A Distributed Algorithm for Large-Scale Generalized Matching

Faraz Makari Manshadi
Max-Planck-Institut für Informatik
fmakari@mpi-inf.mpg.de

Baruch Awerbuch
Johns Hopkins University
baruch@cs.jhu.edu

Rainer Gemulla
Max-Planck-Institut für Informatik
rgemulla@mpi-inf.mpg.de

Rohit Khandekar*
Knight Capital Group
rkhandekar@gmail.com

Julián Mestre
School of IT, The University of Sydney
mestre@it.usyd.edu.au

Mauro Sozio†
Institut Mines-Telecom, Telecom ParisTech, CNRS
mauro.sozio@telecom-paristech.fr

ABSTRACT

Generalized matching problems arise in a number of applications, including computational advertising, recommender systems, and trade markets. Consider, for example, the problem of recommending multimedia items (e.g., DVDs) to users such that (1) users are recommended items that they are likely to be interested in, (2) every user gets neither too few nor too many recommendations, and (3) only items available in stock are recommended to users. State-of-the-art matching algorithms fail at coping with large real-world instances, which may involve millions of users and items. We propose the first distributed algorithm for computing near-optimal solutions to large-scale generalized matching problems like the one above. Our algorithm is designed to run on a small cluster of commodity nodes (or in a MapReduce environment), has strong approximation guarantees, and requires only a poly-logarithmic number of passes over the input. In particular, we propose a novel distributed algorithm to approximately solve mixed packing-covering linear programs, which include but are not limited to generalized matching problems. Experiments on real-world and synthetic data suggest that a practical variant of our algorithm scales to very large problem sizes and can be orders of magnitude faster than alternative approaches.

1. INTRODUCTION

Matching problems arise in a number of applications, including trade markets [24], computational advertising [6, 5], and semi-supervised learning [15]. Consider, for example, the problem of assigning DVDs to customers in online DVD rental. Online video stores such as Netflix or Amazon's LOVEFiLM allow customers to specify which DVDs they would like to rent. Since the number of physically available DVDs is limited, customers are encouraged to provide a large, ranked list of preferred movies; the online video store then automatically selects DVDs to ship to customers based on both preferences and availability. Moreover, customers are recommended movies that they might be interested in; a good recommendation engine should consider availability to minimize customer waiting times for accepted recommendations. The problem of assigning DVDs to customers is non-trivial: for shipping, we want to make sure that users are sent movies ranked as high on their lists as possible, while at the same time maintaining fairness, i.e., every user should be supplied with a sufficient number of DVDs without exceeding the user's DVD budget. Similarly, for recommendation, we want to recommend a few DVDs to each user such that the user is likely to be interested in his recommendations and, in case of acceptance, the recommended DVD is (likely to be) physically available.

We can model problems like the ones above as *generalized bipartite matching* problems. In such a problem, we are given a set of users and a set of items. There is an edge between a user u and an item v if u is interested in v; each edge is associated with a positive weight that captures the degree of interest. Furthermore, each user u and each item v is associated with a lower and an upper bound on the number of edges that can participate in the matching (the matching problem is "generalized" due to the presence of these bounds). Our goal is to find a matching—i.e., subset of the edges—such that all constraints are satisfied and the total weight of the edges in the matching is maximized. This problem is also known in the literature as the *maximum weight degree-constrained subgraph* problem, while the special case with only upper bounds is known as *maximum weight b-matching*.

The problems above can be solved in polynomial time via traditional max-flow techniques [1], linear programming solvers such as Gurobi [12], or using the combinatorial algorithm developed in [8]. Unfortunately, these approaches cannot cope with the massive scale of real-world problem instances which may involve millions of users, items, and edges; for instance, Netflix offers tens of thousands of movies for rental to more than 20M customers.

In this paper, we propose a scalable distributed approximation algorithm for large-scale generalized bipartite matching. Though several scalable algorithms for b-matching have been proposed in the literature [13, 16, 22], our algorithm is the first distributed (i.e., shared nothing) algorithm for generalized bipartite matching

*Part of this work was done while the author was at IBM T. J. Watson Research Center, USA.

†Part of this work was done while the author was visiting the IBM Almaden Research Center, USA.

Proceedings of the VLDB Endowment, Vol. 6, No. 9

problems, and it can cope with massive real-world instances. Our algorithm can also be adapted to capture more complex matching problems with additional constraints; e.g., in the case of DVD recommendation, one may enforce that users are recommended at most one movie per genre to encourage diversity.

Our algorithm produces an approximate solution to the matching program with strong approximation guarantees. Given a (small) error bound ϵ, we compute a solution that is within a factor of $1-\epsilon$ of the optimal solution in expectation, and satisfies the lower- and upper-bound constraints within a factor of $(1-\epsilon)$ and $(1+\epsilon)$, respectively. Our method is based on linear programming (LP) and consists of two phases: (1) Compute an approximate fractional solution to the LP relaxation of the integer linear program corresponding to the matching problem, and (2) round the fractional solution to obtain an integral solution.

Our distributed LP algorithm, called *MPCSolver*, can compute an approximate solution to any mixed packing-covering LP—i.e., an LP in which all coefficients and variables are non-negative—in a poly-logarithmic number of passes over the data. MPCSolver is inspired by the algorithms of [3], which can solve either packing LPs or covering LPs, and is simple and efficient in practice. Once an approximate solution to the LP has been obtained, we combine the centralized rounding scheme of [9] with recent work on "filtering" in the MapReduce setting [18] to obtain an efficient distributed rounding algorithm called DDRounding.

Both our LP and our rounding algorithm are amenable to MapReduce. In this paper, however, we focus on a general shared-nothing architecture for improved efficiency. In fact, it has been observed that MapReduce can be inefficient for the kind of iterative computations performed by our algorithms [19]. We describe implementation issues that are key to good performance in practice, e.g,. how to distribute data effectively across nodes so that communication costs are minimized. Finally, we experimentally compare the efficiency and scalability of our algorithms to existing alternatives on both real-world and synthetic datasets of varying sizes. Our experiments indicate that our algorithms can handle significantly larger problem instances than state-of-the-art linear programming solvers, and are orders of magnitude faster than alternative distributed algorithms.

The remainder of this paper is organized as follows: We formally define the generalized bipartite matching problem in Sec. 2. In Sec. 3, we propose distributed algorithms for solving mixed packing-covering LPs and rounding the solution. We discuss related work in Sec. 4 and give results of our experimental study in Sec. 5. We conclude the paper in Sec. 6.

2. PROBLEM DEFINITION

We are given an undirected bipartite graph $G=(U,V,E)$, where U represents a set of users, V represents a set of items, and there is an edge $(u,v)\in E$ if user u is interested in item v. Each edge (u,v) is associated with a *positive weight* $w(u,v)$ measuring the degree of interest of u in v. For each user and item, we are additionally given a *lower bound* $l(v)$ and an *upper bound* $b(v)$, where $v\in U\cup V$. The bounds constrain the degree of vertex v in the solution; e.g., bounds on the number of recommendations for a user or the availability of a movie. More formally, denote by $\bar{E}\subseteq E$ a subset of the edges in G, and by $\bar{E}_v$ the set of edges incident to vertex v in subgraph $(U,V,\bar{E})$. We say that $\bar{E}$ is *feasible* if $l(v)\le|\bar{E}_v|\le b(v)$ for all $v\in U\cup V$, i.e., all lower and upper bound constraints are met. An instance of our problem is feasible if there exists a feasible $\bar{E}$. The *generalized bipartite matching* (GBM) problem is to determine the best feasible matching between users and movies, i.e., we seek to maximize the *objective function* $f(\bar{E})=\sum_{(u,v)\in\bar{E}}w(u,v)$ over all feasible $\bar{E}\subseteq E$.

Although GBM can be solved in polynomial time via maximum-flow techniques or integer linear programs, available solvers do not scale to the large problem instances that occur in practice, i.e., instances with millions of users and movies, and potentially billions of edges. In this paper, we consider an approximate variant of GBM in which we seek for a "good" (but not necessarily optimal) solution and additionally allow for a small violation of lower- and upper-bound constraints. In particular, denote by $\epsilon>0$ a small *error bound*. We say that $\bar{E}$ is *ϵ-feasible* if lower- and upper-bound constraints are violated by at most a factor of $1-\epsilon$ and $1+\epsilon$ (up to rounding), respectively. If the GBM is feasible and objective $f(\bar{E})$ is within a factor of $1-\epsilon$ of the optimal solution OPT to GBM, we say that $\bar{E}$ is a *$(1-\epsilon)$-approximation* of the GBM. We refer to this relaxed problem as GBM$_\epsilon$.

PROBLEM 1 (GBM$_\epsilon$). *We are given $\epsilon>0$ and a GBM in terms of an undirected bipartite graph $G=(U,V,E)$, a weight function $w: E\to\Re^+$, and lower- and upper-bound functions $l: U\cup V\to\mathbb{N}$ and $b: U\cup V\to\mathbb{N}$, respectively. If the GBM is feasible with optimum objective* OPT*, find a subset $\bar{E}\subseteq E$ such that*

$$\lfloor(1-\epsilon)l(v)\rfloor\le|\bar{E}_v|\le\lceil(1+\epsilon)b(v)\rceil$$

for each $v\in U\cup V$, and

$$\sum_{(u,v)\in\bar{E}}w(u,v)\ge(1-\epsilon)\mathrm{OPT}.$$

If the GBM is infeasible, return any edge set $\bar{E}\subseteq E$.

In Sec. 3, we develop a randomized algorithm to efficiently solve large problem instances of GBM$_\epsilon$ in a distributed environment. Our algorithm guarantees ϵ-feasibility and produces a solution with objective of (at least) $(1-\epsilon)$OPT in expectation. Although we do not make any formal claims here, our experiment suggests that the objective is also concentrated around the expected value, i.e., we obtained an objective that was close or better than $(1-\epsilon)$OPT in every single run of our algorithm.

2.1 ILP formulation and LP relaxation

The GBM problem can be formulated as an integer linear program (ILP) as follows. For each edge $e\in E$, we introduce a binary variable $x_e\in\{0,1\}$; $x_e=1$ if e is included in the solution, otherwise $x_e=0$. The ILP is then given by

$$\begin{array}{lll}\max & \sum_{e\in E}w(e)x_e & \\ \text{s.t.} & \sum_{e\in E_v}x_e\le b(v) & \forall v\in U\cup V,\\ & \sum_{e\in E_v}x_e\ge l(v) & \forall v\in U\cup V,\\ & x_e\in\{0,1\} & \forall e\in E,\end{array}\qquad\text{(GBM-ILP)}$$

where E_v denotes the set of edges adjacent to vertex v in G. Here the first two constraints express the upper- and lower-bound constraints, respectively. Since the constraint matrix of GBM-ILP (see MPC-LP below) is totally unimodular and the right-hand sides are all integer-valued, an optimum solution can be computed in polynomial time [1]. However, our experiments in Sec. 5 suggest that state-of-the-art ILP solvers cannot handle very large problem instances.

We obtain the so-called *LP relaxation*, denoted GBM-LP, by allowing x_e to be fractional, i.e., to take any value in $[0,1]$. As described in Sec. 3, we make use of the LP relaxation in our algorithms in that we first compute a solution to GBM$_\epsilon$-LP, and then round (sensibly) to obtain an integral solution. GBM-LP belongs to the class of *mixed packing-covering LPs*,[1] which have general form:

$$\begin{array}{lll}\max & w^\top x & \\ \text{s.t.} & \boldsymbol{P}x\le p & \\ & \boldsymbol{C}x\ge c & \\ & x\ge 0, & \end{array}\qquad\text{(MPC-LP)}$$

[1]The mapping of GBM-LP to MPC-LP is described in Sec. 3.2.

where $x \in \Re_+^n$ denotes a vector of variables, $w \in \Re_+^n$ a vector of weights, $\boldsymbol{P} \in \Re_+^{m \times n}$ a *packing-constraint matrix* with right-hand side $p \in \Re_+^m$, and $\boldsymbol{C} \in \Re_+^{k \times n}$ a *covering-constraint matrix* with right-hand side $c \in \Re_+^k$. MPC-LPs constitute a "simple" (but still expressive) subclass of LPs.

3. ALGORITHMS

As mentioned in Sec. 2.1, we solve GBM$_\epsilon$ by computing a solution to the corresponding LP relaxation, which is subsequently rounded to an integral solution. We first present a novel distributed algorithm that can compute an ϵ-feasible solution to any MPC-LP (Secs. 3.1 and 3.2) and show how the special structure of GBM can be exploited to reduce communication costs (Sec. 3.3). We then show how our algorithm can be used to achieve a $(1-\epsilon)$-approximation to the MPC-LP (Sec. 3.4). Finally, we combine ideas from randomized rounding [9] with "filtering" techniques for MapReduce [18] to obtain an integral solution (Sec. 3.5).

3.1 Solving MPC-LP (feasibility)

We develop a distributed algorithm for approximately solving general MPC-LP problems. Our algorithm is inspired by, but more general than, the work of [3], which can handle either packing or covering constraints, but not both. In this section, we consider only the feasibility version of the problem. To simplify our discussion, we describe our algorithm in a centralized setting and generalize to the distributed setting in Sec. 3.2.

Recall the definition of an MPC-LP given above. Without loss of generality, we assume that $p = \mathbb{1}$ and $c = \mathbb{1}$, where $\mathbb{1}$ denotes an all-one vector of the appropriate dimensionality, and that each of the non-zero entries in $\boldsymbol{P}$ and $\boldsymbol{C}$ is equal to or larger than 1.[2] Denote by M the largest entry in $\boldsymbol{P}$ and $\boldsymbol{C}$. We aim to find a vector x such that

$$\begin{array}{rcl} \boldsymbol{P}x & \leq & \mathbb{1} \\ \boldsymbol{C}x & \geq & \mathbb{1} \\ x & \geq & 0. \end{array} \tag{1}$$

Denote by $\boldsymbol{A}_i$ the i-th row of any matrix $\boldsymbol{A}$, by $\boldsymbol{A}_j^\top$ the j-th row of $\boldsymbol{A}^\top$, and by $\boldsymbol{A}_{ij}$ the (i,j)-entry of $\boldsymbol{A}$. For any value of $x \in \Re_+^n$, let $y(x) = (y_1, \ldots, y_m)^\top \in \Re_+^m$ and $z(x) = (z_1, \ldots, z_k) \in \Re_+^k$ be given as follows

$$y_i(x) = \exp\left[\mu \cdot \left(\boldsymbol{P}_i x - 1\right)\right] \tag{2}$$

$$z_i(x) = \exp\left[\mu \cdot \left(1 - \boldsymbol{C}_i x\right)\right], \tag{3}$$

where μ is a scaling factor (defined in Alg. 1) and $y_i(x)$ and $z_i(x)$ denote the i-th entry of $y(x)$ and $z(x)$, respectively. For brevity, we suppress the dependence of $y(x)$ and $z(x)$ on x when the value of x is clear from context. One may think of y_i as an exponential "penalty" function of packing constraint $\boldsymbol{P}_i x \leq 1$. Penalty y_i is small if the constraint is satisfied ($y_i \leq 1$ if $\boldsymbol{P}_i x \leq 1$) and large otherwise ($y_i > 1$ if $\boldsymbol{P}_i x > 1$). Similarly, z_i is a penalty function for covering constraint $\boldsymbol{C}_i x \geq 1$. In what follows, we refer to y_i and z_i as the *dual variable* of the corresponding constraint. Our algorithm tries to minimize the overall penalty, i.e., the *potential function*

$$\Phi(x) = \sum_{i=1}^{m} y_i(x) + \sum_{i=1}^{k} z_i(x).$$

The scaling constant μ is chosen sufficiently large so that if Φ is (approximately) minimized, the corresponding solution is ϵ-feasible whenever the MPC-LP is feasible. Since Φ is differentiable and convex in x, we use a version of gradient descent to find the optimal solution (i.e., the one with lowest penalty). Consider the partial derivative of Φ w.r.t. x_j:

$$\frac{\partial \Phi}{\partial x_j} = \mu \boldsymbol{P}_j^\top y - \mu \boldsymbol{C}_j^\top z.$$

At the minimum, all partial derivatives are zero. When the derivative is negative (positive), we will increase (decrease) x_j by a carefully chosen amount.

[2]This can be achieved by first rescaling the rows, then the columns of the coefficient matrices. Note that column scaling will change the solution; we can recover the solution to the original problem by "inversely" scaling the result of the modified problem.

Algorithm 1 MPCSolver for mixed packing-covering LPs

Require: packing constraint $\boldsymbol{P}$, covering constraints $\boldsymbol{C}$, error bound ϵ

$\epsilon' = \epsilon/10$ *// internal error bound*
$\mu \leftarrow \ln(mkM/\epsilon')/\epsilon'$ *// scaling constant*
$\alpha \leftarrow \epsilon'/4$ *// update threshold*
$\beta \leftarrow \alpha/(20\mu)$ *// multiplicative step size*
$\delta \leftarrow \beta/nM$ *// additive increase*
Start with any $x \in \Re_+^n$
repeat
  Compute $y_i(x) = \exp\left[\mu \cdot \left(\boldsymbol{P}_i x - 1\right)\right]$ for $i = 1, \ldots, m$
  Compute $z_i(x) = \exp\left[\mu \cdot \left(1 - \boldsymbol{C}_i x\right)\right]$ for $i = 1, \ldots, k$
  for $j = 1, \ldots, n$ **do**
    if $\frac{\boldsymbol{P}_j^\top y(x)}{\boldsymbol{C}_j^\top z(x)} \leq 1 - \alpha$ **then**
      $x_j \leftarrow \max\{x_j(1+\beta), \delta\}$
    if $\frac{\boldsymbol{P}_j^\top y(x)}{\boldsymbol{C}_j^\top z(x)} \geq 1 + \alpha$ **then**
      $x_j \leftarrow x_j(1-\beta)$
until convergence (Sec. 3.3)

Our algorithm is given in Alg. 1; we refer to this algorithm as MPCSolver. Here parameter ϵ' is an internal error bound, α acts as an *update threshold*, β as a multiplicate *step size*, and δ as an *additive increase*; each parameter is chosen carefully and depends on error bound ϵ. The algorithm starts with an arbitrary initial point $x_0 \in \Re^+$. In each *round* (i.e., each iteration of the repeat-until loop), we first compute the values of the dual variables y and z. We update variable x_j only if its partial derivative is sufficiently far away from zero. The algorithm terminates once all variables are left unmodified (or one of the alternative convergence tests of Sec. 3.3 applies); we then obtain an approximate minimizer of Φ. In particular, we update x_j if and only if ratio $\boldsymbol{P}_j^\top y(x)/\boldsymbol{C}_j^\top z(x)$ lies outside $(1-\alpha, 1+\alpha)$; thus α acts as an update condition. If the ratio exceeds $1+\alpha$ (positive gradient), we decrease x_j; if the ratio is below $1-\alpha$ (negative gradient), we increase x_j. Step size parameter β determines how quickly we move through the parameter space. Updates are multiplicative: We add or subtract βx_j from x_j; thus the larger x_j, the more it is changed. Finally, we ensure that $x_j \geq \delta$ after an increase so that we can quickly move away from 0 when x_j is very small. Note that our algorithm can be implemented in a few lines of code.

MPCSolver is designed such that it converges quickly to an ϵ-feasible solution. Our main theoretical result is as follows:

THEOREM 1 (MAIN). *For any $0 < \epsilon \leq 0.5$, Alg. 1 terminates after $\widetilde{O}(\epsilon^{-5} \ln^3(kmMnx_{max}))$ rounds, where $x_{max} = \max\{\delta, x_{0,1}, \ldots, x_{0,n}\}$ and $x_{0,j}$ denotes the j-th element of starting point x_0 (note that $\delta \ll \epsilon$). If the given MPC-LP instance is feasible, Alg. 1 produces an ϵ-feasible solution. For $\epsilon > 0.5$, all properties are retained w.r.t. $O(\epsilon)$.*

The theorem asserts that the number of rounds required by MPCSolver is poly-logarithmic in the input, which ensures fast conver-

gence. Here we used $\widetilde{O}$-notation to hide lower-order terms; a more precise bound is given by $O(\epsilon^{-5}\ln^3(\epsilon^{-1})\ln^2(kmM)\ln(k\ln(m)Mnx_{max}))$. Note that MPCSolver uses an internal error bound ϵ', which is set to $\epsilon/10$ in Alg. 1. For sufficiently small $\epsilon \leq 0.5$, this ensures convergence to an ϵ-feasible solution. In Sec. 3.3, we show how to dynamically adapt ϵ' to improve performance in practice.

PROOF SKETCH. We give an outline of the proof here; the full proof appears in App. A. Assume for now that the MPC-LP is feasible. We start by partitioning the rounds of MPCSolver into a set of disjoint *intervals*, each consisting of a sequence of consecutive rounds. The first interval is a "warm-up" interval consisting of a poly-logarithmic number τ_0 of rounds. We show that after τ_0 rounds, the potential is sufficiently small, i.e., at most $\Phi_{\text{init}} = m\exp(\mu(1+2\epsilon)) + k\exp(\mu)$ (implied by Lemma 3). Moreover, we prove that after the warm-up interval, the potential Φ is monotonically non-increasing, and that its decrease per round can be bounded from below (Lemma 5). We then divide the remaining intervals into *stationary* and *unstationary* intervals of length $\tau_1 = O(\beta^{-1}\log\delta^{-1})$. An interval is stationary if the potential does not change significantly; as soon as we see a stationary interval, we show that the solution is $\theta(\epsilon)$-feasible (ϵ-feasible for $\epsilon \leq 0.5$, Lemma 7). All other intervals are unstationary; in these intervals, the potential decreases by a factor of at least $\Omega(\epsilon^2)$ (Lemma 6). With our choice of μ, Alg. 1 terminates when the potential drops below $\Phi_{\text{fin}} = m + k$. We conclude that there are at most $O(\log(\Phi_{\text{init}}/\Phi_{\text{fin}})/\epsilon^2)$ unstationary intervals, where $\Phi_{\text{init}}/\Phi_{\text{fin}} = O(\exp(\mu))$. Putting all pieces together, x becomes $\theta(\epsilon)$-feasible after $\tau = \tau_0 + O(\tau_1/\epsilon^2\log(\Phi_{\text{init}}/\Phi_{\text{fin}}))$ rounds. Finally, to handle infeasible problem instances, we always terminate Alg. 1 after τ rounds, whether or not a solution has been found. ■

3.2 Distributing MPCSolver

We focus on in-memory processing in the shared-nothing setting, and exploit shared-memory when multiple threads are run on a compute node. Denote by s the number of compute nodes and by t the number cores (each running a thread) per node; the total number of cores in the cluster is thus given by $T = st$.

General case. We first describe how to distribute MPCSolver for general MPC-LPs, and then specialize to GBM-LPs. Recall from Sec. 3.1 that we associate dual variables y and z with each packing or covering constraint, respectively. In each round, MPCSolver first computes the values of these dual variables (lines 8 and 9). To distribute this computation, we conformingly partition $x^\top$ and matrices $\boldsymbol{P}$ and $\boldsymbol{C}$ column-wise across the s nodes; thus each partition corresponds to a subset of the variables x. Denote by $\boldsymbol{P}_{(i)}$, $\boldsymbol{C}_{(i)}$, and $x_{(i)}$ the respective partitions stored at node i, $1 \leq i \leq s$. Each node i computes the products $\boldsymbol{P}_{(i)}x_{(i)}$ and $\boldsymbol{C}_{(i)}x_{(i)}$ independently and in parallel (using t threads); the local results are subsequently aggregated (i.e., summed up) and broadcasted across the compute cluster to obtain $\boldsymbol{P}x$ and $\boldsymbol{C}x$ (e.g., using `MPI_Allreduce`). Each node then computes the values of the duals corresponding to its variables and broadcasts the results to obtain y and z. Once the duals have been computed, MPCSolver updates the primal variables x (lines 10–14). To do this in a distributed manner, we can reuse the same partitioning of the constraints matrices; primal variables in partition $x_{(i)}$ can be updated independently and in parallel. Since both constraint matrices are accessed once row-wise (to compute the dual variables) and once column-wise (to update the primal variables), we store at each node i two copies of $\boldsymbol{P}_{(i)}$ and $\boldsymbol{C}_{(i)}$ in memory: one in row-major and one in column-major order. Note that the constraint matrices are partitioned once before the algorithm starts, and that only the values corresponding to the dual variables are communicated. The algorithm just described can be inefficient in terms of communication costs when there are many more duals (constraints) than primal variables; however, communication costs can be reduced significantly by exploiting the sparsity of the constraint matrices.

MPCSolver for GBM-LP. Recall the definition of an MPC-LP from Sec. 2.1 and consider the LP-relaxation GBM-LP of a given GBM instance. Then $n = |E|$ denotes the number of edges, $x = (x_1, \ldots, x_n)$ the edge variables, and w the corresponding edge weights. Set $r = |U \cup V|$ and denote by $\boldsymbol{M}$ the $r \times n$ incidence matrix of bipartite graph G. Then $\boldsymbol{C}_{r\times n} = \boldsymbol{M}$ (i.e., $k = r$), c is the vector of lower bounds, $\boldsymbol{P}_{(r+n)\times n} = (\boldsymbol{M}^\top\ \boldsymbol{I}_n)^\top$ (i.e., $m = r + n$), where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix, and p a vector of r upper bounds followed by n ones. Thus $\boldsymbol{P}$ handles both the upper-bound constraints and the constraint that $x_j \leq 1$ for all j. Note that matrices $\boldsymbol{C}$ and $\boldsymbol{P}$ are usually sparse.

Denote by $y_{\boldsymbol{M}}$ and $y_{\boldsymbol{I}}$ the subset of the dual variables corresponding to upper-bound constraints and at-most-one constraints, respectively. We communicate $y_{\boldsymbol{M}}$ and z in every round but keep $y_{\boldsymbol{I}}$ local (only local variables affected). This is advantageous since the size of $y_{\boldsymbol{I}}$ is linear in n (large) but the size of $y_{\boldsymbol{M}}$ and z is linear in r (small). For GBM-LP, only product $\boldsymbol{M}x$ is needed for the computation of both $y_{\boldsymbol{M}}$ and z; we compute this product only once (as described above). Note that all communicated values are vectors of length r, which makes MPCSolver especially attractive for GBM-LP.

3.3 Implementing MPCSolver

We provide some optimizations that we used to speed-up the performance of MPCSolver in practice.

Starting point. MPCSolver can start from any initial point $x \in \Re^n_+$. However, in order to reduce the number of iterations and improve the efficiency of MPCSolver for GBM-LP, we make use of a special initialization, which satisfies all packing constraints as follows. Denote by $\text{nnz}(j)$ the largest number of non-zero entries in any row i of $\boldsymbol{M}$ in which $\boldsymbol{M}_{ij} \neq 0$. We then set

$$x_j = \frac{1}{\text{nnz}(j)\cdot\max_i \boldsymbol{M}_{ij}} \text{ for } j \in [1,n].$$

Adaptive error bounds. The internal error bound parameter ϵ' controls the quality of the final solution, but it also dramatically affects the number of rounds to convergence. Our experiments suggest that setting ϵ' to values larger than $\epsilon/10$ has only a mild impact on the quality of the final solution but leads to faster convergence; in fact, our analysis of MPCSolver is somewhat loose so that our choice of $\epsilon' = \epsilon/10$ is conservative. We devise a simple adaptive method for choosing ϵ', which greatly improves the running time while ensuring ϵ-feasibility. We exploit the fact that the potential function can be efficiently evaluated at the end of each iteration and proceed as follows. Let $\epsilon \leq 0.5$. We first set ϵ' to some value larger than $\epsilon/10$ (e.g., 2). We keep running with value ϵ' as long as the potential improves significantly (e.g., by 0.001%). Whenever the potential stagnates, we decrease ϵ' (e.g., by 1%)—but not below $\epsilon/10$—and update parameters μ, α, β, and δ accordingly. This simple adaptive scheme worked well in our experiments (see Sec. 5).

Convergence test. MPCSolver has converged as soon as one of the following criteria is met: (1) the solution is ϵ-feasible, (2) all variables remain unmodified, or (3) the number of rounds exceeds the poly-logarithmic bound of Theorem 1. If (2) or (3) hold and the solution is not ϵ-feasible, the MPC-LP is infeasible. In practice, the poly-logarithmic bound is usually too large to be useful (in particular when ϵ is small). A heuristic convergence test is to stop MPCSolver when the decrease in potential stagnates; e.g., when it falls below some threshold (say, 0.001%) in two consecutive rounds. For the adaptive scheme, we apply the heuristic convergence test only when ϵ' has been reduced to $\epsilon/10$. Even though the guarantees of The-

orem 1 do not hold when this heuristic is used, our experiments suggest that the test is effective in practice.

3.4 From feasibility to optimization

Up to this point, we have described how to use MPCSolver to obtain an ϵ-feasible solution to MPC-LP. In this section, we show how to derive an $(1-\epsilon)$-approximate solution, i.e., we also optimize the objective function. Our techniques are an adaptation of the techniques of Young [28] to our setting. The key idea of [28] is to push the objective into the constraints. In particular, we consider the following optimization problem: find $\lambda^* = \max\{\lambda : (\exists x)\boldsymbol{P}x \leq b, \boldsymbol{C}x \geq d, w^\top x \geq \lambda\}$. Given the value of λ^*, we can determine x via solving a feasibility problem with the additional covering constraint $w^\top x \geq \lambda^*$. To obtain λ^*, we run a sequence of feasibility problems of form $\{Px \leq p, \boldsymbol{C}x \geq c, w^\top x \geq \lambda\}$, where λ is determined via binary search. Our search strategy is close to the one of [28], but we use a fixed error bound and directly compute an $(1-\epsilon)$-approximation. The following lemma allows us to select an initial value for λ.

LEMMA 1. *Let $\lambda_{min} = \min\{w^\top x : \boldsymbol{C}x \geq c, x \geq 0\}$ and $\lambda_{max} = \max\{w^\top x : \boldsymbol{P}x \leq p, x \geq 0\}$. Let $\bar{x}$ be any feasible solution to MPC-LP, and set $\lambda = w^\top \bar{x}$. Then $(\lambda_{min}/\lambda_{max})\lambda^* \leq \lambda \leq \lambda^*$.*

PROOF. By definition, $\lambda \leq \lambda^*$. Consider the minimization version of MPC-LP that includes only the covering constraints; we have $w^\top x \geq \lambda_{\min}$. Since we ignore packing constraints, $\lambda_{\min}$ is a lower-bound on the objective value of any feasible solution to MPC-LP, i.e., $\lambda \geq \lambda_{\min}$. Using similar arguments, we obtain $\lambda^* \leq \lambda_{\max}$ and the assertion follows. ∎

We use the distributed algorithm of Awerbuch and Khandekar (AK [3]), which obtains a $(1+\epsilon)$-approximation for covering problems ($\hat{\lambda}_{\min}$) and a $(1-\epsilon)$-approximation for packing problems ($\hat{\lambda}_{\max}$). Since MPCSolver is a generalization of AK to MPC, we use our existing implementation and data partitioning; only parameters, update condition, and update rules need to be changed.

Let $\rho = (\hat{\lambda}_{\min}/(1+\epsilon)) / (\hat{\lambda}_{\max}/(1-\epsilon))$ and assume that the MPC-LP is feasible so that MPCSolver is able to obtain an ϵ-feasible solution. As before, denote by λ the objective realized by any such solution. Our binary search algorithm proceeds as follows: We start with $\lambda_0 = \lambda/\rho$, then $\rho \leq \lambda^*/\lambda_0 \leq 1$. We then aim to find the integer l^* such that $(1-\epsilon)^{l^*+1} < \lambda^*/\lambda_0 \leq (1-\epsilon)^{l^*}$ via binary search (on l). Given any l, we run MPCSolver with $\lambda = (1-\epsilon)^l\lambda_0$ as lower bound on the objective. If we obtain an ϵ-feasible solution, then $\lambda^* \geq (1-\epsilon)\lambda \geq (1-\epsilon)^{l+1}\lambda_0$ so that $l^* \leq l$. Otherwise, the problem is infeasible and $l^* > l$. The range used for binary search is given by $0 \leq l \leq \log_{1-\epsilon}\rho$ (since l^* must fall into this range). The solution obtained at $l = l^*$ is now a $(1-\epsilon)$-approximation to the MPC-LP.

LEMMA 2. *Our binary search algorithm computes an ϵ-feasible $(1-\epsilon)$-approximation to any feasible MPC-LP by solving at most $\log_2 \log_{1-\epsilon}(\lambda_{min}/\lambda_{max})$ ϵ-feasibility problems.*

In practice, we need to solve only few ϵ-feasibility problems (up to 7 in our experiments).

3.5 Obtaining an integral solution

We propose a distributed randomized rounding algorithm that takes as input a fractional solution x of GBM$_\epsilon$-LP (such as the one obtained by MPCSolver) and produces an ϵ-feasible integral solution X such that $\mathrm{E}[w^\top X] = w^\top x$. In particular, our algorithm ensures that: (1) $\mathrm{E}[w^\top X] = w^\top x$, (2) all edge variables $X_e \in \{0,1\}$, (3) when $\boldsymbol{P}_i x \leq (1+\epsilon)p_i$, then $\boldsymbol{P}_i X \leq \lceil(1+\epsilon)p_i\rceil$, and (4) when $\boldsymbol{C}_i x \geq (1-\epsilon)c_i$, then $\boldsymbol{C}_i X \geq \lfloor(1-\epsilon)c_i\rfloor$.

A naïve approach is to use an independent rounding scheme, which independently rounds every edge variable such that $P(X_e = 1) = x_e$. Such a scheme, however, may not (and often does not) lead to an ϵ-feasible solution. To see this, consider the (packing) constraint $X_1 + X_2 + X_3 \leq \lceil 1+\epsilon\rceil$ and the fractional solution $x_1 = 0.2, x_2 = 0.3, x_3 = 0.5+\epsilon$ for $\epsilon < 1$; independent rounding sets $X_1 = X_2 = X_3 = 1$ with non-zero probability but this solution is not ϵ-feasible. We thus need some form of dependent rounding, in which variables are rounded dependent on the rounding of other variables.

Gandhi et al. [9] proposed a randomized sequential rounding scheme (we refer to this algorithm as DRounding). DRounding guarantees marginal preservation—i.e., $\mathrm{E}[X_e] = x_e$, addressing (1)—and degree preservation, i.e. $\lfloor\sum_{e\in E_v} x_e\rfloor \leq \sum_{e\in E_v} X_e \leq \lceil\sum_{e\in E_v} x_e\rceil$ for all $v \in U \cup V$, addressing (3)+(4). We briefly describe the algorithm here: Call an edge x_e *integral* if $x_e \in \{0,1\}$ and *fractional* if $x_e \in (0,1)$. DRounding iteratively rounds fractional edges until all edges become integral. Each iteration of DRounding finds either a cycle or a maximal path in the subgraph spanned by the remaining fractional edges. Once such a cycle or path is found, at least one (but possibly more) of its edges are rounded (see [9] for details). The rounding of a cycle or path does not require global information, but only the values of the edges in the cycle or path, respectively. Thus it suffices to maintain the set F of the fractional edges, which shrinks after every rounding step. Since we can detect and process a cycle in $O(r)$ time, the total running time of DRounding is $O(rn)$, where as before $r = |U \cup V|$ and $n = |E|$.

In what follows, we show how to adapt DRounding to a distributed environment. Our distributed algorithm, called DDRounding, is inspired by recent work on "filtering" in the MapReduce framework and, in particular, the filtering algorithm for computing minimum spanning trees [18]. We exploit the fact that the approximation guarantees of DRounding do not depend on the order in which cycles or maximal paths are processed; we are thus free to choose an order that facilitates distributed processing. DDRounding is summarized as Alg. 2. In each iteration i, the algorithm checks whether the set F_{i-1} of remaining fractional edges is small. If so, we run DDRounding on F_{i-1} and output the solution. If not, we evenly distribute F_{i-1} across s_i compute nodes, where s_i is chosen carefully (see below). In parallel, each node then runs on its local partition a version of DDRounding that rounds cycles (can be detected locally) but not maximal paths (cannot be detected locally). After a local partition is rounded, it contains at most $m-1$ remaining fractional edges (since cycles have been rounded so that the remaining fractional edges form a tree). We then (conceptually) merge the remaining fractional edges across partitions to construct the set F_i for the next iteration.

Runtime and memory. The properties of DDRounding are similar to those of the filtering techniques of [18]; we describe them briefly here. Assume without loss of generality that $|E_v| \geq 1$ for all $v \in U \cup V$, i.e., every user and every item has at least one incident edge. Also assume that there are more edges than vertices, i.e., $n = r^{1+c}$ for some $0 < c \leq 1$. Assume that each compute node has insufficient memory to store the whole graph; in particular, each node can store $\eta = O(r^{1+\gamma})$ edges for $0 < \gamma < 1$. Further assume that the input fits in the aggregate memory of all nodes, i.e., there are $\Theta(n/\eta) = \Theta(r^{c-\gamma})$ nodes. Note that these assumptions imply that the set of vertices can be stored on a single node, while the set of edges cannot. We thus model the situation where there are many more edges than nodes, as it is often the case in practice. Set $n_{i-1} = |F_{i-1}|$. In iteration i, we use $s_i = \Theta(n_{i-1}/\eta)$ nodes so that each node stores $O(\eta)$ edges; memory constraints

Algorithm 2 DDRounding

Require: U, V, E, solution x of GBM$_\epsilon$-LP
  $I_0 \leftarrow \{ (e, x_e) : e \in E, x_e \in \{0, 1\} \}$ *// integral edges*
  $F_0 \leftarrow \{ (e, x_e) : e \in E, 0 < x_e < 1 \}$ *// fractional edges*
  for $i \in 1, 2, \ldots$ **do**
    if $|F_{i-1}| < \eta$ **then** *// few fractional edges?*
      $I_i \leftarrow \text{DRounding}(U, V, F_{i-1})$ *// sequential rounding*
      return $\bigcup_{j=0}^{i} I_j$ *// output integral solution*
    else
      $s_i \leftarrow \Theta(|F_{i-1}|/\eta)$ *// number of compute nodes*
      Partition F_{i-1} evenly across s compute nodes to obtain partitions $F_{i-1,1}, \ldots, F_{i-1,s_i}$
      for $j \in \{1, \ldots, s_i\}$ **do** *// in parallel*
        $T \leftarrow \text{RemoveCyclesWithDRounding}(U, V, F_{i-1,j})$
        $I_{i,j} \leftarrow \{ (e, X_e) \in T : X_e \in \{0, 1\} \}$
        $F_{i,j} \leftarrow T \setminus I_{i,j}$
      $I_i = \bigcup_{j=1}^{s_i} I_{i,j}$ *// newly obtained integral edges*
      $F_i = \bigcup_{j=1}^{s_i} F_{i,j}$ *// remaining fractional edges*

are thus preserved. Now observe that by the arguments above $n_i \leq s_i(r-1) = O(n_{i-1}/r^\gamma)$. Since $n_0 \leq n = O(r^{1+c})$, we conclude that $n_i = O(r^{1+c}/r^{i\gamma})$. The algorithm terminates as soon as $n_i = O(\eta)$, i.e., after $O(\lceil c/\gamma \rceil)$ iterations. Since every node runs a (less expensive variant of) DDRounding independently and in parallel, each iteration has time complexity $O(\eta r) = O(r^{2+\gamma})$. The overall time complexity is thus $O(r^{2+\gamma}\lceil c/\gamma \rceil)$.

Quality of approximation. ϵ-feasibility of X follows directly from the degree preservation property of DRounding. The marginal preservation property implies that the objective function is within $(1-\epsilon)$ of the optimum in expectation, since $\mathrm{E}[\, w^\top X \,] = w^\top x$. Note that $S = w^\top X$ is a bounded random variable (both from below and above). Standard arguments then show that if we round sufficiently often, then $w^\top X$ is close to $w^\top x$ with high probability. The number of required rounding steps depend on the problem though (i.e., on value of the optimal solution). In our experiments, we found that even a single run of DDRounding produces results close to or even above $w^\top x$.

Implementing DDRounding. Recall that, after MPCSolver has finished, each node j already stores locally a subset $F_{0,j}$ of the primal variables, where $|F_{0,j}| \leq \lceil n/s \rceil$. We thus set $s_1 = s$ so that there is no need for any data redistribution in the first iteration of DDRounding ($i = 1$). When available, we use multiple threads on each node to remove cycles with DRounding. In each subsequent iteration $i > 1$, we halve the number of available nodes so that $s_i = \lceil s_{i-1}/2 \rceil$. We thus need to communicate only half of the remaining fractional edges, i.e., the remaining fractional edges of every second node. This procedure balances communication cost evenly. Also note that iterations $i > 1$ are faster because every node processes at most $2r + 2$ edges; the bulk of the work is performed in the first iteration.

Implementing DRounding. Cycle detection using DFS can be implemented efficiently as follows. We start by selecting an arbitrary root node $v_0 \in U \cup V$ and perform DFS starting from v_0. Whenever we find a cycle $C = (vv_1 \cdots v_i v_{i+1} \cdots v_l v)$, some of its corresponding edges are rounded. Suppose that by doing so, only edge (v_i, v_{i+1}) becomes integral.[3] This removes an edge of the current path of DFS, i.e., we are in an invalid state. To avoid restarting DFS from scratch, we decompose C into two paths $C_1 = (vv_1 \cdots v_i)$ and, in reverse order, $C_2 = (vv_l \cdots v_{i+1})$. We replace the portion of the DFS stack that corresponds to C by C_1 or C_2, whichever is longer. To the extent possible, this allows us to avoid reprocessing the same paths over and over again. Moreover, once a node is fully processed (i.e., there are no more cycles involving this node), we mark it so that we never need to visit this node again. Both optimizations significantly improved the efficiency of our implementation of DRounding.

[3]We proceed similarly when more than one edge becomes integral.

4. RELATED WORK

The classical optimization task of finding an assignment of entities to users under a given set of constraints has been extensively investigated in various domains of computer science. Entities could be items in an auction [24], advertisements [6], scientific papers [10], social content [22], or multimedia items as in our case. Additionally, matching problems (in particular weighted b-matching) has been shown to be a useful tool in a wide variety of machine learning tasks, including semi-supervised learning [15], spectral clustering [14], graph embedding [26], and manifold learning [27]. Packing problems find applications in allocation of display ads [5]. In some of the applications mentioned above, the input data is not immediately available and decisions need to be made as new data arrives; this is referred to as the online version of the problem. In this paper, we focus on offline algorithms, which can help in solving the corresponding online versions [2].

The problem we studied in this paper is also known in the literature as the maximum weight degree-constrained subgraph problem. The special case with upper bounds only is known as maximum weight b-matching. In a centralized environment, both problems can be solved in polynomial time via linear programming solvers, maximum flow techniques [1], or using the combinatorial algorithm developed in [8]. Unfortunately, these algorithms do not cope well with massive datasets.

For b-matching, a simple greedy algorithm can achieve approximation guarantee of $1/2$, and there is a $(2/3 - \epsilon)$-approximation algorithm with expected running time $O(bm \log \frac{1}{\epsilon})$ [21]. To the best of our knowledge, the algorithm developed in [8] with running time $\Omega(n^{1/2}m)$ is the most efficient algorithm developed for exact b-matching to date. Scalable algorithms for b-matching have been presented in [22], where authors adapted two well-known algorithms for bipartite b-matching to the MapReduce environment. The problem has also been studied using the message passing model of distributed computation [23, 17]. None of the above algorithms can handle lower- and upper-bound constraints simultaneously.

There has been a significant effort in developing parallel and distributed algorithms for linear programs with packing and/or covering constraints. Both packing problems and covering problems have been defined in [25], which also gives efficient algorithms for each class of problems. A first parallel algorithm for packing problems and covering problems was presented in the seminal work of Luby and Nisan [20]. Awerbuch and Khandekar [3] presented a distributed stateless $(1+\epsilon)$-approximation algorithm for solving linear programs with only packing or only covering constraints. MPCSolver is based on this algorithm in that it also uses gradient descent with multiplicative updates; key differences are that MPCSolver can handle mixed packing-covering problems and can start from an arbitrary initial point. Young [28] developed a parallel $(1+\epsilon)$-approximation algorithm for mixed packing-covering problems, which obtains a solution that satisfies all constraints within a $(1 \pm \epsilon)$ factor in a poly-logarithmic number of rounds. Young's algorithm has a better theoretical bound than MPCSolver; in particular, its runtime is independent of the width of the problem (M). Young's algorithm initially sets all variables to sufficiently small values (all packing constraints satisfied), and then gradually increases but never decreases variables until either approximate feasibility is achieved or infeasibility is detected. In contrast to MPCSolver, variable increments cannot be undone. For this reason, Young's algorithm cannot start from an arbitrary starting point and it does not support adaptive

Table 1: Summary of datasets

Dataset	$\|U\|$	$\|V\|$	$\|E\|$
Netflix	456K	18k	99M
KDD	1M	620k	252M
Syn	10M	1M	1B
Semi-Syn	480k	18k	3.2B

changes of the error bound, which greatly affect performance in practice (see Sec. 5).

5. EXPERIMENTAL EVALUATION

We investigated the performance of our and alternative algorithms in an extensive experimental study on both (semi-)synthetic and real-world datasets. We found that our algorithms are competitive with state-of-the-art LP solvers on moderately large problem instances, and multiple orders of magnitude faster than alternative methods on large instances.

5.1 Experimental setup

Computational environment. We implemented MPCSolver, Young's algorithm [28], and DDRounding in C++. To ensure a fair comparison, we used the data distribution techniques of Sec. 3.3 when implementing Young's algorithm, which greatly reduced its communication cost. All algorithms employed MPICH2 for communication.[4] We used two different setups to run our experiments: (1) a single high-memory server and (2) a compute cluster consisting of 16 nodes (with significantly less main memory). The high-memory server had 512GB of main memory and was equipped with an Intel Xeon 2.40GHz processor with 32 cores. Each node in the compute cluster had 48GB of main memory and an Intel Xeon 2.40GHz processor with 8 cores.

Real-word datasets. Table 1 gives a brief overview of the datasets used in our experiments. We used two real-world datasets: the Netflix dataset [4], which consists of roughly 99M ratings (1–5) of 456k Netflix users for 18k movies, and the KDD dataset of Track 1 of KDD-Cup 2011 [7], which consists of approximately 253M ratings of 1M Yahoo! Music users for 625k musical pieces. We excluded users with less than 10 ratings from the Netflix dataset (this ensures feasibility and is more realistic; see the discussion below). In both datasets, the number of ratings per user is unbalanced, i.e., there are users with very few ratings but also users with a large number of ratings. To construct a GBM instance, we converted each dataset to a bipartite graph (users and items form vertices; ratings correspond to weighted edges); our goal was to recommend items to users. We used a lower bound of 3 and an upper bound of 5 for the number of recommendations given to each user. We did not enforce a lower bound for items, but require each item to be recommended at most 200 times (Netflix) or 2000 times (KDD). These choice of bounds ensured that the resulting instance was feasible.

Large-scale datasets. The real-world datasets above are somewhat unrealistic because we recommend items to users that the users have already rated. A more realistic setup is covered by our large-scale semi-synthetic dataset, denoted Semi-Syn. In more detail, we applied the rating prediction algorithm of [11] to predict unknown ratings in the Netflix matrix. Next, we sampled 3.2B entries uniformly from this matrix; each of the samples corresponds to an edge in the bipartite graph, weighted by the predicted rating. Note that this dataset is balanced, i.e., each user has the same number of ratings in expectation. The Semi-Syn dataset has a large number of edges but only a moderate number of vertices. In order to investigate the performance of our algorithms with a large number of vertices, we additionally generated a synthetic dataset (denoted Syn) with 10M users, 1M items, and 1B edges. The edges and their corresponding weights (between 1 and 5) are sampled uniformly at random. We generally use the same lower and upper bounds as for Netflix, except for Syn, where we modify the upper bound on the number of recommendations for each item to 50.

Optimal solution. In order to compute the value of the optimum, we used the latest version of Gurobi optimizer 5.0 [12], a state-of-the-art commercial solver for linear programs (and other problems); Gurobi takes advantage of multiple cores if available. Even though our real-world datasets were only moderately large, Gurobi was not able to solve them on one of our cluster nodes due to insufficient memory. On the high-memory server, however, Gurobi did produce an optimal solution for the real-world datasets. For Syn and Semi-Syn, Gurobi ran out of memory even on the high-memory server; we thus were not able to compute the value of the optimum.

Convergence test. When running MPCSolver and Young's algorithm, we need to detect whether the algorithms have "practically" converged. Say that a solution has *maximum violation* λ if it is barely λ-feasible (i.e., not λ'-feasible for any $\lambda' > \lambda$). For MPCSolver we used the heuristic convergence test as described in Sec. 3.3. Young's algorithm was declared converged as soon as the maximum violation of only the packing constraints and only the covering constraints became equal (or the solution had maximum violation of ϵ). This modified convergence bound for Young's algorithm ensures a fair comparison: If we ran the algorithm any further, the maximum violation would increase (i.e., covering violation gets smaller, packing violation gets larger).

5.2 Results for GBM-LP (feasibility)

In our first set of experiments, we compared MPCSolver and Young's algorithm for GBM-LP feasibility problems with respect to efficiency (in terms of both number of iterations and total time to convergence) and (strong) scalability. Our goal was to produce a 0.05-feasible solution, i.e., $\epsilon = 0.05$. As discussed below, running any of the two algorithms with such a small error bound leads to poor performance in practice. Instead, we ran the algorithms with some value $\epsilon' \geq \epsilon$ in the hope to still get an ϵ-feasible solution. For MPCSolver, we also considered the adaptive scheme of Sec. 3.3 (with the parameters given in that section). The time for rescaling the input problem and computing the starting point was negligible (a few seconds) and is not included in our plots.

Efficiency. We studied the effect of error bound parameter ϵ' for both MPCSolver and Young's algorithm. For MPCSolver, ϵ' refers to the internal error bound, while for Young's algorithm ϵ' refers to the desired error bound given to the algorithm. Note that the choice of ϵ' affects both running time and the feasibility of the final solution. Fig. 1 plots the maximum violation after every iteration until convergence for various choices of ϵ'. Here we only used a single cluster node with 8 parallel threads. All algorithms were run from the same initial point.

First note that ϵ' indeed affects time to convergence for both algorithms, which is in accordance with theory (MPCSolver is linear in $(\epsilon')^{-5}$, Young's algorithm in $(\epsilon')^{-4}$). Nevertheless, both algorithms produce solutions with maximum violation far less than ϵ' (but above $\epsilon = 0.05$); this effect was more pronounced for MPCSolver. For all choices of $\epsilon' \geq 1$, MPCSolver converged faster and achieved a higher precision (i.e., less maximum violation). Moreover, for both algorithms and on both datasets, a choice of $\epsilon' = 1$ worked best within 1500 iterations; the final maximum violation achieved by MPCSolver was slightly better than the one obtained by Young's algorithm (0.19 vs. 0.21 on Netflix, 0.13 vs. 0.16 on KDD). For $\epsilon' = 0.5$, neither of the algorithms converged after 1500 iterations: The iterate moved very slowly towards approximate feasibility. On Netflix (KDD) the achieved error was 0.98 (0.99) for MPCSolver

[4] http://www.mcs.anl.gov/mpi/mpich/

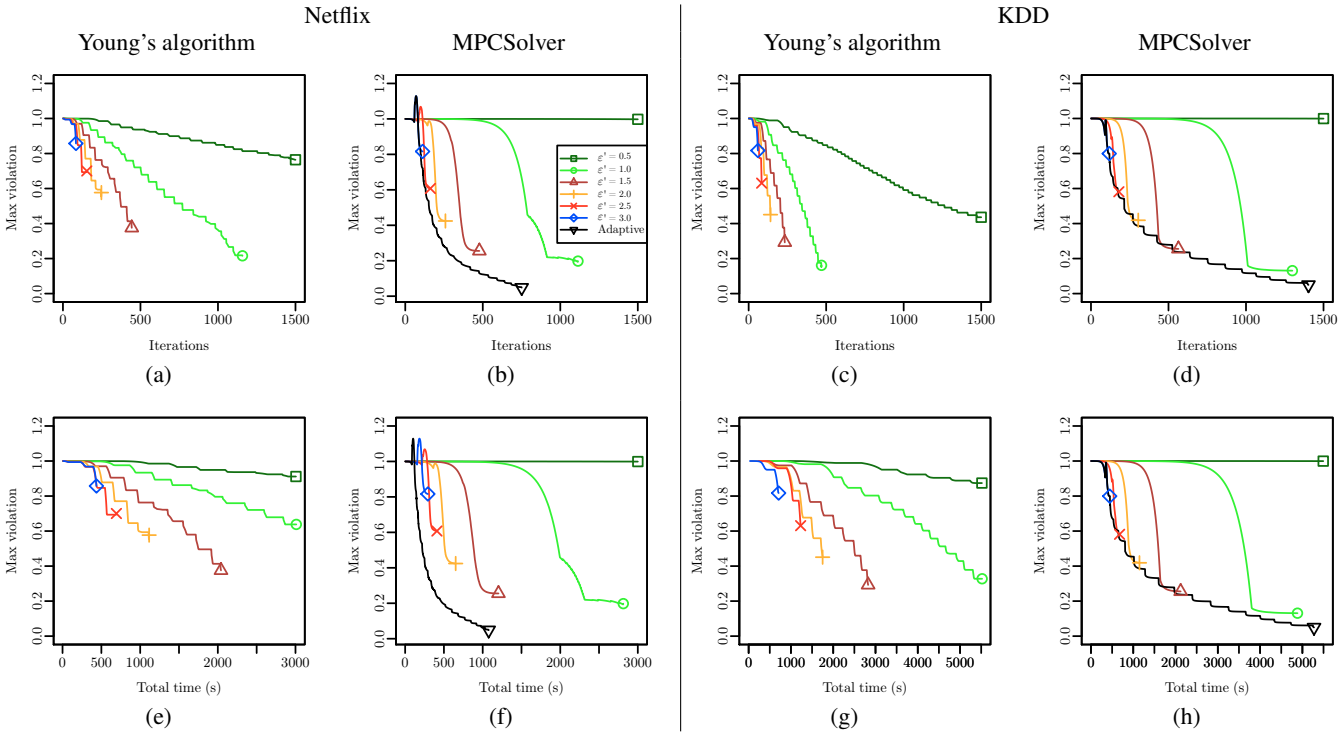

Figure 1: Efficiency of Young's algorithm and MPCSolver for 0.05-feasibility on real-world datasets (1x8)

and 0.76 (0.43) for Young's algorithm, respectively.

For high values of ϵ', the maximum violation dropped quickly in the beginning, but did not improve significantly in further iterations (in fact, most algorithms converged quickly for large ϵ'). For small values of ϵ', the maximum violation improved more slowly but eventually reached a lower value. This behavior motivated our adaptive method for selecting ϵ' (denoted "adaptive") discussed in Sec. 3.3. Recall that Young's algorithm cannot be run with adaptive error bound selection; see Sec. 4. In contrast, MPCSolver is well suited to adaptive error bound selection because it can start from an arbitrary starting point and is able to "undo" bad steps. As can be seen in Fig. 1, MPCSolver with adaptive ϵ' outperforms all fixed choices of ϵ'. As a consequence, MPCSolver with adaptive error bound selection was the only method that achieved 0.05-feasibility in 1500 or less iterations. In particular, the maximum violation fell below 0.05 within 1077s (Netflix) and 5317s (KDD). In what follows, we focus on MPCSolver with adaptive ϵ'.

Regarding running time, we found that an iteration of MPCSolver was slightly faster than an iteration of Young's algorithm (1.8x faster on average). Communication costs were similar, but Young's algorithm is more computationally intensive (since it needs to compute a global parameter from time to time). However, MPCSolver required significantly less iterations to converge, mainly due to its flexibility in terms of selecting ϵ'. Overall, MPCSolver was multiple orders of magnitude faster than Young's algorithm.

As a final remark, Gurobi required about 2h (Netflix) and 3.5h (KDD) to find a feasible solution on our high-memory server. To have a fair comparison to MPCSolver, we also used Gurobi to find 0.05-feasible solution: the running times decreased to 1.9h (Netflix) and 3.3h (KDD). Thus MPCSolver was competitive in terms of overall runtime to state-of-the-art parallel solvers for feasibility problems.

Strong scalability. We investigated the runtime performance (measured as the average time per iteration and the total time until convergence) of MPCSolver and Young's algorithm as we increase the number of compute nodes from 1 to 16, where each node runs 8 threads. We refer to these setups using a "node x threads" abbreviation, i.e., 1x8, 2x8, ..., 16x8. Our results are summarized in Table 2.

We first discuss the results on the moderately-sized real-world datasets (which do fit into the memory of a single cluster node). Compared to the per-iteration execution time of Netflix (KDD) on 1x8, MPCSolver provided 1.6x (1.7x) speed-up on 2x8, and a 1.9x (2.4x) speed-up on 4x8. Here communication overhead becomes significant so that speed-up is sublinear. Young's algorithm has slightly higher speed-up than MPCSolver, but starts at a higher cost initially. In fact, Young's algorithm on 8 nodes takes more time per iteration as MPCSolver on 2 nodes. Similar to MPCSolver, the benefit of moving from 4 to 8 or more nodes was marginal. Turning to overall time to convergence, we found that Young's algorithm did not converge after 24h. Note that we ran Young's algorithm with $\epsilon' = 0.05$ to ensure that we can actually find an ϵ-feasible solution, which caused it to move very slowly. In contrast, MPCSolver converged after 660 (1050) iterations on Netflix (KDD); the total running time was less than 1h on 4x8 and above.

We also investigated the performance of MPCSolver and Young's algorithm on the large Syn and Semi-Syn datasets. For Syn (Semi-Syn), we give results for 4x8, 8x8, and 16x8 (8x8 and 16x8) only; a smaller number of nodes had insufficient aggregate memory to store the data. On Syn, MPCSolver achieved 1.5x speed-up when moving from 4x8 to 8x8. However, the algorithm ran solely 1.3x faster when using 16x8. This sublinear speed-up is caused by increased communication cost (see below). Young's algorithm exhibited similar speed-ups. Each MPCSolver iteration took 8s on average using 16x8; MPCSolver converged to a 0.05-feasible solution after 551 iterations (1.2h). In contrast, iterations of Young's algorithm took longer (14.6s) and the algorithm did not converge within 24 hours. On Semi-Syn, though, we observed a better scalability for both

Table 2: Performance of Young's algorithm (Young) and MPCSolver (MPC) for feasibility problems ($\epsilon = 0.05$)

	1x8		2x8		4x8		8x8		16x8	
	Young	MPC	Young	MPC	Young	MPC	Young	MPC	Young	MPC
Average time/iteration (s) on Netflix	5	2.8	3.5	1.8	2.7	1.5	2.3	1.3	2	1.1
Average time/iteration (s) on KDD	14	7.8	8.7	4.6	6.2	3.3	5.2	3	4.4	2.7
Average time/iteration (s) on Syn					29.2	15.7	19.7	10.5	14.6	8
Average time/iteration (s) on Semi-syn							21	11	13.1	6.9
Time for feasibility (h) on Netflix	>24	0.51	>24	0.33	>24	0.28	>24	0.24	>24	0.2
Time for feasibility (h) on KDD	>24	2.3	>24	1.3	>24	0.96	>24	0.87	>24	0.78
Time for feasibility (h) on Syn					>24	2.4	>24	1.6	>24	1.2
Time for feasibility (h) on Semi-syn							>24	3.4	>24	2.1

MPCSolver and Young's algorithm; both algorithms provided 1.6x speed-up on 16x8. To understand this behavior, recall that the communication cost of each MPCSolver iteration is governed by the number of vertices, whereas the computation cost depends on the number of edges. Semi-Syn has less vertices than Syn (less communication) and many more edges (more computation) so that Semi-Syn is easier to parallelize. On Semi-Syn, each MPCSolver iteration required 6.9s on average using 16x8, and MPCSolver required 1120 iterations (2.1h) to compute a 0.05-feasible solution. Similar to Syn, iterations of Young's algorithm took longer (13.1s) and the algorithm did not converged within 24 hours.

We conclude that MPCSolver scales to very large datasets and is significantly faster than Young's algorithm. Moreover, MPCSolver is competitive to state-of-the-art LP solvers, but can handle much larger problem instances.

5.3 Results for GBM-LP (optimality)

Recall that we need to run a sequence of feasibility instances to determine an optimal solution to the GBM problem. We first present results for Netflix and KDD with $\epsilon = 0.05$. We compared the solution of MPCSolver with the optimal solution computed by Gurobi and found that the desired approximation ratio was obtained (i.e., the value of the objective was at least 95% of the optimum). On 16x8, MPCSolver required 2.9h (Netflix) and 7.4h (KDD) in total. On the high-memory server, Gurobi required 2.2h (Netflix) and 3.8h (KDD).

For both Syn and Semi-Syn, Gurobi ran out of memory. In contrast, MPCSolver required 9.5h (13.6h) for Syn and 14.7h (24h) for Semi-Syn on 16x8 (8x8), respectively. Note that the high-memory server had sufficient memory for MPCSolver to process both Syn and Semi-Syn; our algorithm required 9.7h (Syn) and 18h (Semi-Syn) using all 32 cores. Thus MPCSolver is faster on the high-memory server (with 32 cores) than on the 8x8 cluster setup (with 64 cores). Although the hardware in both setups is not identical, a key reason for the performance difference is that communication between workers is fast in shared-memory systems (high-memory server) but significantly slower in shared-nothing systems (compute cluster).

Overall, we found that MPCSolver is competitive in terms of overall runtime. It's key advantage is that it can scale over a compute cluster and thus to much larger problem instances; it's key disadvantage is that results are approximate up to ϵ.

5.4 Results for distributed rounding

Recall that the approximate solution of GBM-LP is generally not integral. In our next set of experiments, we evaluated the performance of DDRounding to produce an integral solution. As before, we measured performance with respect to quality (i.e., value of objective function after rounding), efficiency, and strong scalability. We used the 1x8 to 16x8 setups described previously.

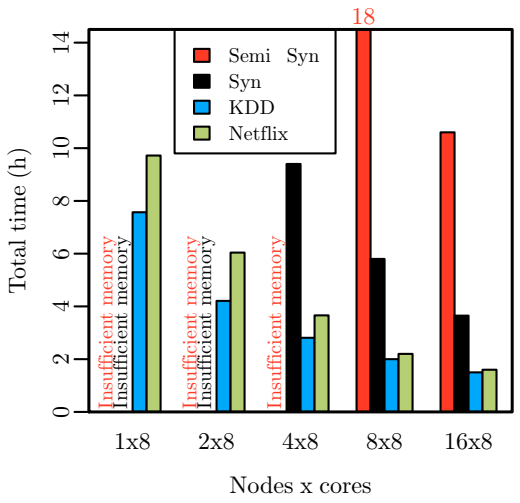

Figure 2: Scalability of DDRounding on different datasets

Quality. For all datasets, we took the fractional solution obtained by MPCSolver with $\epsilon = 0.05$ as input to DDRounding. For each real-world dataset, we executed 10 independent runs of DDRounding, thereby obtaining 10 integral solutions. We found that the value of the objective differed only marginally (within 0.5%) from the value of the fractional solution across all 10 runs. Moreover, for Netflix (KDD) the best run produces solutions that were slightly better than the fractional solution (increase of 0.5% for Netflix, 0.1% for KDD); this increase is possible because the fractional solution is ϵ-optimal but not optimal. For Syn and Semi-Syn, we only executed a single run. As before, the objective was close to the fractional solution ($\approx 0.01\%$ off) for both datasets. Thus the integral solutions obtained by DDRounding rounding were of essentially the same quality as the fractional solutions.

Efficiency and strong scalability. Fig. 2 shows the performance of DDRounding with varying number of cluster nodes. As can be seen, DDRounding clearly benefits from distributed processing, i.e., performance improved significantly when adding more nodes: The local subgraph processed in the DRounding step on each node becomes smaller and so that cycles can be removed more efficiently. We expect, however, that using too many nodes is not beneficial because then the number of cycles per subgraph may become too low (so that there is little work to do). Nevertheless, we found that the performance of DDRounding improves even if we go beyond 8x8, even on our moderately-sized real-world datasets. On 16x8, we achieved an integral solution for all the real-world datasets within 1.6h. For Syn, we obtained an integral solution after 9.4h, 5.8h, and 3.6h using 4x8, 8x8, and 16x8, respectively; on smaller setups, the data did not fit into the main memory. We observed similar speed-ups for Semi-Syn; the time to obtain an integral solution

was reduced from 18h using 8x8 to 10.5h on 16x8; as before, the data exceeded the main memory available when using less than 8 compute nodes.

Constraint violations. Recall that DDRounding preserves the lower- and upper-bound constraints (up to rounding). In our next experiment, we investigated the actual constraint violations obtained by MPCSolver, both before and after applying DDRounding. Table 3 summarizes our results. First note that on all datasets, both the fractional and the integral solution violated only very few constraints (i.e., is almost feasible). As mentioned previously, our error analysis is somewhat loose so that MPCSolver performs better than guaranteed by theory. For Netflix and Semi-Syn, DDRounding further decreased the violations in the constraints (slightly).

5.5 Results for GBM

In our final experiments, we put everything together and investigated how well we can solve the GBM problem. Recall that computation of an optimal integral solution with Gurobi on the high-memory server took 2.2h (3.8h) for Netflix (KDD). With MPCSolver and DDRounding on 16x8 and for our choice of $\epsilon = 0.05$, we obtained a 0.05-feasible solution in 4.5h (8.9h) that was 95.5% (95.1%) of the optimal solution. Thus the desired approximation ratio was indeed realized. Approximately solving GBM-LP took 2.9h (7.4h), rounding to an integral solution took 1.6h (1.5h). Observe that MPCSolver with rounding was slower than Gurobi. However, the individual cluster nodes used by our algorithms were less powerful, both in terms of memory and in terms of number of cores. Moreover, our approach can handle much larger problem instances. For Syn and Semi-Syn, which cannot be handled by Gurobi on our high-memory server, a 0.05-feasible solution to GBM was obtained after 13.1h (9.5h for GBM-LP, 3.6h for rounding) and 25.2h (14.7h for GBM-LP, 10.5h for rounding), respectively, on 16x8. We expect that running time can be further reduced by adding more compute nodes.

6. CONCLUSIONS

We presented distributed algorithms to approximately solve large-scale generalized bipartite matching problems. Our approach is based on linear programming and randomized rounding. In particular, we developed MPCSolver, a distributed algorithm to approximately solve large-scale mixed packing-covering linear programs, and DDRounding, a distributed rounding algorithm to obtain an integral solution of high quality. Our experiments suggest that our algorithms are more scalable and more efficient than alternative approaches.

7. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. A general approach to online network optimization problems. *ACM Transactions on Algorithms*, 2(4):640–660, 2006.

[3] B. Awerbuch and R. Khandekar. Stateless distributed gradient descent for positive linear programs. *SIAM J. Comput.*, 38(6):2468–2486, 2009.

[4] J. Bennett and S. Lanning. The Netflix prize. In *KDD Cup and Workshop*, 2007.

[5] A. Bhalgat, J. Feldman, and V. S. Mirrokni. Online allocation of display ads with smooth delivery. In *KDD*, pages 1213–1221, 2012.

[6] D. X. Charles, M. Chickering, N. R. Devanur, K. Jain, and M. Sanghi. Fast algorithms for finding matchings in lopsided bipartite graphs with applications to display ads. In *ACM Conference on Electronic Commerce*, pages 121–128, 2010.

[7] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer. The Yahoo! Music Dataset and KDD-Cup'11. In *KDDCup 2011 Workshop*, 2011.

[8] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[9] R. Gandhi, S. Khuller, S. Parthasarathy, and A. Srinivasan. Dependent rounding and its applications to approximation algorithms. *J. ACM*, 53(3):324–360, 2006.

[10] N. Garg, T. Kavitha, A. Kumar, K. Mehlhorn, and J. Mestre. Assigning papers to referees. *Algorithmica*, 58(1):119–136, 2010.

[11] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77, 2011.

[12] I. Gurobi Optimization. Gurobi optimizer reference manual, 2012.

[13] B. C. Huang and T. Jebara. Fast b-matching via sufficient selection belief propagation. *Journal of Machine Learning Research - Proceedings Track*, 15:361–369, 2011.

[14] T. Jebara and V. Shchogolev. B-matching for spectral clustering. In *ECML*, pages 679–686, 2006.

[15] T. Jebara, J. Wang, and S.-F. Chang. Graph construction and *b*-matching for semi-supervised learning. In *ICML*, page 56, 2009.

[16] C. Koufogiannakis and N. E. Young. Distributed fractional packing and maximum weighted b-matching via tail-recursive duality. In *DISC*, pages 221–238, 2009.

[17] C. Koufogiannakis and N. E. Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.

[18] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *SPAA*, pages 85–94, 2011.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[20] M. Luby and N. Nisan. A parallel approximation algorithm for positive linear programming. In *STOC*, pages 448–457, 1993.

[21] J. Mestre. Greedy in approximation algorithms. In *ESA*, pages 528–539, 2006.

[22] G. D. F. Morales, A. Gionis, and M. Sozio. Social content matching in mapreduce. *PVLDB*, 4(7):460–469, 2011.

[23] A. Panconesi and M. Sozio. Fast primal-dual distributed algorithms for scheduling and matching problems. *Distributed Computing*, 22(4):269–283, 2010.

[24] M. Penn and M. Tennenholtz. Constrained multi-object auctions and b-matching. *Inf. Process. Lett.*, 75(1-2):29–34, 2000.

[25] S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *FOCS*, pages 495–504, 1991.

[26] B. Shaw and T. Jebara. Minimum volume embedding. *Journal of Machine Learning Research - Proceedings Track*, 2:460–467, 2007.

[27] B. Shaw and T. Jebara. Structure preserving embedding. In *ICML*, page 118, 2009.

[28] N. E. Young. Sequential and parallel algorithms for mixed packing and covering. In *FOCS*, pages 538–546, 2001.

Table 3: Constraint violations before (fractional solution) and after (integral solution) rounding

	# satisfied constraints		# ϵ-feasible (but not feasible)		Avg. rel. violation (infeasible)		Max rel. violation (infeasible)	
	Fractional	Integral	Fractional	Integral	Fractional	Integral	Fractional	Integral
Netflix	930 085	930 086	45	44	0.0473	0.0145	0.0496	0.015
KDD	2 626 509	2 626 509	432	432	0.0407	0.0409	0.0499	0.05
Syn	20 998 614	20 998 623	1386	1377	0.0485	0.0491	0.0499	0.05
Semi-Syn	978 090	978 105	58	43	0.0492	0.0163	0.0498	0.02

APPENDIX

A. PROOF OF MAIN THEOREM

Denote by $x(t), y(t)$, and $z(t)$ the values of x, $y(x)$, and $z(x)$ in round t. The potential in round t is then given by

$$\Phi(t) = \mathbb{1} \cdot y(t) + \mathbb{1} \cdot z(t). \tag{4}$$

Note that y and z are fast-growing functions of x: Any "significant" change in the value of x leads to a "significant" change in the value of Φ. We assume throughout that the problem instance is feasible; otherwise, our analysis implies that if Alg. 1 does not find a solution after the poly-logarithmic number of rounds asserted by Th. 1, the MPC-LP is infeasible.

Our proof follows the outline given in Sec. 3.1. The key ideas underlying our proof are due to [3] (who considered packing LPs or covering LPs, but not MPCs); here we adapt the proof to our setting. We make use of the internal step size parameter $\epsilon' = \epsilon/10$ throughout. We first discuss our choice of parameters. First, parameters β and δ are chosen such that if x satisfies $\boldsymbol{P}x \leq 3 \cdot \mathbb{1}$ in the beginning of a round, the value of $\boldsymbol{P}_j^\top y(x)$ changes by at most a factor of $\alpha/4$ in that round, $1 \leq j \leq n$. Similarly, if x satisfies $\boldsymbol{C}_i^\top x \leq 3$, then the value of $\boldsymbol{C}_j^\top z(x)$ for any j changes by at most $\alpha/4$. Finally, our parameter choice ensures that the increase in $\boldsymbol{P}_i x$ (upper bounded by $\beta + n \cdot M \cdot \delta$) is at most $1/(8\mu)$.

A.1 Warm-up interval

We show in Lemma 3 that after a warm-up interval of poly-logarithmic length, we do not violate the packing and covering constraints by too much. This allows us to bound the value of function Φ so that we can bound the number of stationary and unstationary intervals later on.

LEMMA 3. *Let the algorithm start from a point $x^0 \in \Re_+^n$ and let $x_{\max} = \max_j x_j^0$. After $\tau_0 = O(\beta^{-1} \log(\delta^{-1} nkM(x_{\max} + \delta))$ rounds, as long as $x(t)$ does* not *form a feasible solution to (1), we have*

- $1 - 2\epsilon' \leq \max_i \boldsymbol{P}_i x(t) < 2 + 2\epsilon'$, *and*
- $\min_i \boldsymbol{C}_i x(t) \leq 1 + 2\epsilon'$.

PROOF. Assume that $\max_i \boldsymbol{P}_i x > 2 + \epsilon'$ holds initially. Note that for all i with $\boldsymbol{P}_i x > 2 + \epsilon'$, we have $y_i > mkM/\epsilon' \cdot \exp(\mu)$. Note also that $z_i \leq \exp(\mu)$ for all i. Thus, for all variables j such that $\boldsymbol{P}_{ij} \neq 0$ for some i with $\boldsymbol{P}_i x > 2 + \epsilon'$, it holds that $\boldsymbol{P}_j^\top y / \boldsymbol{C}_j^\top z > 1 + \alpha$. Thus all these variables decrease by a factor of β. Thus after $O(\beta^{-1} \log(nMx_{\max}))$ rounds, $\max_i \boldsymbol{P}_i x \leq 2 + \epsilon'$. Moreover since $\boldsymbol{P}_i x$ in a single round increases to at most $(1+\beta)\boldsymbol{P}_i x + nM\delta$, we have that $\max_i \boldsymbol{P}_i x$ can increase to at most $(2+\epsilon')(1+\beta) + nM\delta < 2 + 2\epsilon'$ in any subsequent round.

Now consider the duration in which x does not form an $O(\epsilon')$-feasible solution to (1). Also assume that $\max_i \boldsymbol{P}_i x \leq 1 - \epsilon'$. Thus $y_i \leq \epsilon'/(mkM)$ for all i. Note that since x is not a feasible solution, we have $\min_i \boldsymbol{C}_i x \leq 1$. Since $\boldsymbol{C}_i x \leq 1$ implies $z_i \geq 1$, we have that all j with $\boldsymbol{C}_{ij} \neq 0$ for some i with $\boldsymbol{C}_i x \leq 1$ satisfy $\boldsymbol{P}_j^\top y / \boldsymbol{C}_j^\top z \leq \epsilon'/k < 1 - \alpha$. Thus all these variables increase by factor $(1+\beta)$. Thus after $O(\beta^{-1} \log \delta^{-1})$ rounds, we have $\max_i \boldsymbol{P}_i x > 1 - \epsilon'$. Since any x_j decreases by a factor of at most $(1-\beta)$ in any single round, we always have $\max_i \boldsymbol{P}_i x > (1-\epsilon')(1-\beta) \geq 1 - 2\epsilon'$.

Now assume that $\min_i \boldsymbol{C}_i x \geq 1 + \epsilon'$. Thus $z_i \leq \epsilon'/(mkM)$ for all i. Note that since x is not a feasible solution, we have $\max_i \boldsymbol{P}_i x \geq 1$. Since $\boldsymbol{P}_i x \geq 1$ implies $y_i \geq 1$, we have that all j with $\boldsymbol{P}_{ij} \neq 0$ for some i with $\boldsymbol{P}_i x \geq 1$ satisfy $\boldsymbol{P}_j^\top y / \boldsymbol{C}_j^\top z \geq m/\epsilon' > 1 + \alpha$. Thus all these variables decrease by factor $(1-\beta)$. Thus after $O(\beta^{-1} \log(kMx_{\max}))$ rounds, we have $\min_i \boldsymbol{C}_i x < 1 + \epsilon'$. Since any x_j increases by a factor of at most $(1+\beta)$ in any single round, we always have $\min_i \boldsymbol{C}_i x < (1+\epsilon')(1+\beta) \leq 1 + 2\epsilon'$. ■

The following rather technical lemma will be needed in the proof of Lemma 7.

LEMMA 4. *After τ_0 rounds, we have*

$$\begin{aligned} (\max_i \boldsymbol{P}_i x - \epsilon')(\mathbb{1} \cdot y) &\leq (1+\epsilon')\, y^\top \boldsymbol{P} x, \\ (\min_i \boldsymbol{C}_i x + \epsilon')(\mathbb{1} \cdot z) &\geq z^\top \boldsymbol{C} x. \end{aligned}$$

PROOF. Let $1 \leq i_0 \leq m$ be such that $\boldsymbol{P}_{i_0} x = \max_i \boldsymbol{P}_i x$. Let $S = \{i \mid \boldsymbol{P}_i x < \boldsymbol{P}_{i_0} x - \epsilon'\}$. Thus $y_i < y_{i_0} \exp(-\mu\epsilon') < y_{i_0}\epsilon'/m$ holds for all $i \in S$. Hence $\sum_{i \in S} y_i \leq \epsilon' y_{i_0}$ and $\mathbb{1} \cdot y < (1+\epsilon') \sum_{i \notin S} y_i$. Therefore $(\max_i \boldsymbol{P}_i x - \epsilon')(\mathbb{1} \cdot y) < (1+\epsilon') \sum_{i \notin S} \boldsymbol{P}_i x \cdot y_i \leq (1+\epsilon')\, y^\top \boldsymbol{P} x$.

Now let $1 \leq i_0 \leq k$ be such that $\sigma := \boldsymbol{C}_{i_0} x = \min_i \boldsymbol{C}_i x$ and let $S = \{i \mid \boldsymbol{C}_i x > \sigma + \epsilon'\}$. From Lemma 3, we get that $\sigma \leq 1 + 2\epsilon'$. It is easy to check that $\epsilon' z_{i_0} = \epsilon' \exp(\mu(1-\sigma)) \geq (\sigma + \epsilon') \exp(\mu(1 - \sigma - \epsilon')) \cdot k$. Now fix $i \in S$ and let $\eta = \boldsymbol{C}_i x$. Since $\eta \exp(\mu(1-\eta))$ is a decreasing function of η for $\eta \geq 1/\mu$, we get that $\epsilon' z_{i_0} \geq \sum_{i \in S} z_i \cdot \boldsymbol{C}_i x$. This implies that $\sum_{i \notin S} z_i \cdot \boldsymbol{C}_i x + \epsilon' z_{i_0} \geq z^\top \boldsymbol{C} x$. From the definition of S, now we get that $(\min_i \boldsymbol{C}_i x + \epsilon')(\mathbb{1} \cdot z) \geq \sum_{i \notin S} z_i \cdot \boldsymbol{C}_i x + \epsilon' z_{i_0} \geq z^\top \boldsymbol{C} x$ as desired. ■

A.2 Potential is monotonically non-increasing

The following lemma shows that the potential is monotonically non-increasing after τ_0 rounds, and that its decrease can be bounded from below (w.r.t. the current values of x, y and z).

LEMMA 5. *Let $\Delta\Phi(t) = \Phi(t+1) - \Phi(t)$ denote the increase in Φ in round t. Similarly let $\Delta x_j(t) = x_j(t+1) - x_j(t)$, $\Delta y_i(t) = y_i(t+1) - y_i(t)$, and $\Delta z_i(t) = z_i(t+1) - z_i(t)$. After τ_0 rounds, we have*

$$-\Delta\Phi(t) \geq \Omega(\alpha) \textstyle\sum_i |\Delta y_i(t)| \tag{5}$$

$$-\Delta\Phi(t) \geq \Omega(\alpha) \textstyle\sum_i |\Delta z_i(t)| \tag{6}$$

$$-\Delta\Phi(t) \geq \Omega(\beta\mu)\big[z(t)^\top \boldsymbol{C} x(t) - (1+\alpha) \cdot y(t)^\top \boldsymbol{P} x(t)\big] \tag{7}$$

$$-\Delta\Phi(t) \geq \Omega(\beta\mu)\big[(1-\alpha) \cdot y(t)^\top \boldsymbol{P} x(t) - z(t)^\top \boldsymbol{C} x(t)\big] \tag{8}$$

PROOF. Note that the potential Φ is a convex differentiable function of x. Thus for any two vectors x^0 and x^1, we have $\Phi'(x^0) \cdot (x^1 - x^0) \leq \Phi(x^1) - \Phi(x^0) \leq \Phi'(x^1) \cdot (x^1 - x^0)$ where $\Phi'(x)$ is the gradient of Φ evaluated at x. Note that $\Phi'_j(x) = \mu(\boldsymbol{P}_j^\top y(x) - \boldsymbol{C}_j^\top z(x))$ for all j. Thus $\Delta\Phi(t) \leq \mu \sum_j \Delta x_j(t) \cdot (\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1))$.

We first prove (5). Note that for any j such that $\Delta x_j(t) > 0$, Lemma 3 implies that $\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1) \leq (1 + \alpha/4) \cdot$

$\boldsymbol{P}_j^\top y(t) - (1-\alpha/4)\cdot \boldsymbol{C}_j^\top z(t) \le -\Omega(\alpha)\cdot \boldsymbol{P}_j^\top y(t)$, where the last inequality follows from $\Delta x_j(t) > 0$. Similarly for any j such that $\Delta x_j(t) < 0$, we have $\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1) \ge (1-\alpha/4)\cdot \boldsymbol{P}_j^\top y(t) - (1+\alpha/4)\cdot \boldsymbol{C}_j^\top z(t) \ge \Omega(\alpha)\cdot \boldsymbol{P}_j^\top y(t)$. From above analysis, we have $\Delta\Phi(t) \le -\Omega(\alpha)\cdot\mu\sum_j |\Delta x_j(t)|\cdot \boldsymbol{P}_j^\top y(t)$. Now note that for any i, we have $\Delta y_i(t) \le \sum_j \mu \boldsymbol{P}_{ij} y_i(t+1)\cdot \Delta x_j(t) \le (1+\alpha/4)\sum_j \mu \boldsymbol{P}_{ij} y_i(t)\cdot \Delta x_j(t)$. Therefore $|\Delta y_i(t)| \le (1+\alpha/4)\sum_j \mu \boldsymbol{P}_{ij} y_i(t)\cdot |\Delta x_j(t)|$. Summing up over all i, we get $\sum_i |\Delta y_i(t)| \le O(1)\cdot \sum_j \boldsymbol{P}_j^\top y(t)\cdot |\Delta x_j(t)|$ and (5) follows. The proof of (6) is similar to the above and is omitted. We now prove (7):

$$\begin{aligned}
\Delta\Phi(t) &\le \mu\textstyle\sum_j \Delta x_j(t)\big[\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1)\big] \\
&\le \mu \sum_{j:\Delta x_j(t)>0} \Delta x_j(t)\big[\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1)\big] \\
&\le \beta\mu \sum_{j:\Delta x_j(t)>0} x_j(t)\big[\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{C}_j^\top z(t+1)\big] \\
&\le \Omega(\beta\mu) \sum_{j:\Delta x_j(t)>0} x_j(t)\big[(1+\alpha)\cdot \boldsymbol{P}_j^\top y(t) - \boldsymbol{C}_j^\top z(t)\big] \\
&\le \Omega(\beta\mu)\textstyle\sum_j x_j(t)\big[(1+\alpha)\cdot \boldsymbol{P}_j^\top y(t) - \boldsymbol{C}_j^\top z(t)\big] \\
&= \Omega(\beta\mu)\big[(1+\alpha)\cdot y(t)^\top \boldsymbol{P}x(t) - z(t)^\top \boldsymbol{C}x(t)\big].
\end{aligned}$$

The third inequality follows from the fact that if $\Delta x_j(t) > 0$ then we indeed have $\Delta x_j(t) \ge \beta x_j(t)$. The second and the fifth inequalities follow from $\Delta x_j(t) < 0$. The proof of (8) is similar and omitted. ■

A.3 Stationary and unstationary intervals

We proceed by defining *stationary* intervals in which the potential function does not change significantly.

DEFINITION 1 (STATIONARY INTERVAL). *An interval $\mathcal{T} = [t_0, t_1]$ of rounds is called* stationary *if all of the following conditions hold:*

- $\sum_{t\in\mathcal{T}}\sum_i |\Delta y_i(t)| \le \kappa_1\cdot\Phi(t_0)$.
- $\sum_{t\in\mathcal{T}}\sum_i |\Delta z_i(t)| \le \kappa_2\cdot\Phi(t_0)$.
- $(1-\alpha)\cdot y(t)^\top \boldsymbol{P}x(t) - z(t)^\top \boldsymbol{C}x(t) \le \kappa_3\cdot\Phi(t_0)$ *for all* $t\in\mathcal{T}$.
- $z(t)^\top \boldsymbol{C}x(t) - (1+\alpha)\cdot y(t)^\top \boldsymbol{P}x(t) \le \kappa_4\cdot\Phi(t_0)$ *for all* $t\in\mathcal{T}$.

Here $\kappa_1,\kappa_2,\kappa_3,\kappa_4 = \theta(\epsilon')$ are small constants. An interval that is not stationary is called unstationary.

Lemma 6 shows that the potential function decreases by a multiplicative factor in any unstationary interval, which in turn bounds the total number of unstationary intervals by a poly-logarithmic function. Its proof follows directly from Definition 1 and Lemma 5.

LEMMA 6. *In any unstationary interval, the potential Φ decreases by a factor of $\Omega(\epsilon'\cdot\min\{\alpha,\beta\mu\}) = \Omega(\epsilon'^2)$.*

Lemma 7 completes the proof. It states that all solutions computed in a "sufficiently" long stationary interval are $\theta(\epsilon')$-feasible. The proof is included below.

LEMMA 7. *Consider a stationary interval $\mathcal{T} = [t_0, t_1]$ where $t_0 \ge \tau_0$ and $t_1 - t_0 \ge \tau_1$ where $\tau_1 = O(\frac{1}{\beta}\log\frac{1}{\delta})$. Let x^0, y^0, z^0 denote the values of x, y, z at round t_0. Then x^0 forms a $\theta(\epsilon')$-feasible solution and, in particular, if $0 < \epsilon' \le 0.05$, x^0 is a $10\epsilon'$-feasible solution.*

Since $\epsilon' = \epsilon/10$, we obtain ϵ-feasibility for $\epsilon \le 0.5$.

PROOF. Assume to the contrary that the solution x^0 is not $\theta(\epsilon')$-feasible, e.g., that $\frac{\min_i \boldsymbol{C}_i x^0}{\max_i \boldsymbol{P}_i x^0} = \lambda \le 1 - 5\epsilon'$. Then from Lemma 4 and Def. 1, we have

$$\begin{aligned}
&(\max_i \boldsymbol{P}_i x^0 - \epsilon')\,(\mathbb{1}\cdot y^0) \\
&\le\ (1+\epsilon')\cdot (y^0)^\top \boldsymbol{P}x^0 \\
&\le\ \frac{1+\epsilon'}{1-\alpha}\cdot (z^0)^\top \boldsymbol{C}x^0 + (1+\epsilon')\cdot\kappa_3\cdot\Phi(t^0) \\
&\le\ \frac{1+\epsilon'}{1-\alpha}\cdot (\min_i \boldsymbol{C}_i x^0 + \epsilon')\cdot \mathbb{1}\cdot z^0 + (1+\epsilon')\cdot\kappa_3\cdot\Phi(t^0) \\
&\le\ (1+\frac{5}{3}\epsilon')\cdot (\min_i \boldsymbol{C}_i x^0 + \epsilon')\cdot \mathbb{1}\cdot z^0 + (1+\epsilon')\cdot\kappa_3\cdot\Phi(t^0).
\end{aligned}$$

Now from Lemma 3, it follows that $\max_i \boldsymbol{P}_i x^0 \ge 1 - 2\epsilon'$. Therefore

$$\begin{aligned}
\mathbb{1}\cdot y^0 &\le \Big(1+\frac{5}{3}\epsilon'\Big)\cdot\frac{\lambda\cdot\max_i \boldsymbol{P}_i x^0 + \epsilon'}{\max_i \boldsymbol{P}_i x^0 - \epsilon'}\cdot(\mathbb{1}\cdot z^0) \\
&\quad + \frac{1+\epsilon'}{\max_i \boldsymbol{P}_i x^0 - \epsilon'}\cdot\kappa_3\cdot\Phi(t^0) \\
&\le (1-\epsilon')(\mathbb{1}\cdot z^0) + O(\kappa_3)\cdot\Phi(t^0).
\end{aligned}$$

Since y and z have low mileage in the interval $\mathcal{T}$, we have

$$\begin{aligned}
&\mathbb{1}\cdot y^0 + \textstyle\sum_{t\in\mathcal{T}}\sum_i |\Delta y_i(t)| \\
&\le\ (1-\epsilon')\cdot\big(\mathbb{1}\cdot z^0 - \textstyle\sum_{t\in\mathcal{T}}\sum_i |\Delta z_i(t)|\big) \\
&\quad + O(\kappa_1+\kappa_2+\kappa_3)\cdot\Phi(t^0).
\end{aligned}$$

Let $\kappa_{123} = \kappa_1+\kappa_2+\kappa_3$. Since $\Phi(t^0) = \mathbb{1}\cdot y^0 + \mathbb{1}\cdot z^0$, we have

$$\begin{aligned}
&(1-O(\kappa_{123}))\cdot\big(\mathbb{1}\cdot y^0 + \textstyle\sum_{t\in\mathcal{T}}\sum_i |\Delta y_i(t)|\big) \\
&\le\ (1-\epsilon'+O(\kappa_{123}))\cdot\big(\mathbb{1}\cdot z^0 - \textstyle\sum_{t\in\mathcal{T}}\sum_i |\Delta z_i(t)|\big).
\end{aligned}$$

Thus

$$\frac{\mathbb{1}\cdot y^0 + \sum_{t\in\mathcal{T}}\sum_i |\Delta y_i(t)|}{\mathbb{1}\cdot z^0 - \sum_{t\in\mathcal{T}}\sum_i |\Delta z_i(t)|} \le 1-\epsilon'+O(\kappa_{123}) \le 1-\alpha.$$

Since the given mixed LP is feasible, there exists $x^* \ge 0$ such that $\boldsymbol{P}x^* \le \mathbb{1}$ and $\boldsymbol{C}x^* \ge \mathbb{1}$. Thus we have

$$\frac{(y^0)^\top \boldsymbol{P}x^* + \sum_{t\in\mathcal{T}}\sum_i |\Delta y_i(t)|\cdot \boldsymbol{P}x^*}{(z^0)^\top \boldsymbol{C}x^* - \sum_{t\in\mathcal{T}}\sum_i |\Delta z_i(t)|\cdot \boldsymbol{C}x^*} \le 1-\alpha.$$

Using an averaging argument, it follows that there exists j such that

$$\frac{\boldsymbol{P}_j^\top y^0 + \sum_{t\in\mathcal{T}}\sum_i |\boldsymbol{P}_j^\top y(t+1) - \boldsymbol{P}_j^\top y(t)|}{\boldsymbol{C}_j^\top z^0 - \sum_{t\in\mathcal{T}}\sum_i |\boldsymbol{C}_j^\top z(t+1) - \boldsymbol{C}_j^\top z(t)|} \le 1-\alpha.$$

Therefore for this j, and for all $t\in\mathcal{T}$, we have

$$\boldsymbol{P}_j^\top y(t) \,/\, \boldsymbol{C}_j^\top z(t) \le 1-\alpha.$$

Our algorithm will increase this variable by a factor of β in each round. Therefore after τ_1 rounds, its value increases so much that the potential Φ becomes larger than $(m+k)\cdot\exp(2\mu)$, contradicting Lemma 3. Therefore we can conclude that the solution x^0 is indeed $\theta(\epsilon')$-feasible.

To establish the final assertion, observe that $\frac{\min_i \boldsymbol{C}_i x^0}{\max_i \boldsymbol{P}_i x^0} > 1 - 5\epsilon'$ (proved above). Let $\epsilon' \le 0.05$. From Lemma 3, we obtain $\min_i \boldsymbol{C}_i x^0 > 1 - 7\epsilon'$ and $\max_i \boldsymbol{P}_i x^0 < 1 + 10\epsilon'$, which ensures $10\epsilon'$-feasibility. ■