5	100
0	1.16	1.04	1.07	1.33	4.7	4.74
0.1	3.4	2.1	0.95	1.01	2.5	4.71
0.5	3.5	0.94	0.98	1.13	2.4	4.64
1	2.6	0.93	1.02	1.21	3.8	4.69
5	1.23	1	1.09	1.35	4.69	4.71
100	1.2	1.04	1.08	1.34	4.69	4.71

Fig. 14 Exploring the impact of the parameter space on the predictive quality of the models

have a variant that weights them based on the frequency a user in $N(\mathcal{P})$ purchased it. This weight can then be used to bias the scores computed from $\mathbf{V}$. Note that if a user has no items, we default to computing a score for each item (*i.e.*, PCMF-NAÏVE).

Alternatively, we may use the social network information to restrict the items to score. For instance, given a user i, compute scores using only the items purchased by friends (social neighbors in $\mathbf{B}$)[9]. This approach is called PCMF-SOCIAL. Notice PCMF-SOCIAL obtains the set of users directly, avoiding the initial BFS step in PCMF-NN and therefore significantly more efficient (assuming similar number of users/friends). Hybrid approaches that use ideas from PCMF-NN and PCMF-SOCIAL are straightforward extensions.

Now we analyze these methods in a practical setting on a number of data sets. For this purpose, we measure the number of requests each method is capable of

[9] Undirected social networks give rise to variants based on in/out/total degree

processing per second. Intuitively, methods for prediction in real-time settings need to be fast and capable of handling hundreds and thousands of users on the fly with minimum wait time. Let us note that these requirements are often more important than accuracy (to an extent), especially since PCMF is significantly more accurate than other approaches. In Table 2 we analyze the *requests served per second* for the proposed methods across three different sets of data. Using the friends in the social network allowed us to process more requests per second. Two factors contribute to this improvement. First, PCMF-SOCIAL resulted in a smaller more refined set of products to score, and secondly similar users are given directly whereas PCMF-NN must compute this set using the purchased items (*i.e.*, an additional BFS step). Other optimization's such as caching, precomputing the dense user-user graph from $\mathbf{A}$, among others may lead to further improvements. Additionally, PCMF-SOCIAL directly leverages the notion of homophily [15,27,6] and therefore more relevant items are likely to be ranked higher.

Table 2 Evaluation of streaming prediction methods

	requests served per second				
data	PCMF-NAÏVE	PCMF-NN	PCMF-SOC	n	d
`Epinions`	1482	122	5468	755k	5
`MovieLens`	3783	398	N/A	65k	40
`Yelp`	4505	3347	N/A	11k	200

We also experimented with the dimensionality parameter and found PCMF-NAÏVE to be impacted the most as d increases, whereas PCMF-NN had less of an impact since finding the smaller set of items is independent of d. For instance, PCMF-NAÏVE in Table 2 processes a significantly larger number of requests per second than PCMF-NN for `MovieLens` with $d = 40$, whereas using a larger number of dimensions (*e.g.*, $d = 200$ for `Yelp`) brings the number of requests processed for the two methods much closer. Further, PCMF-SOCIAL is even more resilient since the set of items from friends in the social network were often much smaller than the set found in PCMF-NN. All methods scale extremely well as we increase the number of workers (processors and cores). This arises since requests are independent, yet the computations to handle a request may also be parallelized if necessary.

We have also explored various other models. In particular, how can we utilize the additional contextual/side information (such as the social network) for predicting the unknown test instances (*e.g.*, movie and item ratings)? Recall the proposed framework uses the additional information for learning where the latent factor matrices are allowed to influence one another based on the dependencies. Using the intuition that users and their social networks should have similar ratings (more so then a user chosen uniformly at random), we estimate the missing/unknown instances in the test set as,

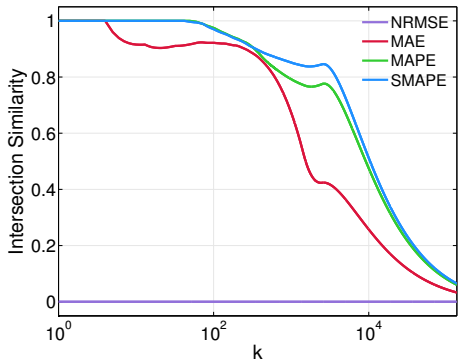


Fig. 15 Intersection similarity of rankings derived from PCMF for each of the evaluation metrics (compared to RMSE). Epinions is used in the above plot with PCMF. Given two vectors $\mathbf{x}$ and $\mathbf{y}$, the intersection similarity metric at k is the average symmetric difference over the top-j sets for each $j \leq k$. If $\mathcal{X}_k$ and $\mathcal{Y}_k$ are the top-k sets for $\mathbf{x}$ and $\mathbf{y}$, then $\text{isim}_k(\mathbf{x},\mathbf{y}) = \frac{1}{k}\sum_{j=1}^{k} \frac{|\mathcal{X}_j \Delta \mathcal{Y}_j|}{2j}$ where Δ is the symmetric set-difference operation. Identical vectors have an intersection similarity of 0. All values were first normalized between 0 and 1.

$$\mathbf{A}' = \big(\mathbf{I} + \beta\mathbf{B}\big)\mathbf{U}\mathbf{V}^\top$$

where $\mathbf{A}'$ is the reconstructed matrix, $\beta > 0$ is a parameter that controls the influence from $\mathbf{B}$, and

$$\sum_{(i,j)\Omega^{\mathbf{B}}} \mathbf{B}_{ij}\mathbf{u}_i^\top \mathbf{v}_j$$

is the weighted sum of predicted ratings for item j from the ith users social network.

4.8 Additional Applications

Additional applications of PCMF are summarized below.

4.8.1 Heterogeneous Link Prediction via Collective Factorization

The PCMF framework is also effective for predicting the existence of links in large-scale heterogeneous social networks. Existing work in social network link prediction has largely used only the past friendship graphs [16], whereas this work leverages PCMF to jointly model heterogeneous social network data. To evaluate the effectiveness of the heterogeneous link prediction approach, we use the LiveJournal data collected by Mislove *et al.*, see [22]. In particular, we use the social network

(user-friends-user) and the group membership bipartite graph (user-joined-group) which indicates the set of groups each user is involved. Note that PCMF may also be used to predict the groups a user is likely to join in the future (heterogeneous link prediction, link between multiple node types) as well as links in a homogeneous context such as friendships. PCMF is used to predict a set of held-out links[10] (*i.e.*, future/missing) and use the NRMSE evaluation metric for comparison. Overall, PCMF consistently resulted in significantly lower error than the baseline. In particular, we observed a 17.5% reduction in the error when PCMF is used instead of the base model that uses only the social network for link prediction. This reveals the importance of carefully leveraging multiple heterogeneous networks for link prediction. We also evaluated the ranking given by different evaluation metrics and found that some of them lead to significantly different rankings as shown in Figure 15. In that example, we measure the intersection similarity between RMSE and the others. In short, the ranking given by NRMSE is identical whereas the others are significantly different (even for users with the largest error).

4.8.2 Edge Role Discovery

Role discovery is becoming increasingly popular [8]. However, existing work focuses on discovering roles of nodes, and has ignored the task of discovering edge roles. In this work, we investigate edge-centric roles using a non-negative factorization variant of PCMF. Following the idea of feature-based roles proposed in [25], we systematically discover an edge-based feature representation. As initial features, we use a variety of edge-based graphlet features of size 2,3, and 4. From these initial features, more complicated features are discovered using the algorithm proposed in [25]. Given this large edge-by-feature matrix, PCMF is used to learn edge-role memberships. Importantly, PCMF provides a fast and parallel method for collective role discovery in large heterogeneous networks. Figure 16 demonstrates the effectiveness of PCMF by visualizing the edge roles learned from a biological network. The edge roles discovered by PCMF are clearly correlated with the class label of the node, and make sense as they capture the structural behavior surrounding each edge in the network.

4.8.3 Improving Relational Classification

PCMF may also be used to learn a more effective relational representation for a variety of relational learning tasks. In particular, Figure 17 shows the impact on the network structure when PCMF is used. Strikingly, PCMF creates edges between nodes of the same class, making them significantly closer compared to the original relational data (see Fig. 16).

[10] Note that these are known actual relationships in the social network, but are not used for learning.

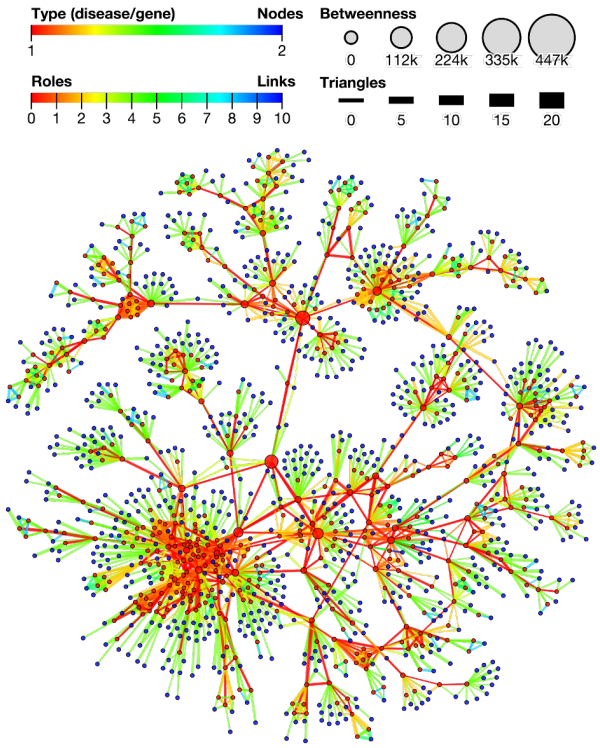


Fig. 16 Discovering edge roles. Edges are colored by the role with largest membership. We visualize the diseasome biological network. Node color indicates the class label (disease/gene).

5 Conclusion & Discussion

This paper proposed a fast parallel collective factorization framework for factorizing heterogeneous networks and demonstrated its utility on a variety of real-world applications. The method is efficient for large data with a time complexity that is linear in the total number of observations from the set of input matrices. Moreover, PCMF is flexible as many components are interchangeable (loss, regularization, etc), and for descriptive modeling tasks it is fully automatic (requiring no user-defined parameters) and data-driven/non-parametric and thus extremely useful in many real-world applications. Compared to recent state-of-the-art single matrix factorization methods, PCMF as well as our single-matrix variant PCMF-basic, are shown to be faster and more scalable for large data while also in many cases providing

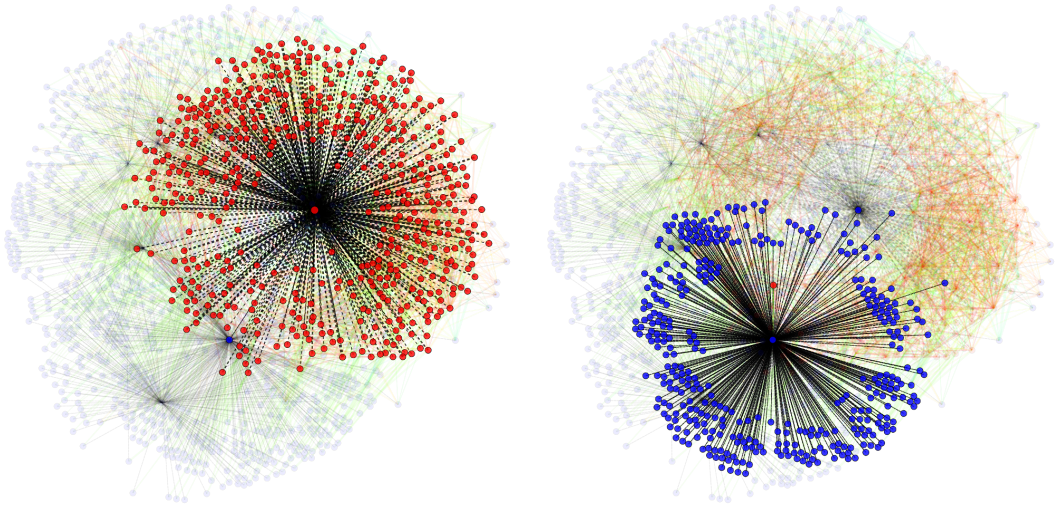


Fig. 17 PCMF improves relational/collective classification by automatically connecting up nodes of the same label. This not only benefits relational classification, but may significantly improve collective approaches that use label propagation by reducing noise and improving the quality of messages passed between such nodes. Nodes are colored by class label (disease/gene) using [4].

higher quality predictions. A main strength of PCMF lies in its generality as it naturally handles a large class of matrices (*i.e.*, contextual/side information), from sparse weighted single typed networks (*e.g.*, social friendship/-comm. networks, web graphs) and multi-typed networks (*e.g.*, user-group memberships, word-document matrix) to dense matrices representing node and edge attributes as well as dense similarity matrices.

References

1. C. C. Aggarwal. *Data streams: models and algorithms*, volume 31. Springer, 2007.
2. C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 81–92. VLDB Endowment, 2003.
3. N. K. Ahmed, J. Neville, and R. Kompella. Network sampling: From static to streaming graphs. *TKDD*, pages 1–54, 2013.
4. N. K. Ahmed and R. A. Rossi. Interactive visual graph analytics on the web. In *International AAAI Conference on Web and Social Media (ICWSM)*, pages 566–569, 2015.
5. H. Akaike. A new look at the statistical model identification. *Transactions on Automatic Control*, 19(6):716–723, 1974.
6. M. Bilgic, L. Mihalkova, and L. Getoor. Active learning for networked data. In *ICML*, pages 79–86, 2010.
7. P. Bonhard and M. Sasse. knowing me, knowing youusing profiles and social networking to improve recommender systems. *BT Technology Journal*, 24(3):84–98, 2006.
8. S. P. Borgatti, M. G. Everett, and J. C. Johnson. *Analyzing social networks*. SAGE Publications Limited, 2013.
9. E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
10. J. Fairbanks, D. Ediger, R. McColl, D. A. Bader, and E. Gilbert. A statistical framework for streaming graph analysis. In *ASONAM*, pages 341–347, 2013.
11. R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *SIGKDD*, pages 69–77, 2011.
12. M. Jamali and M. Ester. A matrix factorization technique with trust propagation for recommendation in social networks. In *RecSys*, pages 135–142, 2010.
13. D. Jiang, J. Pei, and H. Li. Mining search and browse logs for web search: A survey. *TIST*, 4(4):57, 2013.
14. Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.
15. T. La Fond and J. Neville. Randomization tests for distinguishing social influence and homophily effects. In *WWW*, pages 601–610, 2010.
16. D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *JASIST*, 58(7):1019–1031, 2007.
17. W. Liu, J. He, and S.-F. Chang. Large graph construction for scalable semi-supervised learning. In *Proceedings of the 27th International Conference on Machine Learning*, pages 679–686, 2010.
18. E. L. Lusk, S. C. Pieper, R. M. Butler, et al. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17(1), 2010.
19. H. Ma, H. Yang, M. R. Lyu, and I. King. Sorec: social recommendation using probabilistic matrix factorization. In *CIKM*, pages 931–940, 2008.
20. P. Massa and P. Avesani. Trust-aware recommender systems. In *Proceedings of the 2007 ACM conference on Recommender systems*, pages 17–24. ACM, 2007.
21. M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual review of sociology*, pages 415–444, 2001.
22. A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *SIGCOMM*, pages 29–42, 2007.

23. F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *NIPS*, 24:693–701, 2011.
24. B. Recht and C. Ré. Parallel stochastic gradient algorithms for large-scale matrix completion. *Mathematical Programming Computation*, 5(2):201–226, 2013.
25. R. A. Rossi and N. K. Ahmed. Role discovery in networks. *TKDE*, 26(7):1–20, 2014.
26. R. A. Rossi and N. K. Ahmed. An interactive data repository with visual analytics. *SIGKDD Explor.*, 17(2):37–41, 2016.
27. R. A. Rossi, L. K. McDowell, D. W. Aha, and J. Neville. Transforming graph data for statistical relational learning. *JAIR*, 45(1):363–441, 2012.
28. R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. In *NIPS*, volume 1, pages 2–1, 2007.
29. V. Satuluri, S. Parthasarathy, and Y. Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 international conference on Management of data*, pages 721–732. ACM, 2011.
30. P. Singla and M. Richardson. Yes, there is a correlation: from social networks to personal behavior on the web. In *WWW*, pages 655–664, 2008.
31. D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90. ACM, 2004.
32. Y. Sun and J. Han. *Mining Heterogeneous Information Networks: Principles and Methodologies*, volume 3. 2012.
33. J. Tang, X. Hu, and H. Liu. Social recommendation: a review. *SNAM*, 3(4):1113–1133, 2013.
34. M.-H. Tsai, C. Aggarwal, and T. Huang. Ranking in heterogeneous social media. In *WSDM*, pages 613–622, 2014.
35. M. Vorontsov, G. Carhart, and J. Ricklin. Adaptive phase-distortion correction based on parallel gradient-descent optimization. *Optics letters*, 22(12):907–909, 1997.
36. S.-H. Yang, B. Long, A. Smola, N. Sadagopan, Z. Zheng, and H. Zha. Like like alike: joint friendship and interest propagation in social networks. In *WWW*, pages 537–546, 2011.
37. X. Yang, Y. Guo, Y. Liu, and H. Steck. A survey of collaborative filtering based social recommender systems. *Computer Communications*, 2013.
38. Y. Yasui, K. Fujisawa, and K. Goto. Numa-optimized parallel breadth-first search on multicore single-node system. In *Big Data*, pages 394–402, 2013.
39. H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, pages 765–774, 2012.
40. Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.
41. M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4, 2010.