A Distributed Approximation Algorithm for Mixed Packing-Covering Linear Programs

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

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

Abstract

Mixed packing-covering linear programs capture a simple but expressive subclass of linear programs. They commonly arise as linear programming relaxations of a number important combinatorial problems, including various network design and generalized matching problems. In this paper, we propose an efficient distributed approximation algorithm for solving mixed packing-covering problems which requires a poly-logarithmic number of passes over the input. Our algorithm is well-suited for parallel processing on GPUs, in shared-memory architectures, or on small clusters of commodity nodes. We report results of a case study for generalized bipartite matching problems.

1 Introduction

Linear programs (LP) that contain only *non-negative* coefficients and *non-negative* variables are called *mixed packing-covering LPs* (MPC-LP). These problems constitute a "simple" but still expressive subclass of LPs. MPC-LPs commonly arise as LP relaxations of a number of important combinatorial problems and therefore play a crucial role in designing approximation algorithms for such problems. Examples include various multicommodity flow problems [1, 2, 3, 4, 5], min-max/max-min resource sharing problems [6, 7, 8, 9], and generalized bipartite matching [10] (for which a provably good integer solution can be found by rounding the solution to the corresponding LP relaxation).

MPC programs can be solved exactly using standard solvers. However, these solvers cannot cope with very large problem instances that may occur in practice. Parallel approaches, such as algorithms for shared-memory [11, 12, 13] or shared-nothing [12, 14, 10] architectures, are essential for achieving reasonable performance at massive scales.

In this paper, an extended abstract of [10], we develop a parallel approximation algorithm for MPC-LP problems. These problems have general form

$$\max\{w^\top x : \boldsymbol{P}x \le p, \boldsymbol{C}x \ge c, x \ge 0\}, \qquad \text{(MPC-LP)}$$

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*, $p \in \Re^m_+$ the corresponding right-hand side, $\boldsymbol{C} \in \Re^{k\times n}_+$ a *covering-constraint matrix*, and $c \in \Re^k_+$ the corresponding right-hand side. We consider an approximate variant of MPC-LP, in which we seek for a near-optimal solution and additionally allow for a small violation of packing and covering constraints. This "relaxation" allows us to develop a scalable distributed approximation algorithm. In particular, denote by $\epsilon > 0$ a small *error bound*. We say that $x \in \Re^n_+$ is an *ϵ-feasible* solution to MPC-LP if packing and covering constraints are violated by at most a factor of $1-\epsilon$ and $1+\epsilon$, respectively, i.e., $\boldsymbol{P}x \le (1+\epsilon)p$ and $\boldsymbol{C}x \ge (1-\epsilon)c$. For $0 < \eta < 1$, an

Algorithm 1 MPCSolver for MPC-LPs (feasibility)

```
Require: $\boldsymbol{P} \in \Re_+^{m\times n}$, $p \in \Re_+^m$, $\boldsymbol{C} \in \Re_+^{k\times n}$, $c \in \Re_+^k$, $\epsilon$
 1: $\mu \leftarrow \ln(mkM/\epsilon)/\epsilon$ {scaling constant}
 2: $\alpha \leftarrow \epsilon/4$ {update threshold}
 3: $\beta \leftarrow \alpha/(20\mu)$ {multiplicative step size}
 4: $\delta \leftarrow \beta/nM$ {additive increase}
 5: Start with any $x \in \Re_+^n$
 6: repeat
 7:     Compute $y_i(x) = \exp\big[\mu \cdot \big(\boldsymbol{P}_i x - 1\big)\big]$ for $i = 1, \ldots, m$
 8:     Compute $z_i(x) = \exp\big[\mu \cdot \big(1 - \boldsymbol{C}_i x\big)\big]$ for $i = 1, \ldots, k$
 9:     for $j = 1, \ldots, n$ do
10:         if $\frac{\boldsymbol{P}_j^\top y(x)}{\boldsymbol{C}_j^\top z(x)} \leq 1 - \alpha$ then
11:             $x_j \leftarrow \max\{x_j(1+\beta), \delta\}$
12:         if $\frac{\boldsymbol{P}_j^\top y(x)}{\boldsymbol{C}_j^\top z(x)} \geq 1 + \alpha$ then
13:             $x_j \leftarrow x_j(1-\beta)$
14: until convergence
```

ϵ-feasible solution is called an (ϵ, η)-*approximation* if it obtains an objective value within a factor of $(1-\eta)(1-\epsilon)$ of the optimal solution to MPC-LP. In what follows, we present a distributed algorithm called *MPCSolver* that obtains such an approximation for feasible MPC-LPs. Our algorithm performs only a poly-logarithmic number of passes over the data (in input size and the width of the LP), is simple and efficient in practice, and can be readily implemented on GPUs, shared-memory and shared-nothing architectures, as well as on MapReduce.

2 Solving Mixed Packing-Covering Linear Programs

We first discuss how to solve MPC-LP feasibility problems, i.e., ignoring the value of the objective function, and then proceed to the optimization version.

Feasibility. MPCSolver, given as Alg. 1, is inspired by the work of Awerbuch and Khandekar [14] and can be seen as a gradient descent method with multiplicative updates. To simplify our discussion, we first describe our algorithm in a centralized environment and then generalize to the distributed setting. 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 the all-ones 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.[1] Denote by M the largest entry in $\boldsymbol{P}$ and $\boldsymbol{C}$. We aim to find a vector $x \geq 0$ such that $\boldsymbol{P}x \leq \mathbb{1}$ and $\boldsymbol{C}x \geq \mathbb{1}$.

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 $z(x) = (z_1, \ldots, z_k)^\top \in \Re_+^k$ be given as follows:

$$\begin{aligned} y_i(x) &= \exp\big[\mu \cdot \big(\boldsymbol{P}_i x - 1\big)\big] \\ z_i(x) &= \exp\big[\mu \cdot \big(1 - \boldsymbol{C}_i x\big)\big], \end{aligned}$$

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. One may think of $y_i(x)$ as an exponential "penalty" function of packing constraint $\boldsymbol{P}_i x \leq 1$. Penalty $y_i(x)$ is small if the constraint is satisfied ($y_i(x) \leq 1$ if $\boldsymbol{P}_i x \leq 1$) and large otherwise ($y_i(x) > 1$ if $\boldsymbol{P}_i x > 1$). Similarly, $z_i(x)$ is a penalty function for covering constraint $\boldsymbol{C}_i x \geq 1$. In what follows, we refer to $y_i(x)$ and $z_i(x)$ 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

[1] This can be achieved by appropriate rescaling of the rows and the columns of the coefficient matrices.

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

All parameters of MPCSolver depend on ϵ. Here α acts as an *update threshold*, β as a multiplicate *step size*, and δ as an *additive increase*. 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. 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. Moreover, updates are multiplicative: We add or subtract βx_j from x_j. Step size parameter β determines how quickly we move through the parameter space. 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. The algorithm iterates until convergence (see below); we then obtain an approximate minimizer of Φ.

Runtime. If the given MPC-LP is feasible, MPCSolver produces an ϵ-feasible solution in $\widetilde{O}(\,\epsilon^{-5} \ln^3(kmMnx_{max})\,)$ 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. Here $\widetilde{O}$ hides lower-order terms. MPCSolver thus converges fast in the sense that it is guaranteed to find an ϵ-feasible solution after a poly-logarithmic number of passes over the data. Each pass can be parallelized as described below. See [10] for a full analysis of Alg. 1.

Convergence test. MPCSolver has converged as soon as one of the following criteria is met: (i) the solution is ϵ-feasible, (ii) all variables remain unmodified, or (iii) the number of rounds exceeds the poly-logarithmic bound. If (ii) or (iii) hold and the solution is not ϵ-feasible, the MPC-LP instance 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, which was shown to be effective in practice [10], is to stop MPCSolver when the decrease in potential stagnates; e.g., when it falls below some threshold in two consecutive rounds.

Parallel implementation. MPCSolver is straightforward to parallelize on a shared-memory architecture since both primal and dual variables are updated independently of each other. In what follows, we discuss a shared-nothing implementation. In each round, MPCSolver first computes the dual variables y and z associated with each packing or covering constraint, respectively. To distribute the computation, we conformingly partition $x^\top$ and partition matrices $\boldsymbol{P}$ and $\boldsymbol{C}$ column-wise across the cluster nodes. Denote by $\boldsymbol{P}_{(i)}$, $\boldsymbol{C}_{(i)}$, and $x_{(i)}$ the respective partitions stored at some node i. Each node computes the products $\boldsymbol{P}_{(i)}x_{(i)}$ and $\boldsymbol{C}_{(i)}x_{(i)}$ independently and in parallel. The partial results are then summed up locally and broadcast across the compute cluster to obtain $\boldsymbol{P}x$ and $\boldsymbol{C}x$ (e.g., using `MPI_Allreduce`). Each node now is able to compute the value of the duals and update the primal variables in partition $x_{(i)}$ independently and in parallel (lines 9–13). Since matrices $\boldsymbol{P}$ and $\boldsymbol{C}$ are accessed once row-wise (to compute the partial products) 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 only once before the algorithm starts, and that only the partial products are communicated. This implementation can be inefficient in terms of communication costs when there are many more constraints than variables; however, the communication costs can be reduced significantly by exploiting the sparsity of the constraint matrices. Moreover, we can exploit shared-memory parallel processing when multiple thread are available on each compute node.

From feasibility to optimization. To optimize the objective function, we adapt a technique of Young [12] to our setting. The key idea is to push the objective function into the constraints. More precisely, we deal with the following optimization problem: find $\lambda^* = \max\{\lambda : (\exists x)\boldsymbol{P}x \leq p, \boldsymbol{C}x \geq c, w^\top x \geq \lambda\}$. 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 an estimate of λ^*, we run a (small) sequence of feasibility problems of form $\{\boldsymbol{P}x \leq p, \boldsymbol{C}x \geq c, w^\top x \geq \lambda\}$, where λ is determined via binary search. More precisely, we obtain an (ϵ, η)-approximation by running MPCSolver with an error bound of ϵ on $O(\log\log(\lambda_{\text{max}}/\lambda_{\text{min}}) - \log\log(1-\eta)^{-1})$ instances of such feasibility problems. Here λ_{min} denotes a lower bound on the optimal value of the pure covering problem $\min\{w^\top x : \boldsymbol{C}x \geq c, x \geq 0\}$, and λ_{max} refers to an upper bound on the optimal value of the pure packing problem $\max\{w^\top x : \boldsymbol{P}x \leq p, x \geq 0\}$. Our search strategy is closely related to the one of [12] but uses a fixed error bound; see [10] for details.

3 Case Study: Generalized Bipartite Matching

We report the results of a case study with instances of generalized bipartite matching (GBM) problems; see [10] for a description of the problem and additional details about the experimental setup. In GBM problems, we aim to find a maximum-weight matching (subset of the edges) of an undirected bipartite graph such that the degree of each node in the matching lies between a specified lower and upper bound. The LP relaxation of GBM is a mixed packing-covering linear program.

An attractive feature of MPCSolver is its simplicity; it can be implemented in a few lines of code. Note that at each iteration, MPCSolver essentially computes a set of sparse matrix-vector products. Modern GPU architectures allow for massively parallel computation of such products, and GPU implementations can outperform CPU implementations by orders of magnitude. In our case study, we make use of a C++ implementation for CPUs as well as a CUDA implementation for GPUs using CUSPARSE [15] for matrix-vector products. We ran our C++ implementation on a compute node equipped with an Intel Xeon 2.40GHz processor with 8 cores and 48GB main memory, and our CUDA implementation on a NVIDIA GeForce GTX TITAN GPU with 6GB device memory.

For our experiments, we used two real datasets: Movielens10M consisting of 10M movie ratings of 72K users over 10k items, and NetflixTop50 consisting of 24M movie ratings of 480k users over 18k items. To create the latter dataset, we first applied the rating prediction algorithm of [16] to predict unknown ratings in Netflix matrix, and then computed the set of items with the top-50 largest predicted ratings for each user. To construct a GBM instance, we converted each dataset into a bipartite graph in which users and items form vertices and ratings correspond to weighted edges; our goal was to match (i.e., recommend) items to users. To create feasible instances, the following bounds were chosen: We used a lower bound of 3 and an upper bound of 5 on the degree of each user in the matching (i.e., the number of recommendations given to each user). For items, we did not enforce a lower bound but chose an upper bound of 200 (i.e., each item to be recommended at most 200 times). On both datasets, we applied MPCSolver to obtain a 0.05-feasible solution.

In Table 1, we report results of the performance of MPCSolver on the Movielens10M and NetflixTop50 datasets. Compared to a sequential CPU implementation, MPCSolver obtained a 38.6x and 26x speed up on NetflixTop50 and Movielens10M, respectively, when running on GPU. We also compared MPCSolver to Young's algorithm [12]. On both datasets and when running with 8 threads, Young's algorithm required a few hours to obtain a 0.05-feasible solution. In the same setup, MPCSolver achieved the desired precision within 9.8 and 3.7 minutes, respectively. Results of additional experiments on larger datasets are given in [10].

Table 1: Experimental results on Movielens10M and NetflixTop50 (1x1 refers to 1 node running 1 thread, 1x8 refers to 1 node running 8 threads).

Dataset	Average time/iteration (ms)			Time for 0.05-feasibility (s)		
	1x1	1x8	GPU	1x1	1x8	GPU
Movielens10M	727	250	**28**	652	223	**25**
NetflixTop50	2513	750	**65**	1971	588	**51**

4 Conclusion

We presented MPCSolver, a distributed algorithm for approximately solving mixed packing-covering linear programs. MPCSolver is easy to implement and parallelize and requires only a poly-logarithmic number of passes over the data. Our experiments indicate fast convergence and good scalability.

References

[1] Farhad Shahrokhi and David W. Matula. The maximum concurrent flow problem. *Journal of the ACM*, 37(2):318–334, 1990.

[2] Philip N. Klein, Serge A. Plotkin, Clifford Stein, and Éva Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, 1994.

[3] Frank Thomson Leighton, Fillia Makedon, Serge A. Plotkin, Clifford Stein, Éva Stein, and Spyros Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Sciences*, 50(2):228–243, 1995.

[4] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[5] George Karakostas. Faster approximation schemes for fractional multicommodity flow problems. *ACM Transactions on Algorithms*, 4(1), 2008.

[6] Michael D. Grigoriadis and Leonid G. Khachiyan. Fast approximation schemes for convex programs with many blocks and coupling constraints. *SIAM J. Optimization*, 4(1):86–107, 1994.

[7] Michael D. Grigoriadis and Leonid G. Khachiyan. Coordination complexity of parallel price-directive decomposition. *Mathematics of Operations Research*, 21:321–340, 1996.

[8] Michael D. Grigoriadis, Leonid G. Khachiyan, Lorant Porkolab, and J. Villavicencio. Approximate max-min resource sharing for structured concave optimization. *SIAM Journal on Optimization*, 11(4):1081–1091, 2001.

[9] Klaus Jansen and Hu Zhang. Approximation algorithms for general packing problems with modified logarithmic potential function. In *IFIP TCS*, pages 255–266, 2002.

[10] Faraz Makari Manshadi, Baruch Awerbuch, Rainer Gemula, Rohit Khandekar, Julián Mestre, and Mauro Sozio. A distributed algorithm for large-scale generalized matching. *PVLDB*, 6(9):613–624, 2013.

[11] Michael Luby and Noam Nisan. A parallel approximation algorithm for positive linear programming. In *STOC*, pages 448–457, 1993.

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

[13] Christos Koufogiannakis and Neal E. Young. Beating simplex for fractional packing and covering linear programs. In *FOCS*, pages 494–504, 2007.

[14] Baruch Awerbuch and Rohit Khandekar. Stateless distributed gradient descent for positive linear programs. *SIAM Journal of Computing*, 38(6):2468–2486, 2009.

[15] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC*, 2009.

[16] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77, 2011.