Efficient Parallel Implementation of the Multiplicative Weight Update Method for Graph-based Linear Programs

CALEB JU, Georgia Institute of Technology
SERIF YESIL, University of Illinois at Urbana-Champaign
MENGYUAN SUN, University of Illinois at Urbana-Champaign
CHANDRA CHEKURI, University of Illinois at Urbana-Champaign
EDGAR SOLOMONIK, University of Illinois at Urbana-Champaign

Positive linear programs (LPs) model many graph and operations research problems. One can solve for a $(1+\epsilon)$-approximation for positive LPs, for any selected ϵ, in polylogarithmic depth and near-linear work via variations of the multiplicative weight update (MWU) method. Despite extensive theoretical work on these algorithms through the decades, their empirical performance is not well understood.

In this work, we implement and test an efficient parallel algorithm for solving positive LP relaxations, and apply it to graph problems such as densest subgraph, bipartite matching, vertex cover and dominating set. We accelerate the algorithm via a new step size search heuristic. Our implementation uses sparse linear algebra optimization techniques such as fusion of vector operations and use of sparse format. Furthermore, we devise an implicit representation for graph incidence constraints. We demonstrate the parallel scalability with the use of threading OpenMP and MPI on the Stampede2 supercomputer. We compare this implementation with exact libraries and specialized libraries for the above problems in order to evaluate MWU's practical standing for both accuracy and performance among other methods. Our results show this implementation is faster than general purpose LP solvers (IBM CPLEX, Gurobi) in all of our experiments, and in some instances, outperforms state-of-the-art specialized parallel graph algorithms.

ACM Reference Format:
Caleb Ju, Serif Yesil, Mengyuan Sun, Chandra Chekuri, and Edgar Solomonik. 2024. Efficient Parallel Implementation of the Multiplicative Weight Update Method for Graph-based Linear Programs. 1, 1 (February 2024), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designing general graph libraries for massively-parallel machines is challenging as combinatorial graph algorithms often have limited concurrency and arithmetic intensity [30]. Many have parallel depth at least proportional to graph diameter, though in practice these algorithms contain sufficient concurrency to achieve high-efficiency on shared-memory machines [17, 33, 40, 41]. Designing efficient distributed-memory parallelizations is more challenging, though it has been achieved with use of graph partitioning [43], formulations via sparse matrix products [12, 13, 42].

Another method for solving graph problems is to reformulate or relax [46, 48] the graph problem into a linear program (LP). Recent theoretical breakthroughs in both sequential and parallel algorithms for several graph problems [5, 26, 28, 39, 44, 45] indicate that carefully designed LP solvers can enable algorithms with tunable accuracy that have lower cost and depth than their combinatorial counterparts. Given a matrix $\boldsymbol{A} \in \mathbf{R}^{m\times n}$, vector $\boldsymbol{c} \in \mathbf{R}^n$, and vector $\boldsymbol{b} \in \mathbf{R}^m$, an LP is the optimization problem,

$$\max_{\boldsymbol{x}\in\mathbf{R}^n} \langle \boldsymbol{c}, \boldsymbol{x}\rangle \text{ s.t. } \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \tag{1}$$

where $\langle \boldsymbol{c}, \boldsymbol{x}\rangle = c_1x_1 + c_2x_2 + \ldots + c_nx_n$, and $\boldsymbol{u} \leq \boldsymbol{v}$ means $u_i \leq v_i\ \forall i$. [46, 48].

Typically, to solve a general LP, one employs variants of the simplex or interior point methods. While these algorithms are efficient in the sequential setting, they often have limited parallelism. LP solving is P-Complete, which implies that a poly-logarithmic depth parallel algorithm is believed unlikely to exist [23].

However, many graph problems can be solved via LPs that contain only positive entries, and these class of LPs admit efficient parallelizable solvers if some approximation is allowed. These approaches yield a parallel algorithm with depth that is a polynomial in $1/\epsilon$ and $\log n$ (the number of variables), and is independent of the structure of the constraint matrices $\boldsymbol{P}$ and $\boldsymbol{C}$. It outputs a result that is a $(1+\epsilon)$-approximation, which, for a maximization problem entails a relative error of $(1-\epsilon)$ in the objective value, and for a minimization problem, a relative error of $(1+\epsilon)$.

The polynomial scaling with $1/\epsilon$ has been improved in recent work. Our implementation is based on an accelerated version of a more recent algorithm with an improved depth of $\tilde{O}(\epsilon^{-3})$ for mixed problems and $\tilde{O}(\epsilon^{-2})$ for pure covering or pure packing problems. The only other previously published study [32] of a distributed and parallel LP solver built using this approach implemented and compared run times of two earlier algorithms: MPCSolver and Young's algorithm, with $\tilde{O}(\epsilon^{-5})$ and $\tilde{O}(\epsilon^{-4})$ depth, respectively. Further details on theoretical developments in approximately solving positive LPs in parallel are provided in Section 2.

The algorithm we focus on is from Mahoney et. al. [31], which offers the fastest theoretical performance for pure packing, pure covering, and mixed packing and covering LPs in a parallel setting. We describe these problems and their associated graph problems formally in Section 3. This algorithm we call MWU since it utilizes the weights vector from the more general MWU method [6]. Despite the algorithm's strong theoretical foundation, little is known about its empirical performance, especially for solving real-world graph problems over large datasets. Furthermore, understanding how this algorithm compares to other LP solvers and specialized graph libraries is an open question.

In this paper, we create a practical and scalable implementation of a parallel $(1+\epsilon)$-approximate positive LP solver by adapting the

Authors' addresses: Caleb Ju, cju33@illinois.edu, Georgia Institute of Technology; Serif Yesil, University of Illinois at Urbana-Champaign, Atlanta, GA; Mengyuan Sun, ms65@illinois.edu, University of Illinois at Urbana-Champaign; Chandra Chekuri, chekuri@illinois.edu, University of Illinois at Urbana-Champaign; Edgar Solomonik, solomon2@illinois.edu, University of Illinois at Urbana-Champaign.

2024. XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

MWU algorithm with the current fastest theoretical performance, and we also provide a comprehensive empirical study comparing MWU against exact solvers, specialized libraries, and previous related work. We also test a line search method that finds the largest step-size permitted without violating theoretical guarantees. This step-size search reduces in the number of overall MWU iterations in practice by multiple orders of magnitude. Our results suggest that MWU can be fast in both theory and practice, with run times comparable to even specialized libraries but with the added benefit of, one, being generalizable and, two, more versatile, as it can benefit from high-performance sparse linear algebra libraries. We believe this marks MWU as a viable alternative for solving graph problems approximately on a large dataset. Overall, our paper makes the following contributions:

- the first shared and distributed-memory implementation of a general-purpose approximate solver based on Mahoney et. al. [31] for positive LPs, called MWU
- a step size search method for MWU that empirically reduces the iteration count by up to three orders of magnitude without adding significant overhead
- an efficient implementation of MWU using an implicit representation of common constraint matrices observed in graph LPs as well as other standard techniques for SLA. We show these optimizations provide good scalability and lead up to 5.2x speedup relative to MWU implemented with PETSc (a library for parallel sparse linear algebra).
- the first comparison of an approximate positive LP solver against general LP solvers as well as specialized parallel graph algorithms. In particular, MWU finds $(1+\epsilon)$-relative ($\epsilon = 0.1$) solutions up to 3-2800x and 5-1180x faster than CPLEX and Gurobi (which find exact solutions for implicit ILPs and exact fractional solutions for relaxed LPs), respectively, for solving several graph problems on large real-world graphs on the Stampede2 supercomputer using KNL compute nodes.

2 BACKGROUND ON LINEAR PROGRAM SOLVERS

2.1 Positive LP Solvers

Fast approximate solvers for positive LPs in the sequential settings have been developed since the early 1990's [20, 35], and there is extensive and continued attention to this line of work. We mostly focus on parallel algorithms and refer the reader to [36, 47] for extensive pointers. Luby and Nisan provide the first parallel algorithm for explicit positive LPs which obtain a $(1+\epsilon)$-approximation in $\tilde{O}(\epsilon^{-4})$ iterations[1] for pure packing and covering LPs [29]. Young clarified and extended this work to solve the more general mixed packing and covering LPs in parallel in $\tilde{O}(\epsilon^{-4})$ iterations [49]. The dependence on ϵ has remained unchanged for over 10 years until the work of Allen-Zhu and Orecchia, who solve pure packing and pure covering LPs in parallel with $\tilde{O}(\epsilon^{-3})$ iterations [3]. They also obtained better dependence on ϵ in the sequential setting [4] for pure packing and covering LPs (see also [31]).

[1]We write $\tilde{O}(f(n))$ to be proportion to $f(n)$ and a polylogarithmic of $f(n)$, i.e., $\tilde{O}(f(n)) \propto O(f(n)\log^{O(1)}(f(n)))$

Recently, Mahoney et. al. [31] utilized ideas of Young [50] on faster near-linear time sequential solver to develop new parallel algorithms for positive LPs. For mixed packing and covering LPs, their algorithm converges in $O\big(\log(m_p + m_c)\log(n/\epsilon)/\epsilon^3\big)$ iterations and is also work-efficient, i.e., work is near linear to the number of nonzeros in the LP. For pure packing and pure covering LPs, the dependence on ϵ is ϵ^{-2} instead. When ϵ is not too small, these algorithms have low depth in the PRAM model.

Empirical studies of these fast approximate algorithms have been mainly limited to relatively small problems. Koufogiannakis and Young adapt a sequential mixed packing and covering LP and show it outperforms simplex on randomly generated binary matrices with dimension up to 2222 [25]. Allen-Zhu and Orecchia compare their $\tilde{O}(\epsilon^{-1})$-dependent sequential algorithm [2] to the algorithms of Luby and Nisan [29] and Awerbuch and Khandekar [7] for solving pure packing LPs with a randomly generated matrix of size 60×40. Jelic et. al. implement a parallel primal-dual method on GPUs to solve positive LPs, although their constraint matrices are randomly generated binary matrices with dimensions up to 25000 [22]. The most closely related work to ours is that of Makari et. al. [32], who implement a gradient descent algorithm to solve generalized matching on large real-world and synthetic graphs. Their implementation, based on the algorithm from [7], has a $\tilde{O}(1/\epsilon^5)$ number of iterations, and they confine their attention to a single graph problem.

In this paper we focus on only obtaining fractional solutions to the LP problems. Since all LPs are polynomial time solvable, whereas the discrete formulation of some of the graph problems we consider are NP-hard, this allows us to have a more uniform and fair comparison to prior art (e.g., [32], as well as state-of-the-art software for solving LPs like Gurobi).

There also exists rounding techniques (which convert a fractional solution to an integral one) or specialized algorithms to solve or approximately solve the discrete problems, but these methods are specific to the problem and have different levels of parallelism, efficiency, and approximation guarantees. For example, an exact parallel rounding technique for a maximum matching problem is implemented in [32], but the run time can be three orders of magnitude longer than solving the LP problem. On the other hand, rounding techniques for dominating set are compared in [27], but these only return approximately optimal solutions and the rounding is not parallelized. There are also specialized libraries for the aforementioned graph problems [8, 18] (more details can be found in Section 6). But again, these implementations are tailored to the problem at hand, hence it would not be fair to compare the run time against our general-purpose solver in terms of obtaining an integral solution.

2.2 The MWU Algorithm

We now introduce MWU, the algorithm of Mahoney et. al. [31] for approximately solving the standard mixed packing and covering LP in parallel. This section does not contain our modifications for improving the empirical performance, which are found in Section 4.

First, we start describing how MWU solves the mixed packing and covering feasibility LP [31, 49]. We will discuss how to modify this into an optimization problem later in the section. The feasibility

LP is:

$$\exists\, \boldsymbol{x} \in \mathbf{R}^n \text{ s.t. } \boldsymbol{P}\boldsymbol{x} \leq \mathbb{1}, \boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \tag{2}$$

where $\boldsymbol{P}$ and $\boldsymbol{C}$ are *nonnegative*. Vectors $\mathbb{1}$ and $\mathbb{0}$ are the all ones and zeros vector, respectively. The algorithms we consider seek a $(1+\epsilon)$-relative approximation: that is, a solution $\boldsymbol{x}$ such that $\boldsymbol{P}\boldsymbol{x} \leq (1+\epsilon)\mathbb{1}$ and $\boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}$.

MWU (Algorithm 1) ensures that both packing and convering constraints, $\boldsymbol{P}\boldsymbol{x} \leq \mathbb{1}$ and $\boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}$, are approximately satisfied by approximating $\max(\boldsymbol{P}\boldsymbol{x})$ and $\min(\boldsymbol{C}\boldsymbol{x})$ with smoothed maximum and minimum functions,

$$\text{smax}_\eta(\boldsymbol{x}) = \frac{1}{\eta}\log\Big(\sum_{i=1}^{n}\exp(\eta \cdot x_i)\Big),$$

$$\text{smin}_\eta(\boldsymbol{x}) = -\frac{1}{\eta}\log\Big(\sum_{i=1}^{n}\exp(-\eta \cdot x_i)\Big),$$

where $\eta > 2$ is a smoothing parameter. The MWU algorithm and step size search make use of their gradients,

$$\nabla\text{smax}_\eta(\boldsymbol{x}) = \frac{\exp(\eta \cdot \boldsymbol{x})}{\langle \mathbb{1}, \exp(\eta \cdot \boldsymbol{x})\rangle}, \nabla\text{smin}_\eta(\boldsymbol{x}) = \frac{\exp(-\eta \cdot \boldsymbol{x})}{\langle \mathbb{1}, \exp(-\eta \cdot \boldsymbol{x})\rangle}.$$

For more details on these functions, see Chapter 2 of [36].

Algorithm 1 Multi-Update MWU Method for Mixed Packing and Covering LPs

1: **procedure** MWU($\boldsymbol{P} \in \mathbf{R}_+^{m_P \times n}$, $\boldsymbol{C} \in \mathbf{R}_+^{m_C \times n}$, ϵ)
2: $\eta \leftarrow 10\log(m)/\epsilon$ where $m := m_P + m_C$
3: $x_i \leftarrow \frac{\epsilon}{n\|\boldsymbol{P}_{:,i}\|_\infty}\ \forall i \in [n]$ ▷ $\|\boldsymbol{x}\|_\infty = \max_i |x_i|$
4: **while** constraints not approximately satisfied and $\boldsymbol{C} \neq \emptyset$ **do**
5: $\boldsymbol{g} \leftarrow \boldsymbol{P}^\mathsf{T}\nabla\text{smax}_\eta(\boldsymbol{P}\boldsymbol{x})$
6: $\boldsymbol{h} \leftarrow \boldsymbol{C}^\mathsf{T}\nabla\text{smin}_\eta(\boldsymbol{C}\boldsymbol{x})$
7: $d_i \leftarrow \frac{1}{2\eta}\max\{0, 1 - \frac{g_i}{h_i}\} \cdot x_i\ \forall i$
8: **if** $\max(\boldsymbol{d}) = 0$ **then**
9: Return "INFEASIBLE"
10: $\boldsymbol{x} \leftarrow \boldsymbol{x} + \boldsymbol{d}$
11: $\boldsymbol{C} \leftarrow \{c_i\ :\ c_i^T x < 1\}$ ▷ Keep unsatisfied constraints
12: **return** x

The algorithm initializes the vector $\boldsymbol{x}$ with small values so that each starting packing constraint is at most ϵ (Line 3). The smoothing parameter η is set so that both smax_η and smin_η are within an ϵ additive error of max and min, respectively. In each MWU iteration, the algorithm multiplicatively updates $\boldsymbol{x}$. This is done by defining a step or update vector, $\boldsymbol{d}$, where d_i is a multiple of x_i (Line 7), and adding $\boldsymbol{d}$ to $\boldsymbol{x}$ (Line 10). Vectors $\boldsymbol{g}$ and $\boldsymbol{h}$, which are gradients of the smoothed max packing and min covering constraints, respectively, are also utilized to define $\boldsymbol{d}$ (Lines 5, 6). In particular, $\boldsymbol{d}$ is an approximate solution to the Lagrangian relaxation,

$$\begin{aligned}\exists \boldsymbol{d} \in \mathbf{R}^n_{\geq 0} \text{ s.t. } \langle \boldsymbol{w}_p, \boldsymbol{P}\boldsymbol{d}\rangle = \langle \boldsymbol{P}^T\boldsymbol{w}_p, \boldsymbol{d}\rangle \leq 1,\\ \langle \boldsymbol{w}_c, \boldsymbol{C}\boldsymbol{d}\rangle = \langle \boldsymbol{C}^T\boldsymbol{w}_c, \boldsymbol{d}\rangle \geq 1,\end{aligned} \tag{3}$$

where $\boldsymbol{w}_p = \nabla\text{smax}_\eta(\boldsymbol{P}x)$ and $\boldsymbol{w}_c = \nabla\text{smin}_\eta(\boldsymbol{C}x)$.

If the positive LP is infeasible, then there exists some MWU iterations where $\boldsymbol{d} = \mathbb{0}$, in which case the algorithm reports the LP is infeasible (Line 8) [31]. Assuming otherwise, the theoretical analysis guarantees MWU will return an $(1+\epsilon)$-relative solution. Throughout the algorithm, we drop satisfied covering constraints, as these can unnecessarily slow down progress (Line 11). Finally, the algorithm returns $\boldsymbol{x}$ when all the covering constraints are satisfied or when there exists no covering constraints.

We note that Algorithm 1 can also solve pure packing or pure covering LPs, which are, respectively,

$$\max \langle \mathbb{1}, \boldsymbol{x}\rangle \text{ s.t. } \boldsymbol{P}\boldsymbol{x} \leq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \boldsymbol{x} \in \mathbf{R}^n$$

$$\min \langle \mathbb{1}, \boldsymbol{x}\rangle \text{ s.t. } \boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \boldsymbol{x} \in \mathbf{R}^n.$$

For example, to solve a pure packing LP, we embed the objective function as the added constraint, $\frac{1}{M}\mathbb{1}^T\boldsymbol{x} \geq 1$, where M is the estimate of the maximum value, e.g., $M = \sum_{i=1}^{n}\max_{j:p_{ji}>0} 1/p_{ji}$. Then we do binary search over M, using Algorithm 1 to determine if the resulting mixed packing and covering LP is a feasible. Since there is one covering constraint, then $\text{smin}_\eta(\boldsymbol{C}\boldsymbol{x}) = \min(\boldsymbol{C}\boldsymbol{x})$. This exact approximation permits one to scale the step direction (Line 7) by a factor of 2 in the theoretical analysis, which improves the number of iterations by a factor of ϵ [31]. Also, noting $\boldsymbol{C} = \frac{1}{M}\mathbb{1}^T$ and $\nabla\text{smin}_\eta(\boldsymbol{C}\boldsymbol{x}) = 1$, we have $\boldsymbol{h} = \frac{1}{M}\mathbb{1}$ (Line 6), so we do not need to explicitly compute $\boldsymbol{h}$. Solving a pure covering LP is done via a similar transformation. Solving a mixed covering and packing optimization problem, likewise, involves embedding the constraint that corresponds to the direction of optimization.

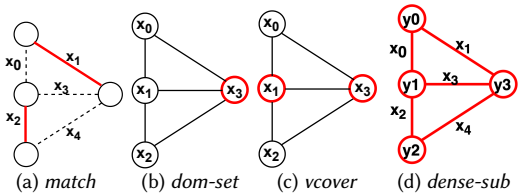
3 GRAPH PROBLEMS AS POSITIVE LPS

We now consider several graph problems. We first define integer programming (IP) formulations which exactly model the underlying graph problem. We then obtain an LP by relaxing the integrality constraints. For some problems, such as bipartite matching and densest subgraph, the solution to the LP relaxation matches the IP's solution (i.e., the solution is integral), whereas for NP-hard problems dominating set and vertex cover there is an *integrality gap* between the solution to the LP relaxation and IP. See [46, 48] for the role of LP relaxations in the development of approximation algorithms, and also [27] for a performance study on rounding an LP relaxation to an integral solution for dominating set. Our goal is to design a general-purpose solver, and the design and performance of rounding schemes are problem dependent (see the end of subsection 2.1 for further details). Therefore, we do not consider rounding in our implementation nor performance comparisons.

Let $G = (V, E)$ be an unweighted, undirected graph where V is the set of vertices and E is the set of edges, with $n = |V|$ and $m = |E|$. For simplicity, we assume G has no self-loops. Note that our formulations can be extended towards weighted graphs as well. For a vertex $v \in V$, let $N(v)$ be the neighbor vertices of v (v is not included in $N(v)$), and $\text{inc}(v)$ be the set of edges incident to v.

The neighbor relations between the vertices of G can be represented as an adjacency matrix, $\boldsymbol{A} \in \{0,1\}^{n\times n}$, which is symmetric and has a nonzero for each edge $e \in E$. The incidence relation between the vertices and the edges can be represented as a vertex-edge incidence matrix, $\boldsymbol{M}$, where

$$\boldsymbol{M}_{u,e} = \begin{cases} 1\ :\ u \in e, e \in E \\ 0\ :\ \text{otherwise}\end{cases},\ \boldsymbol{M} \in \{0,1\}^{n\times m}. \tag{4}$$



(a) *match* (b) *dom-set* (c) *vcover* (d) *dense-sub*

Fig. 1. Four graph problems run on the same graph. Variables of LP with a nonzero value are highlighted in red. An example of matching is given in Figure 1a. The set of edges in the matching are marked with thick red lines.

In this work, we consider four graph applications: *maximum matching*, *dominating set*, *vertex cover*, and *densest subgraph*.

Maximum Matching (*match*). A matching is a subset $F \subset E$ of edges such that no vertex is incident to more than one edge in F. We can write this optimization problem as the IP,

$$\max \sum_{e \in E} x_e \text{ s.t. } \sum_{e \in \text{inc}(v)} x_e \leq 1 \; \forall v \in V \quad x_e \in \{0,1\} \; \forall e \in E. \tag{5}$$

The x_e variables indicate whether edge e is in set F. The constraints are defined over the vertices of the graph such that at most one edge (x_e) in the incident edges of a vertex v ($\text{inc}(v)$) can be selected for matching set F. When the input graph is a bipartite graph, we call this problem **maximum bipartite matching** (*bmatch*).

Given the vertex-edge incidence matrix $\boldsymbol{M}$ of the graph, we can write the LP relaxation for maximum matching as the pure packing LP,

$$\max \langle \mathbb{1}, \boldsymbol{x} \rangle \text{ s.t. } \boldsymbol{M}\boldsymbol{x} \leq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \; \boldsymbol{x} \in \mathbf{R}^m. \tag{6}$$

It is well-known that this LP relaxation has no integrality gap for *bmatch* while for general graphs there is an integrality gap of 2/3; there is an exponential sized exact LP relaxation for general graph matching but we do not consider it here.

Dominating Set (*dom-set*). Dominating set is the problem of finding the smallest subset of vertices $S \subseteq V$ such that every vertex in the graph is either in S or is a neighbor of a vertex in S. We can formulate the *dom-set* problem as the IP,

$$\min \sum_{v \in V} x_v \text{ s.t. } x_v + \sum_{u \in N(v)} x_u \geq 1, \; \forall v \in V \quad x_v \in \{0,1\} \; \forall v \in V. \tag{7}$$

The variable x_v indicates whether vertex v is in set S or not. The constraints are defined over the vertices such that either vertex v itself or one of its neighbors is in the set S. The LP relaxation is a pure covering LP,

$$\min \langle \mathbb{1}, \boldsymbol{x} \rangle \text{ s.t. } (\boldsymbol{I} + \boldsymbol{A})\boldsymbol{x} \geq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \; \boldsymbol{x} \in \mathbf{R}^n, \tag{8}$$

where $\boldsymbol{I}$ is the identity matrix.

Vertex Cover (*vcover*). In the vertex cover problem the goal is to find the smallest subset of vertices $S \subseteq V$ such that every edge has one of its endpoints in S (hence S covers all the edges). A simple IP formulation is

$$\min \sum_{v \in V} x_v \text{ s.t. } x_u + x_v \geq 1 \; \forall (u,v) \in E \quad x_v \in \{0,1\} \; \forall v \in V. \tag{9}$$

Here, the x_v variables determines whether a vertex v is in set S. These variables are defined over the vertices. There is one constraint per edge. The LP relaxation is a pure covering LP,

$$\min \langle 1, \boldsymbol{x} \rangle \text{ s.t. } \boldsymbol{M}^{\mathsf{T}} x \geq \mathbb{1}, \boldsymbol{x} \geq \mathbb{0}, \boldsymbol{x} \in \mathbf{R}^n, \tag{10}$$

where $\boldsymbol{M}^T$ is the transpose of the vertex-edge incidence matrix of the graph.

Densest Subgraph (*dense-sub*). Densest subgraph finds a subgraph $S \subset G$ that maximizes the edge to vertex count ratio, i.e., $|E(S)|/|S|$. The LP, as formulated in [14], is

$$\max \sum_{e \in E} x_e \text{ s.t. } x_e \leq y_u, x_e \leq y_v \;\; \forall e = (u,v) \in E \quad \sum_{v \in V} y_v \leq 1 \quad x_e, y_v \geq 0 \; \forall v \in V, \forall e \in E. \tag{11}$$

The variables x_e represent the edges and the variables y_v represent the vertices, which are no longer binary. Since this problem is not a positive LP, we consider its dual [11],

$$\min D \text{ s.t. } z_{u,e} + z_{v,e} \geq 1 \; \forall e = (u,v) \in E \quad \sum_{e \in \text{inc}(v)} z_{v,e} \leq D \; \forall v \in V \quad z_{v,e} \geq 0 \; \forall v \in V, e \in \text{inc}(v). \tag{12}$$

While (12) is still not a positive LP, we can convert it to a mixed packing and covering LP by fixing D to be a constant and treating (12) as a feasibility problem instead. We find an approximate minimum value to (12) via binary search for D and solving the feasibility LP for each choice.

Unlike previous LPs, the variables $z_{v,e}$ represent a vertex-edge pair instead of vertices or edges, so we require new constraint matrices. Let $\boldsymbol{I}$ be an identity matrix of size m, and let the function interweave take two equally-sized matrices and put the first and second matrices' first columns as the first two columns of the combined matrix, then their second columns as the next pair of columns, and so on. We call the resulting matrix $\boldsymbol{W}$ the *interweaved identity matrix*, where

$$\boldsymbol{W}_{e,2e}, \boldsymbol{W}_{e,2e+1} = 1, \; \forall e \in E, \boldsymbol{W} \in \{0,1\}^{n \times 2m} \tag{13}$$

In order to model the vertex-edge pair variables, we can form a matrix called vertex-edge pairs matrix. Vertex-edge pairs matrix will have a column for each vertex v and edge (u,v). Specifically, we can form the vertex-edge pair matrix, $\boldsymbol{O} \in \{0,1\}^{n \times 2m}$, from a graph G as follows:

$$\boldsymbol{O}_{u,2e+b} = \begin{cases} 1 & : b = 0, e = (u,v) \in E \\ 1 & : b = 1, e = (v,u) \in E \\ 0 & : \text{otherwise} \end{cases} \tag{14}$$

Then the feasibility variant of the dual to densest subgraph is

$$\exists z \in \mathbb{R}^{2m} \text{ s.t. } \boldsymbol{W}z \geq \mathbb{1}, \; \boldsymbol{O}z \leq D \cdot \mathbb{1}, \; z \geq \mathbb{0}. \tag{15}$$

4 MWU WITH LINE SEARCH

We propose two methods for finding a step size in `StepSize` (Line 11). To motivate these methods, we cast them as 1-D optimization problems, similar to line search methods for (gradient) descent methods [34]. However, instead of finding a step size that minimizes the objective function [34], we take the largest step that ensures a local invariance condition is satisfied.

Algorithm 2 Multi-Update MWU Method with step size search for Mixed Packing and Covering LPs

1: **procedure** MWU_STEPSIZE_SEARCH($\boldsymbol{P} \in \mathbf{R}_+^{m_P \times n}$, $\boldsymbol{C} \in \mathbf{R}_+^{m_C \times n}$, ϵ)
2: Initialize η, and x_i as in Algorithm 1
3: $\boldsymbol{y} \leftarrow \boldsymbol{P}\boldsymbol{x}$, $\boldsymbol{z} \leftarrow \boldsymbol{C}\boldsymbol{x}$
4: **while** constraints not approximately satisfied and $\boldsymbol{C} \neq \emptyset$ **do**
5: $\boldsymbol{g} \leftarrow \boldsymbol{P}^\intercal \nabla \text{smax}_\eta(\boldsymbol{y})$ ▷ Packing gradient
6: $\boldsymbol{h} \leftarrow \boldsymbol{C}^\intercal \nabla \text{smin}_\eta(\boldsymbol{z})$ ▷ Covering gradient
7: $d_i \leftarrow \frac{1}{2\eta} \max\{0, 1 - \frac{g_i}{h_i}\} \cdot x_i \ \forall i$ ▷ New step direction
8: **if** $\max(\boldsymbol{d}) = 0$ **then**
9: Return "INFEASIBLE"
10: $\boldsymbol{d}^{(y)} \leftarrow \boldsymbol{P}\boldsymbol{d}$, $\boldsymbol{d}^{(z)} \leftarrow \boldsymbol{C}\boldsymbol{d}$
11: $\alpha \leftarrow \text{StepSize}(\boldsymbol{d}, \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{d}^{(y)}, \boldsymbol{d}^{(z)}, \eta)$ ▷ Step size search
12: **if** $\alpha < 1$ **then**
13: Return "INFEASIBLE"
14: $\boldsymbol{x} \leftarrow \boldsymbol{x} + \alpha \cdot \boldsymbol{d}$
15: $\boldsymbol{y} \leftarrow \boldsymbol{y} + \alpha \cdot \boldsymbol{d}^{(y)}$, $\boldsymbol{z} \leftarrow \boldsymbol{z} + \alpha \cdot \boldsymbol{d}^{(z)}$
16: $\boldsymbol{C} \leftarrow \{c_i \ : \ c_i^T x < 1\}$
17: **return** x

We store the packing and covering constraints $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ and $\boldsymbol{z} = \boldsymbol{C}\boldsymbol{x}$ as well as $\boldsymbol{d}^{(y)} = \boldsymbol{P}\boldsymbol{d}$ and $\boldsymbol{d}^{(z)} = \boldsymbol{C}\boldsymbol{d}$ (Line 3 and 10) to minimize the number of sparse matrix-vector products, or SpMVs. The step direction $\boldsymbol{d}$ is unchanged (Line 7). While the algorithm drops satisfied constraints (Line 16), in practice we keep satisfied constraints since this simplifies the implementation and we did not find it impacts the convergence on the problems we tested.

The sub-routine `StepSize` (Line 11) takes the step vector $\boldsymbol{d}$ and constraint vectors, and returns a step size $\alpha > 0$. We call this modification *step size search* and design algorithms for it in the next subsection. When $\alpha < 1$, we report that finding a solution is infeasible, because otherwise a step size of $\alpha = 1$ is always possible due to the theoretical analysis of Mahoney et. al. [31]. Therefore, we call a step size $\alpha = 1$ found without step size search the *standard step size*. Assuming $\alpha \geq 1$, we scale the step direction $\boldsymbol{d}$ by α and add it to $\boldsymbol{x}$ (Line 14). Afterwards, we update $\boldsymbol{y} = \boldsymbol{P}\boldsymbol{x}$ and $\boldsymbol{z} = \boldsymbol{C}\boldsymbol{x}$ without SpMVs (Line 15). Note that α may be large enough so that $\boldsymbol{P}\boldsymbol{x} = \boldsymbol{y} + \alpha \cdot \boldsymbol{d}^{(z)} \geq 1$, in which case we terminate MWU.

4.1 Line Search as a Constrained Optimization Problem

In this section, we consider algorithms for finding a step size for `StepSize`. When selecting α, we want it to be sufficiently large to accelerate MWU convergence while ensuring that we recover a feasible solution to (2).

First, we consider the case of the mixed packing covering LP. One of the qualities of Mahoney et. al's algorithm is that the algorithm reaches a $(1+\epsilon)$-approximation when the difference in a potential function $f(x) = \frac{1}{\eta}(\text{smax}_\eta(\boldsymbol{P}\boldsymbol{x}) - \text{smin}_\eta(\boldsymbol{C}\boldsymbol{x}))$ becomes sufficiently small [31]. Each step that their algorithm takes is non-increasing on $f(x)$.

We can show that in order for the potential function to be non-increasing, $f(x^{(t+1)}) - f(x^{(t)}) = \Psi(\alpha) - \Phi(\alpha) \leq 0$ where,

$$\Phi(\alpha) = \text{smin}_\eta(\boldsymbol{C}(\boldsymbol{x} + \alpha \cdot \boldsymbol{d})) - \text{smin}_\eta(\boldsymbol{C}\boldsymbol{x})$$
$$\Psi(\alpha) = \text{smax}_\eta(\boldsymbol{P}(\boldsymbol{x} + \alpha \cdot \boldsymbol{d})) - \text{smax}_\eta(\boldsymbol{P}\boldsymbol{x}).$$

This gives us an equivalent invariant $f(\alpha) = \Phi(\alpha)/\Psi(\alpha) \geq 1$. Therefore, if we find a step size α for which this invariant holds, then for this α we can still say $f(x)$ is non-increasing, and furthermore, if we can reach a point where $\boldsymbol{P}\boldsymbol{x} \leq (1+\epsilon)\mathbb{1}$ and $\boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}$ then we have converged to a feasible solution. Hence, our method is to find the largest step size $\alpha > 0$ such that the "bang-for-buck" value is at least one, or

$$f(\alpha) = \Phi(\alpha)/\Psi(\alpha) \geq 1, \tag{16}$$

For pure packing and pure covering problems we instead have these invariants, respectively:

$$\begin{aligned} \langle \mathbb{1}, \alpha\boldsymbol{d} \rangle / \Psi(\alpha) &\geq 1 \\ \langle \mathbb{1}, \alpha\boldsymbol{d} \rangle / \Phi(\alpha) &\leq 1 \end{aligned} \tag{17}$$

We now show MWU with line search maintains the same theoretical properties as MWU with the standard step size [31]. Recall $\boldsymbol{x} \in \mathbf{R}^n$ and m is the number of rows in the matrices $\boldsymbol{P}$ and $\boldsymbol{C}$.

Theorem 4.1. *MWU with line search (Algorithm 2) either returns an $(1+\epsilon)$-relative approximate solution, i.e., an $x \geq \mathbb{0}$ such that $\boldsymbol{P}\boldsymbol{x} \leq (1+\epsilon)\mathbb{1}$ and $\boldsymbol{C}\boldsymbol{x} \geq \mathbb{1}$, or correctly reports the LP is infeasible. The number of iterations is at most $\tilde{O}(\epsilon^{-3})$, where $\tilde{O}$ hides polylogarithmic dependence on n, m, and ϵ.*

The proof is similar to the one shown in [31], which implicitly sets the step size to $\alpha = 1$. There will be two main differences in the convergence proof, which we highlight here. First, the proof of correctness in [31] shows the potential function, defined as $\text{smax}_\eta(\boldsymbol{P}\boldsymbol{x}) - \text{smin}_\eta(\boldsymbol{C}\boldsymbol{x})$, is monotonically decreasing by taking a first-order approximation of smooth max and min. On the other hand, our bang-for-buck invariance (16) explicitly ensures this monotonicity property. Second, the argument in [31] upper bounds the number of MWU iterations by lower bounding the values in step direction vector $\boldsymbol{d}$. Since line search only increases $\boldsymbol{d}$ because $\alpha \cdot \boldsymbol{d} \geq \boldsymbol{d}$, then line search can only decrease the number of MWU iterations.

While line search can decrease the number of iterations (in fact, quite significantly in our experiments), finding a step size increases the work per iteration. In the following lemma, we leverage the monotonicitiy of $f(\alpha)$ to design efficient line search algorithms.

Proposition 4.2. *f is monotonically decreasing for $\alpha \in \mathbf{R}_+$.*

Proof. We show that as α increases $\Psi(\alpha)/\alpha$ is increasing while $\Phi(\alpha)/\alpha$ is decreasing, hence

$$f(\alpha) = (\Phi(\alpha)/\alpha)/(\Psi(\alpha)/\alpha)$$

is decreasing. Note that Ψ is convex since smax_η is convex, and Φ is concave since smin_η is concave. Since Ψ is convex, $\Psi(\alpha) \leq \Psi(0) + \alpha\Psi'(\alpha) = \alpha\Psi'(\alpha)$. Hence, we can show that $\Psi(\alpha)/\alpha$ is increasing, since

$$(\Psi(\alpha)/\alpha)' = \frac{1}{\alpha}(\Psi'(\alpha) - \Psi(\alpha)/\alpha) \geq 0.$$

Analogously, since Φ is concave, the inequalities above are reversed, and so $\Phi(\alpha)/\alpha$ must be strictly decreasing in α. □

4.2 Implementing Line Search

To approximate the maximum step size α^* satisfying (16), we first perform exponential search to find an integer p where $f(2^p) \geq 1$ and $f(2^{p+1}) < 1$. By Proposition 4.2, $\alpha^* \in [2^p, 2^{p+1})$. Next, we run binary search starting with a lower and upper bound of $l = 2^p$ and $u = 2^{p+1}$, and update the lower and upper bounds so that l (resp. u) is the largest (smallest) value such that $f(l) \geq 1$ ($f(u) < 1$). Once we find an ϵ-relative step size, or when $\frac{u-l}{l} \leq \epsilon$, we return l. We formalize the aforementioned procedure in Algorithm 3.

When $f(\alpha) \geq 1$ and $\max\left(\boldsymbol{C}(\boldsymbol{x} + \alpha \cdot \boldsymbol{d})\right) \geq 1$ (Line 4), this means the step size can lead MWU to completion, so we immediately return α. Moreover, binary search makes use of $\boldsymbol{y} = \boldsymbol{Px}$, $\boldsymbol{z} = \boldsymbol{Cx}$, $\boldsymbol{d}^{(y)} = \boldsymbol{Pd}$, and $\boldsymbol{d}^{(z)} = \boldsymbol{Cd}$, where $\boldsymbol{x}$ is our current solution and $\boldsymbol{d}$ is the computed MWU update direction, to avoid additional SpMVs.

A similar line search, which is a coordinate binary search, was proposed in [36, Section 2.8], but there are some important differences compared to our binary search. First, rather than taking a step in the full gradient direction $\boldsymbol{d} \in \mathbb{R}^n$, the coordinate binary search updates sequentially in each of the n indices. Thus, the binary search can have a critical path of length up to n and is therefore not parallel. Second, the coordinate binary search replaces the difference in the smooth min and smooth max from Φ and Ψ with their respective first-order approximations, which incurs approximation errors in (16) and can lead to more conservative step sizes. By conducting line search in the full gradient $\boldsymbol{d}$ and using the exact difference for Φ and Ψ, our proposed binary search improves upon the previous line search in non-trivial ways and can take more aggressive (i.e., larger) step sizes while maintaining feasibility.

Algorithm 3 Finding a step size via binary search

```
1:  procedure BinSearch({y, d^(y)} ∈ R^{mp}_{≥0}, {z, d^(z)} ∈ R^{mc}_{≥0}, ε)
2:      α ← 1
3:      while f(α) ≥ 1 do                          ▷ Exponential search. See (16)
4:          if min(z + α · d^(z)) ≥ 1 then
5:              return α                           ▷ Return early if constraints are satisfied
6:          α ← 2 · α
7:      lb, ub ← α/2, α
8:      while ub − lb > (1 − ε)lb do               ▷ Binary search
9:          β ← avg(lb, ub)
10:         if f(β) ≥ 1 then                       ▷ See (16)
11:             lb ← β
12:         else
13:             ub ← β
14:     α ← lb and return α
```

We can derive another line search method by using Newton's method, which has the update,

$$\alpha_{k+1} = \alpha_k - g(\alpha_k)/g'(\alpha_k), \text{ where } g(\alpha) = f(\alpha) - 1.$$

Because Newton's method converges when its solution is in the neighborhood of the optimal solution, we require estimates of α^* to ensure convergence. We do so via a warm start for Newton's search, where we set our initial α_0 to the previous optimal step size, if available, or use exponential search. The reason for the former strategy is we observed in our tests that the optimal step size between two MWU iteration are relatively close. Finally, we note that once Newton's method converges to some solution, it may not strictly satisfy (16). Thus, we multiplicatively decrease the solution by a factor of $(1-\epsilon)^p$ for some integer p (p is typically small) until (16) is satisfied.

5 SOFTWARE OPTIMIZATIONS AND PARALLELIZATION

We now describe the details of our implementation and parallelization of linear algebra operations within MWU. To efficiently perform sparse matrix-vector products with matrices introduced in Section 3, such as the vertex-edge adjacency matrix, we leverage implicit representations derived from a standard sparse vertex-vertex adjacency matrix data structure. We accelerate vector operations with loop fusion and vectorization.

5.1 Shared-Memory Optimizations

We design implicit SpMVs for matrices that arise in graph-based positive LPs such as vertex-edge incidence matrices. We adapt previous fusion techniques for the needs of MWU framework [38].

5.1.1 Choice of Matrix Format. To efficiently traverse the non-zeros in the adjacency matrix during SpMVs, we use the Compressed Sparse Blocks (CSB) format [12], which can achieve good cache locality for both SpMVs of the matrix and its transpose.

CSB divides the matrix into two-dimensional $r \times k$ tiles. Each tile is represented as a list of tuples, where each tuple stores the non-zeros in column major order in coordinate (COO) format. The group of tiles that belong to r consecutive rows is called a *row-block* while the group of tiles that belong to c consecutive columns is called a *column-block*. In our implementation, we store the tiles in row-major order.

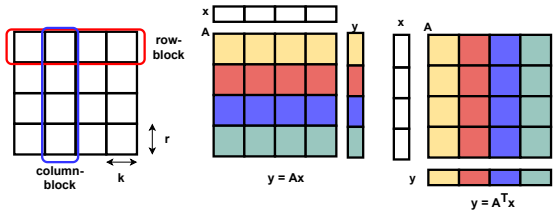


Fig. 2. CSB representation and its SpMV operation.

Similar to [1], we parallelize $\boldsymbol{y} = \boldsymbol{Ax}$ and $\boldsymbol{y} = \boldsymbol{A}^T\boldsymbol{x}$ over the row-blocks and column-blocks, respectively. Provided the tile size ($r \times k$) is selected carefully, the block of the input vector ($\boldsymbol{x}$) processed by a tile and output vector $\boldsymbol{y}$ corresponding to a row-block (updated by a single thread) is contained in private L1 or L2 caches. Figure 2 illustrates the parallelization of SpMV using CSB format.

5.1.2 Implicit Representations. In each iteration of MWU, we perform one SpMV with the constraint matrix and with its transpose. As seen in Section 3, the constraint matrices of many graph-based LPs are the vertex-edge incidence matrix of the graph. We notice that the edge (u, e) encoded in an incidence matrix can be described over over a vertex and a vertex pair $(u, (v, w))$ where the pair (v, w) represents an edge in the adjacency matrix. In addition, in the incidence matrix, the value is 1 if $u = v$ or $u = w$ and 0 otherwise.

Therefore, we can store the nonzero data in an vertex-edge incidence matrix $\boldsymbol{M}$ implicitly as an adjacency matrix in memory. This reduces the memory cost of the constraint matrix by about half and therefore also reduce the number of accesses to the memory subsystem, particularly cache.

Having an implicit representation means that linear algebra operations on these matrices can also be expressed implicitly by formulating the computations on the adjacency matrix, $\boldsymbol{A}$.

We emulate SpMVs for $\boldsymbol{M}$ and $\boldsymbol{O}$ by storing the edge, row, and column index for each non-zero in $\boldsymbol{A}$, which we denote by e, r, and c, respectively. For example, we can compute $\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}$ by accumulating element x_e to y_r and y_c for each (r, c, e) in $\boldsymbol{A}$, and likewise for $\boldsymbol{y} = \boldsymbol{O}\boldsymbol{x}$, then evaluating $y = y_r + y_c$. To parallelize this SpMV while avoiding race conditions, we first traverse in row major order while reading the row indices of $\boldsymbol{A}$, then in column major order while reading the column indices.

We compute $\boldsymbol{y} = \boldsymbol{M}^T\boldsymbol{x}$ by accumulating elements x_r and x_c to y_e for all (r, c, e) in $\boldsymbol{A}$, and likewise for $\boldsymbol{y} = \boldsymbol{O}^T\boldsymbol{x}$. Parallelizing these SpMVs are straightforward, since we accumulate to any given element of $\boldsymbol{y}$ two and one times, respectively.

5.1.3 Loop Fusion and Vectorization Opportunities. In each iteration of the MWU algorithm, we do several vector operations and our augmentation to step size search adds many vector operations, too. For some problems, such as vertex-cover or densest subgraph, the vectors in these operations have size $|E|$, which means they can be as costly as a single SpMV.

Since these vector operations loop to apply simple arithmetic to each element in the vector, combining multiple vector operations in one pass via loop fusions can accelerate these methods. We identify two operations for fusion: (1) the gradient calculations using *smax* and *smin* (Lines 5 and 6) and (2) the calculation of the new step direction (16). In both cases, loop fusion can reduce memory accesses and facilitate automatic vectorization.

5.2 Distributed Parallelization

Distributed-memory parallelization of vector operations and explicit sparse matrix vector products in MWU can be done with standard techniques. Therefore, in this section, we will focus on describing and analyzing the benefits of the implicit representation for distributed-memory communication.

We use the same implicit representation described in Section 5.1.2. We leverage a 2D matrix distribution of the adjacency matrix to perform implicit SpMVs with the incidence matrix. A 2D data layout is communication-efficient for matrix vector products since each processor computes on only $n/\sqrt{p}$ entries and contributes to $n/\sqrt{p}$ outputs (for an $n \times n$ matrix on a $\sqrt{p} \times \sqrt{p}$ processor grid). They are commonly employed for parallel processing of adjacency matrices [13].

With a 2D layout of the adjacency matrix, we perform vertex-edge incidence products with twice the communication cost of an adjacency matrix product. To do so, we store vector information corresponding to edges in the same processor layout as the adjacency matrix A. This means that for an edge (u, v) in A, the machine owning the edge would store vector information for indices corresponding to u and to v.

For simplicity, we assume a square processor grid. With this approach, the product with the vertex-edge incidence matrix, $\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}$, requires only a reduction of contributions to $\boldsymbol{y}$ along rows and columns of the processor grid. While for the product $\boldsymbol{y} = \boldsymbol{M}^T\boldsymbol{x}$, only a broadcast of entries of $\boldsymbol{x}$ along rows and columns of the processor grid is needed. In both cases, each processor sends or receives a subvector of size $O(\sqrt{n}/p)$.

6 EXPERIMENTAL SETUP

6.1 System Setup

We use Intel Knights Landing (KNL) nodes on the Stampede2 supercomputer as our testbed. Each KNL node has 68 1.4 GHz cores. Each core has a 32 KB private L1 cache, and 2 neighboring cores share a 1 MB L2 cache. KNL processors also support AVX2 and AVX-512 vector instructions. Each Stampede2 node has 112 GB of memory capacity with 96 GB DRAM and 16 GB MCDRAM used in cache mode.

6.2 Implementations

MWU Implementations.

We implement two different versions of MWU 1: (1) *MWU-PETSc*, and (2) *MWU-opt*. *MWU-PETSc* relies on an efficient parallel BLAS library PETSc [9] while *MWU-opt* is our hand-optimized implementation using optimizations discussed in Section 5. In our implementation of Algorithm 1, we do not drop satisfied constraints (i.e., we skip Line 13). This simplifies the implementation, and we did not find this affects convergence. We set $\epsilon = 0.1$ and terminated the algorithm if it exceeds 5000 iterations. To verify correctness, we compare the solution from MWU with an exact solution, if available. `max_iter=5000`. These parameters control the accuracy and maximum number of iterations of MWU, respectively, which were found by hand-tuning the algorithm. To verify correctness, we compare the solution from MWU with an exact solution, if available.

PETSc is a suite of data structures and routines for large-scale distributed operations, including vector and sparse matrix operations (which calls (sparse) BLAS under the hood), with a Python interface (`petsc4py`) [9]. On Stampede2, we use PETSc with MKL version 19.1.1. We use C++ for our *MWU-opt* optimized implementation and OpenMP for parallelism. We compile our code with Intel compiler version 19.1.1 and enable *-O3*, and *-mAVX512* compiler flags. We run the *MWU-opt* implementation by binding threads to physical cores using *numactl –physcpubind*. For *MWU-PETSc*, we find that creating N processes each with 1 thread gives the best performance.

General LP Solvers. We compare our optimized MWU implementation to general LP solvers. We use *IBM CPLEX* [15] and *Gurobi* [21], both in multi-threaded settings. If the problem is an ILP, we do not round the fractional solution.

For *CPLEX*, we set the run mode to *opportunistic* to achieve the fastest (but non-deterministic) run time. For *Gurobi*, we use the concurrent optimization setting, which concurrently runs *primal simplex*, *dual simplex*, and the *barrier method*. We report the fastest run time out of these three methods. When the barrier method finishes first, we report its run time *before crossover* (unless otherwise noted) for a more fair comparison to MWU, which outputs fractional solutions. We implement all applications discussed in Section 3 with

both *CPLEX* and Gurobi. Finally, we limit the solve time for both *CPLEX* and *Gurobi* to 4 hours. All other parameters were set to defaults.

Specialized Algorithms. We consider optimized custom implementations as baselines for the two implicitly integral LP problems: bipartite matching and densest subgraph. For the former, we use *ms-bfs-graft* [8], which employs the serial Karp-Sipser greedy initialization step [24] followed by a specialized breadth-first searches to find augmenting paths. For the latter problem, we used the Graph Based Benchmark Suite's [18] (*GBBS*) approximate densest subgraph algorithm, which implements Charikar's greedy 2-approximation algorithm [14]. Both *ms-bfs-graft* and *GBBS* are implemented in C++ with OpenMP. We compile *ms-bfs-graft* using OpenMP and the Intel compiler (version 19.1.1) with the *-O2* flag. We compiled *GBBS* with the g++ compiler version 9.1.0.

6.3 Input Graphs

We select a variety of real-world and synthetic undirected graphs from the SuiteSparse Matrix Collection [16] and list them in Table 1. Our real-world graphs come from diverse domains, such as a road, social, and user-product network. We also use two sets of synthetic graphs, the first set being random geometric graphs (*rgg*) which have a planar-like structure, and the second set being Kronecker graphs (*kron*) from Graph500 which show a strong community structure.

Note that none of the graphs we selected are bipartite, which is required in *bmatch*. To obtain bipartite graphs, we read the input adjacency matrix as a biadjacency matrix, meaning that the rows and columns of the matrix correspond to the left and right sets of vertices, respectively, where edges can only go between between vertices in different sets.

Table 1. List of real-world and synthetic graphs

Graphs (Abv.)	$\|V\|$	$\|E\|$
usroads (usroads)	129,164	330,870
com-Amazon (amazon)	334,863	1,851,744
coPapersCiteseer (papers)	434,102	32,073,440
hollywood-2009 (hollyw)	1,139,905	113,891,327
com-Orkut (orkut)	3,072,441	234,370,166
kron-X	X=2^{17}-2^{21}	≈X×80
rgg-Y	Y=2^{17}-2^{24}	≈Y×15

7 EXPERIMENTAL RESULTS

In this section, first, we compare our implementation to state-of-the-art software including general LP solvers, *CPLEX* and *Gurobi*, to specialized parallel implementations for particular graph problems, and to the parallel implementation of another multiplicative weights update algorithm from Makari et. al [32].

Then, we evaluate the effectiveness of our algorithmic improvements and software optimizations. We start by finding how the incorporation of a step size search reduces the number of MWU iterations. We then test the performance improvements from our software optimizations and its scalability by comparing performance between *MWU-PETSc* and *MWU-opt*.

7.1 Comparison of MWU to Other Algorithms

We now compare the *MWU-opt* implementation of MWU with Newton's method and all the software implementation optimizations to other state-of-the-art optimization libraries and custom implementations (*ms-bfs-graft* for *bmatch* and *GBBS* for *dense-sub*). All experiments are run with 64 threads on a single KNL node. Table 2 shows the execution times to find $(1+\epsilon)$-relative solutions for four positive LPs on various graphs where $\epsilon = 0.1$. A "-" in a cell means that the input graph was either too large to be processed by the library, or the run time exceeded 4 hours.

Comparison with Exact LP Solvers.

For all LP solvers, we do not do rounding as post-processing. Therefore, for the exact solvers, we have integral solutions to graphs problems that have implicitly integral LPs, and exact fractional solutions for the relaxed LPs with integrality gaps. For approximate solvers, we are not guaranteed an integral solution on integral LPs. Note that *CPLEX* and *Gurobi* return exact solutions for the target LPs, while MWU finds an $(1+\epsilon)$-relative solution with a target value of $\epsilon = 0.1$. We find that our *MWU-opt* implementation is able to find an $\epsilon = 0.1$ solution in all cases except *bmatch* problem with *rgg-20*. However, even for this case, our error rate is 0.104.

Our results show *MWU-opt* consistently outperforms *CPLEX* and *Gurobi* libraries. For *bmatch*, *dom-set*, *vcover*, and *dense-sub* graph LPs, *MWU-opt* is up to 2548x (*rgg-21*), 1482x (*rgg-19*), 43x (*kron-19*), and 2860x (*kron-17*) faster than *CPLEX*, respectively. Although *Gurobi* is generally faster than *CPLEX*, *MWU-opt* outperforms *Gurobi* by up to 1070x (*rgg-21*), 55x (*rgg-19*), 5x (*rgg-21*), and 816x (*rgg-20*) for *bmatch*, *dom-set*, *vcover*, and *dense-sub* graph LPs, respectively, before crossover occurs. When comparing the time when after crossover or one of the simplex methods from *Gurobi* terminates (whichever comes first), the relative speedups are 1462x (*rgg-21*), 3510x (*rgg-19*), 10x (*usroads*), and 878x (*rgg-20*) for *bmatch*, *dom-set*, *vcover*, and *dense-sub* graph LPs, respectively. In addition to significant speedups, we also observe that MWU is capable of running much larger problems. For instance, both *CPLEX* and *Gurobi* can only solve *kron-21* for *dom-set* but not for *dense-sub*, the latter which contains three times more nonzeros than the former. Moreover, both LP solvers fail to solve any of the problems with the largest graphs, *hollywood* and *orkut*.

Comparison with Custom Implementations. We compare *MWU-opt* performance to *ms-bfs-graft* for *bmatch* and to *GBBS* for *dense-sub*. The MS-BFS algorithm returns an exact solution. *MWU-opt* returns an approximate solution with $\epsilon = 0.1$. The MS-BFS algorithm [8] initializes with a serial Karp-Singer greedy step and finds augmenting paths in parallel using specialized BFS. The performance of *ms-bfs-graft* heavily depends on the graph structure. In general, we observed the *MWU-opt* can outperform *ms-bfs-graft* for graphs with planar structures by 1.8-22.4x for the *rgg* graphs and *usroads*. On the other hand, for graphs which contain a strong community-structure or vertices with high degrees, *ms-bfs-graft* outperforms MWU. For example, amongst the *kron* graphs, *ms-bfs-graft* can be up to 450x faster than *MWU-opt*. For these types of graph instances, *bmatch* generally spends much less time on the grafting process than with planar-structured graphs, which we often find to be the dominating cost of *bmatch*.

Table 2. Run time (in seconds) of *MWU-opt* compared to other optimization libraries and custom applications. Cells with a dash indicate the algorithm took 4+ hours to run or had a memory error, with the exception of *vcover* with *Gurobi* on all the *kron* graphs and *hollyw.*, which had a `ConstraintError`.

	bmatch				*dom-set*			*vcover*			*dense-sub*			
	MWU	**CPLEX**	**Gurobi**	**graft**	**MWU**	**CPLEX**	**Gurobi**	**MWU**	**CPLEX**	**Gurobi**	**MWU**	**CPLEX**	**Gurobi**	**gbbs**
rgg-15	0.08	9.15	6.93	0.07	0.38	5.56	3.60	2.38	3.99	7.88	0.16	24.01	12.90	0.04
rgg-16	0.09	21.23	14.38	0.14	0.34	15.62	10.64	3.37	10.72	6.86	0.22	54.57	25.61	0.05
rgg-17	0.24	45.35	31.68	0.32	0.68	43.11	22.91	5.65	28.98	15.90	0.54	141.95	119.43	0.09
rgg-18	0.15	114.31	76.76	0.71	3.54	561.23	49.01	9.85	82.43	38.42	0.66	349.34	344.72	0.13
rgg-19	0.54	283.79	170.81	1.86	2.08	3,045.28	111.90	23.48	114.63	86.94	1.41	1,202.33	877.80	0.21
rgg-20	0.43	716.66	406.77	4.44	7.79	-	255.15	44.84	292.40	226.01	2.34	4,081.44	2,017.68	0.32
rgg-21	0.85	2,186.03	917.63	11.12	17.25	-	555.60	87.95	659.18	504.65	5.15	-	-	0.56
rgg-22	2.81	-	-	30.67	46.37	-	1,313.81	183.67	-	-	13.21	-	-	0.90
rgg-23	3.92	-	-	80.99	247.68	-	-	367.86	-	-	22.40	-	-	1.69
rgg-24	21.00	-	-	226.12	115.02	-	-	856.06	-	-	74.87	-	-	3.18
kron-16	3.80	95.61	136.6	0.00	1.38	19.22	23.22	4.53	81.83	-	1.81	3,169.42	-	0.11
kron-17	1.87	200.79	335.18	0.01	10.97	63.81	59.44	55.16	194.90	-	2.85	8,053.81	-	0.17
kron-18	2.60	462.66	642.27	0.01	3.07	214.97	155.06	33.63	414.28	-	10.53	-	-	0.26
kron-19	4.38	-	-	0.01	10.24	657.11	379.04	53.06	2,354.63	-	12.88	-	-	0.43
kron-20	10.36	-	-	0.02	32.51	2,292.96	1,048.8	82.93	3,091.02	-	24.30	-	-	0.67
kron-21	32.33	-	-	0.04	584.85	7,072.73	2,342.73	210.92	-	-	56.28	-	-	1.13
usroads	0.07	20.24	12.30	0.04	1.49	16.30	8.85	1.07	12.31	5.22	0.27	35.25	16.80	0.03
amazon	0.71	93.93	126.34	0.05	22.14	115.10	98.99	2.13	72.38	-	1.52	750.42	1,040.63	0.09
papers	6.44	3,549.09	542.19	0.33	10.14	35.14	49.15	149.95	767.30	443.05	3.78	-	3,945.37	0.39
hollyw.	39.13	-	-	0.56	43.86	-	-	130.50	-	-	24.79	-	-	1.89
orkut	29.99	-	-	29.12	162.24	-	-	334.70	-	-	201.09	-	-	3.18

For *dense-sub*, *GBBS* implements Charikar's greedy 2-approximation algorithm for densest subgraph [14], but the relative error is usually much better in practice (but worse than $\epsilon = 0.1$). Again, we run *MWU-opt* with $\epsilon = 0.1$. We observe that *GBBS* always outperforms MWU, achieving a maximum speedup of 63.2x and minimum speedup of 4x.

Comparison with Previous Work. We compare MWU (Algorithm 2) and an implementation of a gradient descent algorithm with adaptive error [32], which uses a less theoretically efficient multiplicative weights update algorithm but is the only other distributed study of multiplicative weights update methods on graph problems. Their paper compared their algorithm, called MPCSolver, against their implementation of Young's parallel algorithm for feasibility generalized matching, which is a mixed packing and covering LP, for an $(1+\epsilon)$-relative solution ($\epsilon = 0.05$) and found that the implementation of MPCSolver outperformed Young's algorithm. We provide a detailed description of the problem, datasets, and gradient descent algorithm in Appendix A.1.

We run the same experiment as them with *MWU-opt*, using the same datasets as well: the Netflix and KDD datasets [10, 19]. Because both algorithms solve the same LP with a multiplicative weight update approach, there are only minor differences in vector operations between the two algorithms. Consequently, for this section, we compare iteration counts rather than time between the two algorithms. The two proposed algorithms are compared in Figure 3. For MWU, we consider both the standard step size and the Newton's method for step size search. The gradient descent data is manually extracted from [32] using WebPlotDigiter [37]. The plot shows both MWU with Newton's method and gradient descent with adaptive error find a $(1+\epsilon)$-relative solution in less than 2000 iterations, whereas MWU with standard step size converges much more slowly. Moreover, MWU with Newton's method incurs 10× and 41× fewer iterations than gradient descent for Netflix and KDD, respectively.

These results highlight the effectiveness of using a step size strategy, such as Newton's method, over the standard step size. Furthermore, MWU with Newton's method converges more rapidly than gradient descent with an adaptive error. However, since both methods use heuristics to accelerate the method, testing additional positive LPs and datasets would be needed for a comprehensive understanding of the trade-offs between MWU and gradient descent. The heuristic that MPCSolver uses prematurely stops the algorithm once it detects that the per-iteration decrease in constraint violation falls below a threshold.

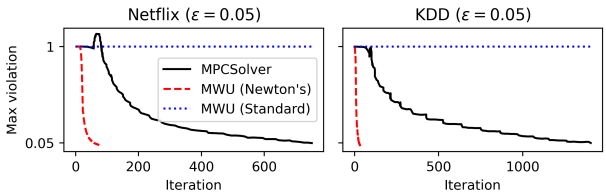


Fig. 3. Max violation, defined as $\max\{0, \max(\boldsymbol{Px}) - 1, 1 - \min(\boldsymbol{Cx})\}$ for MPCSolver, which is a gradient descent algorithm with adaptive error [32], and MWU (Algorithm 2) with standard step size and Newton's method.

7.2 Parallel Scalability of MWU

We now analyze the strong scaling behavior of *MWU-PETSc* and *MWU-opt*. Figure 4 displays the speedup with respect to single-threaded execution of the *MWU-opt* implementation.

When executing on a single node, all LP problem types are run along a range of thread counts from single threaded to 68 threads, which is the maximum number of hardware threads on one machine. Here, *MWU-opt* is able to achieve speedup over 16x with 68 threads in 90% of experiments and over 32x in 50% of experiments. Overall, the *MWU-opt* implementation achieves speedups of 13-55x on 68 threads compared to its single threaded run times. The largest differences between *MWU-opt* and *MWU-PETSc* are observed on graph applications where we use specialized matrices such as *vcover*, *bmatch*, and *dense-sub* problems. High parallel efficiency in *MWU-opt* is achieved due to load balancing and high locality in matrix-vector multiplications of transposed specialized matrices and vector operations. On the other hand, we see that the *MWU-opt* can only achieve 2-3x speedup compared to *MWU-PETSc* for dominating

set LP (*dom-set*) application. Note that, for *dom-set*, *MWU-opt* can only make use of format selection and memory access minimization optimizations for SpMV operations.

For multi-node results, we execute *MWU-opt* and *MWU-PETSc* with 64 MPI processes per node and 1 thread per process (this was the most performant configuration for *MWU-PETSc* on a single machine). We only run experiments where the total number of processes is square, as that is a requirement for our implicit representation. We run all algorithms with Newton's optimization for step size search and limit Newton's methods to 5000 iterations. On distributed memory, except for vertex cover on *rgg-24*, *MWU-opt* runs faster than *MWU-PETSc* at scale. For the distributed problem, we also observe almost linear scaling for all graphs except *rgg-24* in *MWU-opt*. A matrix-vector product on the incidence matrix of banded matrices, like *rgg-24*, reduces communication on a 1D data distribution pattern, so a 1D parallelization (e.g., row-wise distribution of the matrix) is more efficient than a 2D distribution, which is the layout we use.

For *MWU-PETSc*, we observe good scaling on *dom-set*, which we expect as the LP for this problem does not use implicit representation and is a 1D problem. We observe poor scaling in all other problems on all graphs except *rgg-24*. For *rgg-24*, *MWU-PETSc* achieves good performance due to its internal representation [9], which, we believe communicates only the vector entries needed by each processor based on the sparsity pattern of rows assigned to it. However, *MWU-PETSc* performs extremely poorly on *dense-sub* when we use multiple processors and does not complete in under 2 hours. In conclusion, for general graphs, the implicit 2D representation scales well compared to a explicit 1D representation.

7.3 Improvements from Step Size Strategy

We first evaluate the effectiveness of step size search (Section 4). We run *MWU-PETSc* using 64 MPI proccesses, each with 1 thread and list the results for *rgg-18* in Table 3. We choose this graph since we have exact solutions for all five graph problems, and the run time with standard step size is not too large. The speedups for other graphs are within an order of magnitude of the ones listed here.

Table 3. Convergence of MWU to find $(1+\epsilon)$-relative ($\epsilon = 0.1$) solution on *rgg-18* with standard step strategy (**Std**), binary search (**Bin**), and Newton's method (**Nwt**). For the latter two step strategies, we use the previous step size as the initial step size.

	# MWU iters			Avg # step size iters		Time (sec)		
	Std	**Bin**	**Nwt**	**Bin**	**Nwt**	**Std**	**Bin**	**Nwt**
match	25477	13	13	8.31	4.86	79.3	0.87	0.94
bmatch	28210	15	13	8.00	5.07	261	1.08	1.05
dom-set	18837	96	166	5.78	2.58	41.3	1.30	1.58
vcover	30531	76	110	5.93	2.68	106	1.99	2.19
dense-sub	20021	21	18	8.00	4.79	170	0.60	0.47

The results verify that a step size search strategy significantly reduces the number of MWU iterations compared to the standard step size prescribed in theoretical algorithms. Since an MWU iteration tends to be more expensive than a search step iteration (due to the SpMV), these results suggest that finding accurate step sizes, at the expense of a higher search cost, reduces the overall run time. The performance difference between binary search or Newton's method is relatively small. While Newton's method on average requires fewer step size search iterations than binary search, it has more MWU iterations than binary search for the two pure covering problems, *dom-set* and *vcover*. The additional MWU iterations observed when using Newton's method may be attributed to the $(1-\epsilon)$ multiplicative decrease (where $\epsilon = 0.1$) applied to step sizes violating the bang-for-buck inequality (16).

7.4 Effect of Software Optimizations

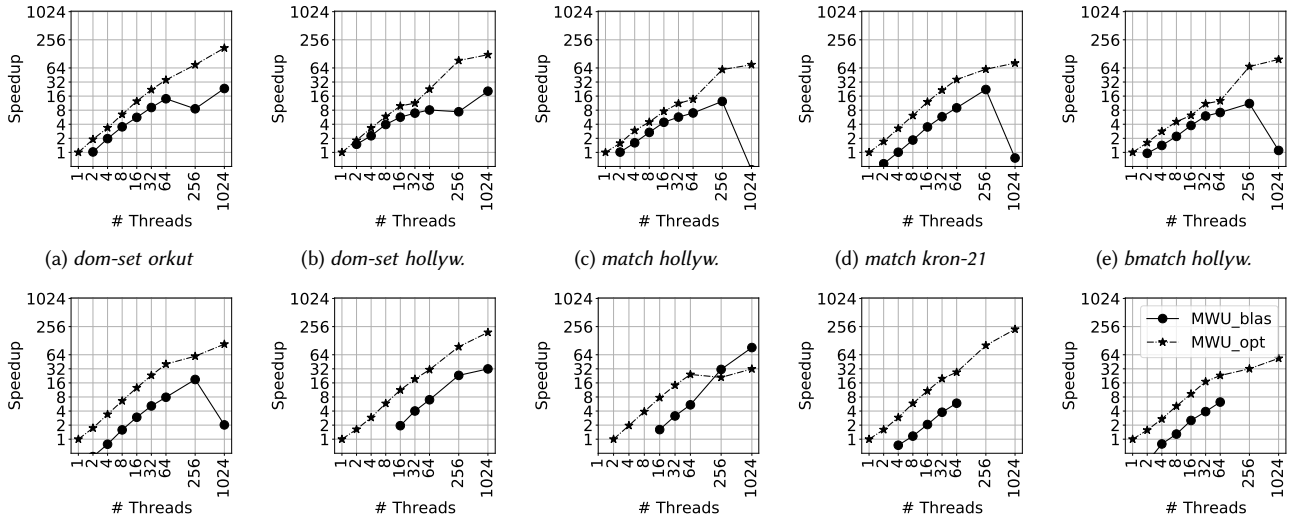
We now analyze the acceleration of an MWU iteration with our software optimizations. To do so, we will compare the execution times of *MWU-PETSc* and *MWU-opt* implementations with 68 threads. Later, we will also compare the execution times of *MWU-PETSc* and *MWU-opt* implementations for problems with implicit matrix vector multiplication.

7.4.1 Performance Breakdown. First, we consider where the cycles are spent in our *MWU-PETSc* implementation. Figure 5a shows the fraction of time spent in matrix-vector products (*matvec*), step size search (*search*), and other vector operations (*vec*). The gradients (Line 5, 6) and new direction (Line 7) are included in the *vec* category while all other vector operations done during step size search are included in the search category.

We observe that both applications and input graphs affect the distribution of execution cycles among these three components. For example, while *match*, *bmatch* and *dom-set* problems spend most of their time during *matvec*, *vcover* and *dense-sub* problems spend more than 50% of their execution time for *vec* and *search* operations. *matvec* takes on average 75%, 78%, and 82% of the execution time for *match*, *bmatch*, and *dom-set* problems, respectively. In contrast, for *vcover* and *dense-sub*, *matvec* takes only 45% and 38% of the execution time on average. Due to this variable behavior, it is crucial to optimize both matrix-vector multiplications and vector operations for MWU.

7.4.2 Shared-Memory Performance Optimizations. Figure 5c shows the speedup obtained by our optimized implementation relative to the PETSc-based implementation when executing on a single node. In this section, we report geometric mean speedups when referring to average speedup across graphs. For *dom-set*, the speedup is obtained from using a favorable format (CSB) and minimizing memory accesses. Our optimizations accelerate *matvec* operations by 1.8x on average. Although *vec* and *search* operations also get speedups, their contribution to overall performance is smaller. On the other hand, for *match*, *bmatch*, and *vcover* problems, we can observe the benefit of specialized vertex-incidence matrix vector multiplications. In these cases, *matvec* operations are 3.28x, 5.06x, and 3.49x faster on average, respectively. For *dense-sub* problem, we also see benefits of vertex-edge pair matrix and interweaved-identity matrix specializations. *matvec* operations are 4.64x faster on average.

Moreover, *vcover* and *dense-sub* problems spend a large amount of time for *vec* and *search* operations. We see that, in these cases, *MWU-opt* implementation can obtain significant speedups for both *vec*



(a) *dom-set orkut* (b) *dom-set hollyw.* (c) *match hollyw.* (d) *match kron-21* (e) *bmatch hollyw.*

(f) *bmatch kron-21* (g) *vcover orkut* (h) *vcover rgg-24* (i) *dense-sub orkut* (j) *dense-sub* rgg-24

Fig. 4. Scalability of *MWU-opt* and *MWU-PETSc* (see subfigure (j) for the legend). Axes are in log_2 scale. All values are normalized to single-threaded execution of *MWU-opt*. We omit the results for *MWU-PETSc*, if the execution time is slower than single-threaded execution of *MWU-opt*.

(6.85x, and 4.08x on average, respectively) and *search* (8.92x and 5.79x on average, respectively) thanks to fusing and SIMD optimizations.

7.4.3 Distributed-Memory Optimizations. We record run time improvements in the context of multi-node exeuction *MWU-opt* over *MWU-PETSc* in Table 4. In parenthesis is the ratio of *matvec* product time to *matvec* communication time for *MWU-opt*. All experiments are run with 64 OMP threads per MPI process. We observe that for 4 nodes, the use of implicit matrix-vector products accelerates *matvec* operations in *MWU-opt* by 1.4-3x compared to *MWU-PETSc* for all graphs except *rgg-24*, for which it is slower by 9x. As previously discussed in Section 7.2, *MWU-PETSc* uses a 1D communication layout, which is more efficient on banded matrices like *rgg-24* than our 2D communication layout.

Table 4. Speed-up in run time of our implicit implementation of the product of the edge-incidence matrix and a vector. The ratio of computation to communication time in *MWU-opt* is parenthesized.

	hollyw	*orkut*	*rgg-24*	*kron-21*
4 nodes	1.4 (0.57)	3 (1.1)	0.11 (0.09)	3 (0.83)
16 nodes	120 (0.39)	46 (0.19)	0.04 (0.02)	97 (0.18)
64 nodes	186 (0.3)	259 (0.12)	0.09 (0.01)	516 (0.19)

8 CONCLUSION

Our work demonstrates that approximate positive LP solvers are an efficient and scalable approach for solving a wide range of graph problems. We show that with carefully chosen modifications and implementation of the MWU algorithm from Mahoney et. al. [31] – namely, a step size search strategy and specialized linear algebra operations that leverage shared and distributed-memory resources – the algorithm exceeds the performance of general purpose LP solvers for finding a $(1+\epsilon)$-relative solution. Our implementation also matches the performance of hand-tuned parallel graph libraries for some graphs.

ACKNOWLEDGMENTS

This research has been supported by funding from the United States National Science Foundation (NSF) via grant #1942995 and #2016136, as well as NSF grant CCF-1910149. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0022158.

REFERENCES

[1] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 1213–1222. https://doi.org/10.1109/IPDPS.2014.125

[2] Zeyuan Allen-Zhu and Lorenzo Orecchia. 2014. Using optimization to break the epsilon barrier: A faster and simpler width-independent algorithm for solving positive linear programs in parallel. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 1439–1456.

[3] Zeyuan Allen-Zhu and Lorenzo Orecchia. 2016. Using Optimization to Solve Positive LPs Faster in Parallel. arXiv:1407.1925 [cs.DS]

[4] Zeyuan Allen-Zhu and Lorenzo Orecchia. 2019. Nearly linear-time packing and covering LP solvers. *Mathematical Programming* 175, 1-2 (2019), 307–353.

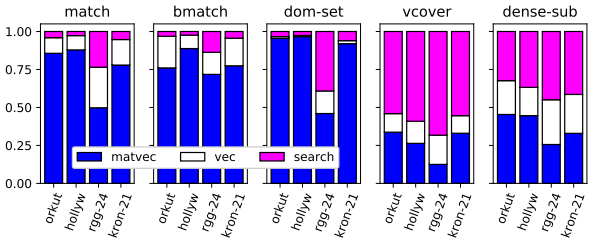
[5] Alexandr Andoni, Clifford Stein, and Peilin Zhong. 2020. Parallel approximate undirected shortest paths via low hop emulators. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 322–335.

[6] Sanjeev Arora, Elad Hazan, and Satyen Kale. 2012. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing* 8, 1 (2012), 121–164.

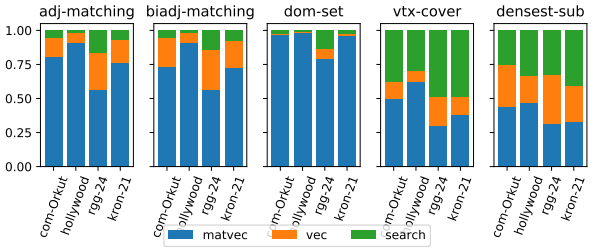
[7] Baruch Awerbuch and Rohit Khandekar. 2009. Stateless distributed gradient descent for positive linear programs. *SIAM J. Comput.* 38, 6 (2009), 2468–2486.

[8] Ariful Azad, Aydın Buluç, and Alex Pothen. 2016. Computing maximum cardinality matchings in parallel on bipartite graphs via tree-grafting. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 44–59.

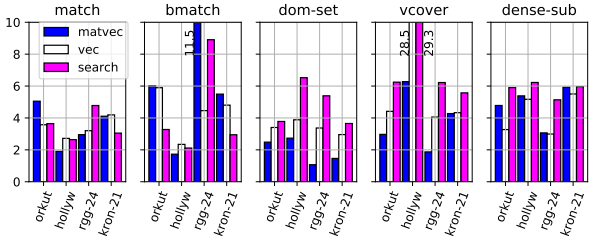
[9] Satish Balay, Shrirang Abhyankar, Mark Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, W Gropp, et al. 2019. PETSc users manual. (2019).



(a) *MWU-PETSc* execution time breakdown



(b) OMP Breakdown



(c) Speedups of MWU components with *MWU-opt* compared to *MWU-PETSc*

Fig. 5. Breakdown of execution times and speedups obtained with *MWU-opt* for different components.

[10] James Bennett, Stan Lanning, et al. 2007. The Netflix prize. In *Proceedings of KDD cup and workshop*, Vol. 2007. New York, NY, USA., 35.

[11] Digvijay Boob, Yu Gao, Richard Peng, Saurabh Sawlani, Charalampos Tsourakakis, Di Wang, and Junxing Wang. 2020. Flowless: extracting densest subgraphs without flow computations. In *Proceedings of The Web Conference 2020*. 573–583.

[12] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[13] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509.

[14] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*. Springer, 84–95.

[15] IBM ILOG Cplex. 2009. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation* 46, 53 (2009), 157.

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[17] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 918–934.

[18] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. 2020. The graph based benchmark suite (gbbs). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–8.

[19] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! music dataset and KDD-cup'11. In *Proceedings of KDD Cup 2011*. PMLR, 3–18.

[20] Michael D. Grigoriadis and Leonid G. Khachiyan. 1994. Fast Approximation Schemes for Convex Programs with Many Blocks and Coupling Constraints. *SIAM J. Optim.* 4, 1 (1994), 86–107. https://doi.org/10.1137/0804004

[21] LLC Gurobi Optimization. 2021. Gurobi Optimizer Reference Manual. http://www.gurobi.com

[22] Slobodan Jelić, Sören Laue, Domagoj Matijević, and Patrick Wijerama. 2015. A fast parallel implementation of a PTAS for fractional packing and covering linear programs. *International Journal of Parallel Programming* 43, 5 (2015), 840–875.

[23] Richard M Karp and Vijaya Ramachandran. 1989. A survey of parallel algorithms for shared-memory machines. (1989).

[24] Richard M Karp and Michael Sipser. 1981. Maximum matching in sparse random graphs. In *22nd Annual symposium on foundations of computer science (sfcs 1981)*. IEEE, 364–375.

[25] Christos Koufogiannakis and Neal E Young. 2014. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica* 70, 4 (2014), 648–674.

[26] Jason Li. 2020. Faster parallel algorithm for approximate shortest path. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 308–321.

[27] Jonathan S Li, Rohan Potru, and Farhad Shahrokhi. 2020. A performance study of some approximation algorithms for minimum dominating set in a graph. *arXiv preprint arXiv:2009.04636* (2020).

[28] Yang P Liu and Aaron Sidford. 2020. Faster energy maximization for faster maximum flow. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 803–814.

[29] Michael Luby and Noam Nisan. 1993. A parallel approximation algorithm for positive linear programming. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 448–457.

[30] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[31] Michael W Mahoney, Satish Rao, Di Wang, and Peng Zhang. 2016. Approximating the Solution to Mixed Packing and Covering LPs in Parallel $\tilde{O}(\epsilon^{-3})$ Time. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Available at urlhttps://drops.dagstuhl.de/opus/volltexte/2016/6333/pdf/LIPIcs-ICALP-2016-52.pdf.

[32] Faraz Makari, Baruch Awerbuch, Rainer Gemulla, Rohit Khandekar, Julián Mestre, and Mauro Sozio. 2013. A distributed algorithm for large-scale generalized matching. (2013).

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP'13)*. Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[34] Jorge Nocedal and Stephen Wright. 2006. *Numerical optimization*. Springer Science & Business Media.

[35] Serge A Plotkin, David B Shmoys, and Éva Tardos. 1995. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research* 20, 2 (1995), 257–301.

[36] Kent Quanrud. 2019. Fast Approximations for Combinatorial Optimization via Multiplicative Weight Updates. https://www.ideals.illinois.edu/bitstream/handle/2142/106153/QUANRUD-DISSERTATION-2019.pdf.

[37] Ankit Rohatgi. 2020. Webplotdigitizer: Version 4.4. https://automeris.io/WebPlotDigitizer

[38] Gerald Roth and Ken Kennedy. 1998. Loop Fusion in High Performance Fortran. In *Proceedings of the 12th International Conference on Supercomputing* (Melbourne, Australia) *(ICS '98)*. Association for Computing Machinery, New York, NY, USA, 125–132. https://doi.org/10.1145/277830.277857

[39] Jonah Sherman. 2013. Nearly maximum flows in nearly linear time. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 263–269.

[40] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[41] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[42] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[43] Xin Sui, Donald Nguyen, Martin Burtscher, and Keshav Pingali. 2010. Parallel graph partitioning on multicore architectures. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 246–260.

[44] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2020. Bipartite matching

in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 919–930.

[45] Jan van den Brand, Yin Tat Lee, Aaron Sidford, and Zhao Song. 2020. Solving tall dense linear programs in nearly linear time. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 775–788.

[46] Vijay V Vazirani. 2013. *Approximation algorithms*. Springer Science & Business Media.

[47] Di Wang. 2017. *Fast Approximation Algorithms for Positive Linear Programs*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-126.html

[48] David P Williamson and David B Shmoys. 2011. *The design of approximation algorithms*. Cambridge university press.

[49] Neal E Young. 2001. Sequential and parallel algorithms for mixed packing and covering. In *Proceedings 42nd IEEE symposium on foundations of computer science*. IEEE, 538–546.

[50] Neal E Young. 2014. Nearly linear-work algorithms for mixed packing/covering and facility-location linear programs. https://arxiv.org/abs/1407.3015. *arXiv preprint arXiv:1407.3015* (2014).

A SUPPLEMENTARY MATERIAL

A.1 Further Details on Generalized Matching Experiments

Let $G = (V, E)$ be an undirected, unweighted graph. For generalized matching, a vertex v can be matched $b(v)$ times, where $\text{lb}(v) \leq b(v) \leq \text{ub}(v)$ are lower and upper bounds on the number of unique vertices matching with v. More precisely, the IP formulation is

$$\begin{aligned} \exists\, \boldsymbol{x} \text{ s.t. } \text{lb}(v) \leq \sum_{e \in \text{inc}(v)} x_e \leq \text{ub}(v), \forall v \in V, \\ x_e \in \{0, 1\}, \forall e \in E. \end{aligned} \tag{18}$$

Maximum matching is equivalent to generalized matching with $\text{lb}(v) = 0$ and $\text{ub}(v) = 1, \forall v \in V$, as well as a (maximization) objective function of $\sum_e x_e$. The LP relaxation is the feasibility mixed packing and covering LP,

$$\exists\, \boldsymbol{x} \in \mathbf{R}^m \text{ s.t. } \boldsymbol{M}\boldsymbol{x} \geq \boldsymbol{l}, \boldsymbol{M}\boldsymbol{x} \leq \boldsymbol{u}, \boldsymbol{x} \geq \mathbb{0},$$

where $\boldsymbol{M}$ is the vertex-edge incidence matrix, and $\boldsymbol{l}, \boldsymbol{u} \in \mathbf{R}^n$ are the vector of lower and upper bounds for each vertex.

A.2 Dataset Preprocessing

Now, let us describe how to pre-process the Netflix [10] and KDD [19] datasets, as detailed in [32]. Both datasets contain users and items (e.g., movies in Netflix, music tracks in KDD) as vertices, and edges correspond to a user rating an item. This dataset is represented as a bipartite graph, where users and items form the two partitions, and edges go only between vertices in separate partitions. For the number of matchings with each user, we enforce a lower bound of three and upper bound of five. For items, no lower bound is set, but an upper bound of 200 and 2000 is chosen for Netflix and KDD, respectively. Finally, to ensure there is a feasible matching satisfying these bounds, we exclude users with less than ten ratings from the Netflix dataset. After this pre-processing step, the two datasets have 473k and 1.6m vertices, as well as 100m and 252m edges, respectively.

A.3 Gradient Descent with Adaptive Error

Finally, we review the gradient descent algorithm with an adaptive error of [32]. In short, the algorithm minimizes the convex function via gradient descent,

$$\Gamma(\boldsymbol{x}) = \sum_{i=1}^{m_P} y_i(\boldsymbol{x}) + \sum_{i=1}^{m_C} z_i(\boldsymbol{x}),$$

where for some $\mu > 0$,

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

The algorithm contains two error values. There is the error bound ϵ, where the algorithm seeks to find an $\boldsymbol{x}$ that is a $(1+\epsilon)$-relative solution. Then there is the *internal* error bound ϵ', which is used to specify μ as well as which coordinates of x_i to update, and by how much. The authors of [32] found that they can set $\epsilon' > \epsilon$. For example, when $\epsilon = 0.05$, they can choose $\epsilon' = 1$. Then they run the algorithm until it stagnates, and if $\boldsymbol{x}$ is not an $(1+\epsilon)$-relative solution, they decrement ϵ' and warm-start the algorithm by setting the initial $\boldsymbol{x}_0$ to the solution of the previous, stagnated algorithm. This strategy is called *adaptive error*, since it adaptively updates the internal error bound.