X-Wim: Massive Parallelization of Weighted Matching in Bipartite Graphs

Dayi Fan
The Ohio State University
fan.1090@osu.edu

Simon Zhang
The Ohio State University
zhang.680@osu.edu

Rubao Lee
The Ohio State University
lee.11875@osu.edu

Hanqi Guo
The Ohio State University
guo.2154@osu.edu

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

ABSTRACT

The maximum weight perfect matching (MWPM) problem in bipartite graphs has extensive applications in database, machine learning, financial markets, quantum computing, and other data-intensive domains, and serves as a general formulation of weighted matching problems. The Hungarian algorithm is widely adopted for solving bipartite MWPM, and substantial research efforts have focused on improving its sequential time complexity. As data volumes grow and real-time processing demands escalate, parallel solutions become increasingly essential. However, efficient parallelization remains highly nontrivial due to the algorithm's intricate execution patterns, inherently sequential data dependencies, frequent phase switching, and the single-path-per-iteration search constraint.

These critical issues motivate us to develop X-Wim, a massively parallel framework. It is built on a new phase-decoupled approach that breaks the strong interleaving between algorithmic phases, eliminates frequent global updates, enables concurrent search for multiple disjoint paths, and incorporates an adaptive search strategy. These algorithmic design efforts lead to substantial performance gains, even in the single-threaded setting. Extensive experiments on real-world datasets indicate that X-Wim surpasses state-of-the-art baselines, achieving up to a 9.93x speedup with 1 core and up to a 56.3x speedup with 8 cores. It also exhibits strong scalability. In tests up to 96 cores, it achieves an average 1.70x speedup each time the number of threads doubles. To the best of our knowledge, X-Wim is the fastest solution for this class of graph algorithms.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Davis-Fan/X-Wim.git.

1 INTRODUCTION

Bipartite weighted matching is one of the most classical problems in graph theory and has found wide-ranging applications across diverse domains. Specifically, in machine learning, it is widely used in multi-object tracking [25, 106] and recommender systems [56, 77] to obtain optimal allocations. In economics, it serves as a pivotal tool for optimizing two-sided market [2, 79], job assignment [95], and auction mechanisms [10, 100]. In database systems, it is applied to schema linkage [13, 18] and rank aggregation [53]. Moreover, on social platforms, bipartite weighted matching is employed in partner recommendation [58, 113] and advertisement delivery [86, 128]. Recent advances in quantum computing [3, 57, 118] further adopt weighted matching techniques for quantum error correction. Beyond these, bipartite weighted matching also plays a crucial role in robotics and health care, where it has been applied in multi-robot coordination [27, 83, 98] and organ transplant programs [12, 127].

A *matching* in a graph is a set of edges with no common vertices. A *perfect matching* is a matching that covers all vertices of the graph. The *maximum weight perfect matching* (MWPM) is a perfect matching that maximizes the total weight. Figure 1 presents an example. MWPM provides a general formulation for weighted matching problems because other variants, such as maximum weight matching and minimum weight perfect matching, can be easily reduced to MWPM by adding dummy vertices or negating weights.

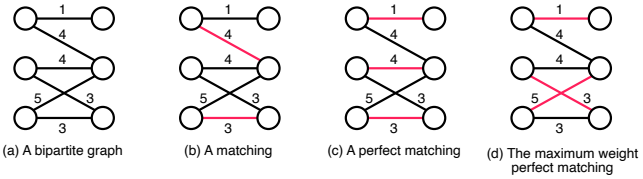


Figure 1: (a) shows a bipartite graph; (b) is a valid matching (edges in the matching are shown in red); (c) shows a perfect matching; (d) is the maximum weight perfect matching

Despite its broad applicability, solving MWPM in bipartite graphs remains computationally expensive due to its intricate execution pattern and high time complexity. The *Hungarian algorithm* [71, 88] is widely recognized as the first polynomial-time solution to the bipartite MWPM problem and has since become the most widely used approach in practice. The original algorithm has a time complexity of $O(|V|^2|E|)$, where $|V|$ and $|E|$ denote the vertex count and the edge count, respectively. Over several decades, extensive studies [7, 26, 50, 52] have concentrated on improving the sequential time complexity of the Hungarian algorithm by utilizing more efficient data structures [41, 47, 65, 112] or exploiting special cases [39, 99, 111]. As data scales, modern applications increasingly involve graphs with millions or even billions of vertices and edges [16, 28, 31, 97], making sequential execution infeasible for latency-sensitive workloads. Hence, developing a fast and scalable parallel framework for solving bipartite MWPM has become imperative.

However, parallelizing the computation of MWPM in bipartite graphs is far from straightforward. The basic idea of the Hungarian algorithm is to iteratively search for an augmenting path to increase the current matching size while maintaining the optimality conditions, until a perfect matching is obtained. In each iteration, it builds

a search tree and repeatedly switches between the primal search phase and the dual label update phase (updating vertex labels) until an augmenting path is found. Parallelizing this procedure is highly nontrivial due to three fundamental challenges: (1) The search tree expansion is inherently sequential. When expansion stalls, the algorithm applies the smallest label update needed to enable the next expansion step, so the search can only progress incrementally. (2) The algorithm repeatedly switches between primal search and dual label updates, yielding a strongly interleaved control flow. Each update must be applied consistently before the search can continue, so frequent dual phases form hard synchronization barriers and incur significant overhead. (3) Each iteration finds only one augmenting path, and successive iterations depend on the updated matching and labels from the previous iteration, creating strong dependencies and limiting scalability on large graphs. To the best of our knowledge, there is still no parallel solver for MWPM on general bipartite graphs. Existing parallel efforts [32, 60, 66, 82] primarily target the linear assignment problem and parallelize execution only within each phase, without systematically addressing these critical issues, thereby becoming impractical at scale on general instances.

Motivated by these challenges, we develop a massively parallel framework for solving MWPM in general bipartite graphs. We first propose a new sequential phase-decoupled Hungarian algorithm that breaks the data dependencies between primal search and dual label updates and eliminates their strongly interleaved execution. The search tree expansion no longer depends on the label update, avoiding strictly incremental search progress. Moreover, each iteration performs a single primal search to find an augmenting path, followed by one dual phase that applies a consolidated label update, removing repeated phase switching. This sequential phase-decoupled algorithm reduces the overhead of frequent label updates and, more importantly, exposes key opportunities for parallel computing. Building on this, we develop an efficient parallel framework that concurrently finds multiple disjoint augmenting paths within a single iteration, thereby exploiting large-scale parallelism and reducing runtime by aggregating label updates across multiple paths. In addition, we introduce an adaptive search strategy that dynamically selects the search direction to mitigate path conflicts, further improving primal search efficiency and delivering substantial speedups. Our contributions are summarized as follows.

- We examine the intricate execution pattern of the Hungarian algorithm, identify fundamental obstacles to parallelization, and derive key insights from its primal–dual structure.
- We propose a sequential phase-decoupled Hungarian algorithm that eliminates interleaved primal–dual execution and removes repeated phase switching, exposing key opportunities for parallelization. We provide a formal proof of correctness.
- We develop a parallel phase-decoupled Hungarian algorithm that concurrently finds multiple disjoint augmenting paths to exploit large-scale parallelism and incorporates an adaptive search strategy for dynamically selecting the search direction.
- Taken together, we develop X-Wim, a massively parallel computation framework for solving MWPM in general bipartite graphs, implemented on multicore processors. Extensive experiments show that X-Wim consistently outperforms existing solutions and exhibits excellent scalability. Compared with state-of-the-art baselines, even the sequential version of X-Wim achieves up to a 9.93x speedup on real-world datasets. With 8 cores, X-Wim attains speedups of up to 53.6x. Scalability tests up to 96 cores show that X-Wim delivers an average speedup of 1.70x for each doubling of the thread count.

The paper is organized as follows. The MWPM problem and the Hungarian algorithm are presented in §2. We identify challenges and key insights in §3. The sequential phase-decoupled algorithm is proposed in §4, and the parallel phase-decoupled framework is presented in §5. Experimental results are shown in §6. We review related work in §7 and summarize our contributions in §8.

2 BACKGROUND

2.1 Matching and Augmenting Path

A bipartite graph is a graph whose vertex set V can be partitioned into two disjoint sets, L and R, such that every edge connects a vertex in L to one in R. A weighted bipartite graph is denoted by $G = (L \cup R, E, w)$, where $w : E \to \mathbb{R}$ assigns a weight to each edge.

A matching M is a set of edges such that no two edges share a common vertex. Its size $|M|$ is the number of edges in M. An edge is matched if it belongs to M; otherwise, it is unmatched. A vertex is matched if it is incident to an edge in M, and unmatched otherwise.

An *augmenting path* is a path in the graph that alternates between unmatched and matched edges, with both endpoints being unmatched vertices. Augmenting paths are essential for increasing the matching size. By flipping all unmatched edges to matched and all matched edges to unmatched along an augmenting path, the two previously unmatched endpoints become matched, and a new matching is obtained whose size is increased by one.

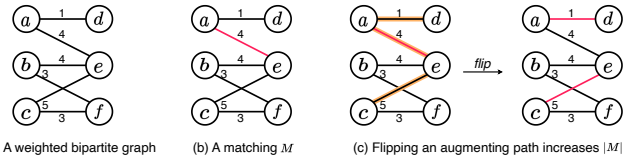


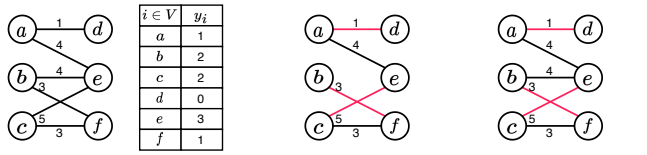
Figure 2: A matching and an augmenting path. Red edges are matched edges, and black edges are unmatched edges.

Figure 2(a) shows a weighted bipartite graph, and Figure 2(b) is a valid matching $M = \{(a, e)\}$, containing a single edge. In Figure 2(c), the path (c, e, a, d) is an augmenting path, as the edges (c, e), (e, a), and (a, d) alternate between being unmatched and matched, and both endpoints c and d are unmatched. By flipping the unmatched edges (c, e) and (a, d) to matched, and flipping the matched edge (e, a) to unmatched, the matching becomes $M = \{(c, e), (a, d)\}$, increasing its size to 2, and both c and d become matched vertices.

2.2 MWPM in Bipartite Graphs

A perfect matching is a matching that covers all vertices. Given a weighted bipartite graph, the maximum weight perfect matching (MWPM) problem is to find a perfect matching that maximizes the total weight $\sum_{e \in M} w(e)$. The Hungarian algorithm [71, 88] is widely recognized as the first polynomial-time method for solving the MWPM problem in bipartite graphs and remains the most

widely used approach in practice. Its theoretical foundation is a primal–dual framework that assigns a dual variable y_i (also referred to as a label) to each vertex $i \in V$. A set of labels $\{y_i\}_{i \in V}$ is said to be *feasible* if $y_u + y_v \geq w(u, v)$ for every edge $(u, v) \in E$. An edge (u, v) is *tight* if $y_u + y_v - w(u, v) = 0$. Under a feasible set of labels, the *equality graph* is the subgraph of the original graph that contains only tight edges. A fundamental fact is that if the equality graph contains a perfect matching, then that matching is an MWPM of the original graph [71, 88].



(a) G and a feasible set of dual labels (b) The equality graph G' and a perfect matching M' (c) M' is the maximum weight perfect matching in G

Figure 3: A perfect matching in the equality graph is the MWPM in the original graph

Figure 3(a) shows a bipartite graph G and a feasible set of dual labels $\{y_i\}_{i \in V}$. The edge (a, d) is tight because $y_a + y_d - w(a, d) = 1 + 0 - 1 = 0$. Similarly, the edges (a, e), (b, f), (c, e), and (c, f) are tight. The equality graph G', which contains these five tight edges, is shown in Figure 3(b). A perfect matching $M' = \{(a, d), (b, f), (c, e)\}$ can be found in G'. This matching M' is thus the MWPM in G, with a total weight of 9, as shown in Figure 3(c).

2.3 Execution Pattern in Hungarian Algorithm

The task of solving MWPM reduces to finding a perfect matching in the equality graph. The main idea of the Hungarian algorithm is to iteratively search for an augmenting path in the current equality graph and augment the matching. Given a weighted bipartite graph, the algorithm first initializes a feasible set of dual labels for all vertices. For each vertex $i \in L$, its dual label is set to the maximum edge weight incident to i; for each vertex $i \in R$, its dual label is set to 0. This initialization satisfies label feasibility. The algorithm then proceeds in iterations. In each iteration, it searches the current equality graph for an augmenting path from an unmatched L-side vertex to an unmatched R-side vertex; flipping edges along this path increases the matching size by one and makes both endpoints matched. The path search is implemented by growing a search tree in the equality graph. If no augmenting path can be found, the algorithm enters the dual phase and updates the labels to create additional tight edges, thereby expanding the current equality graph and enabling further tree growth. Within an iteration, the algorithm repeatedly switches between the primal phase (path search) and the dual phase (label update) until an augmenting path is found. It terminates when all vertices are matched, i.e., a perfect matching is obtained in the equality graph. The details are in Algorithm 1.

2.3.1 Primal Phase. Each iteration begins with the primal phase. The algorithm selects an unmatched L-side vertex as the root and builds a search tree in the equality graph using tight edges. The tree grows by adding one unmatched tight edge and one matched tight edge at each step (line 18); thus, all tree edges are tight. If the tree reaches an unmatched R-side vertex via a tight edge, an augmenting path is found (line 14). Flipping the matched and unmatched edges along this path increases the matching size by one. If no augmenting path is found and the tree can no longer be extended using tight edges, the algorithm switches to the dual phase.

2.3.2 Dual Phase. In the dual phase, the algorithm updates the dual labels of vertices in the current search tree to create additional tight edges. For any edge (u, v), the slack value is defined as $\delta(u, v) = y_u + y_v - w(u, v)$, which measures how far the edge is from being tight. A *frontier edge* (u, v) is an edge such that u is an L-side vertex in the tree and v is an R-side vertex outside the tree. Let Δ be the minimum slack value among all frontier edges. The algorithm then decreases the labels of all L-side tree vertices by Δ and increases the labels of all R-side tree vertices by Δ (lines 26-27). This update maintains label feasibility and makes at least one new edge tight, allowing the tree to expand. The algorithm then returns to the primal phase to continue searching for an augmenting path.

Algorithm 1 The Hungarian Algorithm

Input: A weighted bipartite graph $G = (L \cup R, E, w)$ that admits a perfect matching
Output: A maximum weight perfect matching M

1: $M \leftarrow \emptyset$, $y \leftarrow$ Initialize-Dual-Labels ▷ y_i is the label of vertex i
2: **for** each unmatched $i \in L$ **do**
3: $P \leftarrow$ Find-Augmenting-Path(G, M, i, y)
4: augment M with P ▷ flip edges along P
5: **end for**
6: **procedure** Find-Augmenting-Path(G, M, i, y)
7: construct a single-node tree rooted at i
8: $S \leftarrow \{i\}$; $T \leftarrow \emptyset$ ▷ S: L-side tree vertices; T: R-side tree vertices
9: **while** True **do** ▷ repeated switching
10: $\mathcal{E} \leftarrow$ Get-Eligible-Tight-Edges(G, S, T, y)
11: **while** $\mathcal{E} \neq \emptyset$ **do**
12: select and remove an edge (u, v) from $\mathcal{E}$
13: **if** v is unmatched **then**
14: **return** path$(i, \ldots, u) + (u, v)$
15: **else**
16: $x \leftarrow$ the vertex matched with v in M
17: **if** $x \notin S$ **then**
18: add edges (u, v) and (v, x) to the tree
19: add v to T and add x to S
20: $\mathcal{E} \leftarrow$ Get-Eligible-Tight-Edges(G, S, T, y)
21: **end if**
22: **end if**
23: **end while** — *primal*
24: ▷ *slack*: $\delta(u, v) = y_u + y_v - w(u, v)$
25: $\Delta \leftarrow \min_{u \in S,\, v \in R \setminus T} \delta(u, v)$
26: **for** each $s \in S$ **do** $y_s \leftarrow y_s - \Delta$
27: **for** each $t \in T$ **do** $y_t \leftarrow y_t + \Delta$ — *dual*
28: **end while**
29: **end procedure**
30: **procedure** Get-Eligible-Tight-Edges(G, S, T, y)
31: ▷ u is an L-side tree vertex; v is an R-side vertex outside the tree
32: **return** $\{(u, v) \in E \mid u \in S, v \in R \setminus T, y_u + y_v - w(u, v) = 0\}$
33: **end procedure**

Figure 4 illustrates an example. Initially, M is empty, and both L and R contain three vertices. The L-side vertices are shown in yellow and the R-side vertices in blue. Since the maximum edge weight incident to a is 4, its dual label is initialized to $y_a = 4$. Similarly, $y_b = 4$ and $y_c = 5$, while all R-side labels are initialized to zero. The algorithm then proceeds iteratively to search for augmenting paths.

In the first iteration, it enters the primal phase and selects the unmatched L-side vertex a as the root of the search tree. The edge (a, e) is tight and connects the root to the unmatched R-side vertex e, immediately forming an augmenting path (a, e) of length 1 (a single

unmatched edge between two unmatched vertices). Flipping along this path turns (a, e) into a matched edge, resulting in $M = \{(a, e)\}$.

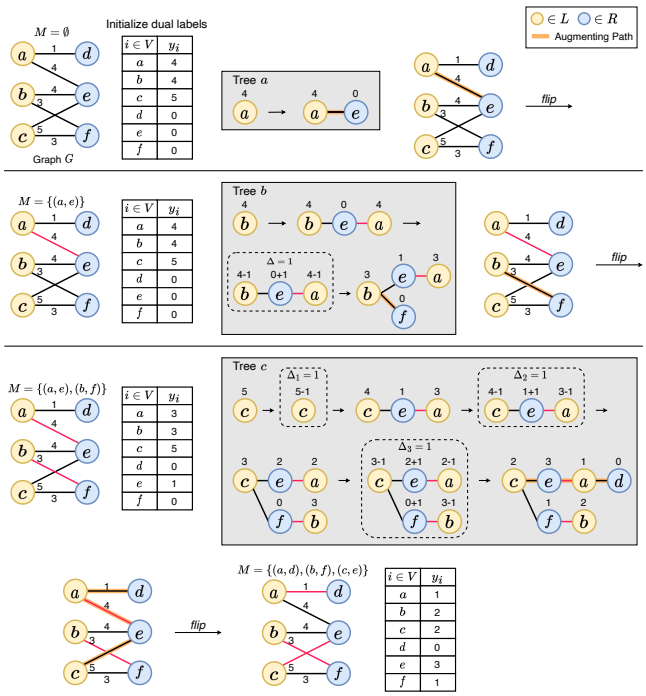


Figure 4: The execution pattern of the Hungarian algorithm on a graph G. Each row shows one search iteration. Dual label values are shown above vertices in the current search tree.

In the second iteration, it selects the next unmatched L-side vertex b as the root. The tree then expands by adding the unmatched tight edge (b, e) and the matched tight edge (e, a), bringing a into the tree. At this point, the tree cannot be extended further using tight edges, so the algorithm switches to the dual phase. The minimum slack over frontier edges from the L-side tree vertices $\{b, a\}$ to R-side vertices outside the tree is $\Delta = \min\{\delta(b, f), \delta(a, d)\} = 1$. Accordingly, it decreases the labels of the L-side tree vertices y_b and y_a by 1 and increases the label of the R-side tree vertex y_e by 1. Returning to the primal phase, the edge (b, f) becomes tight; since f is a unmatched R-side vertex, it forms an augmenting path (b, f). Flipping along this path yields $M = \{(a, e), (b, f)\}$.

In the third iteration, it selects the final unmatched L-side vertex c as the root. No tight edge is incident to c, so it enters the dual phase and performs a label update with $\Delta_1 = 1$. Returning to the primal phase, edge (c, e) becomes tight and is added to the tree, followed by the matched edge (e, a). When the tree becomes stuck again, the algorithm switches back to the dual phase and performs a second label update with $\Delta_2 = 1$. Back in the primal phase, the new tight edge (c, f) and the matched edge (f, b) are added. Again, no tight edge can extend the tree, so the algorithm switches to the dual phase a third time and performs a third update with $\Delta_3 = 1$, which makes (a, d) tight. This connects the tree to the unmatched R-side vertex d and yields the augmenting path (c, e, a, d). Flipping along this path obtains the perfect matching $M = \{(a, d), (b, f), (c, e)\}$ with total weight 9, which is the MWPM of G.

3 CHALLENGES AND KEY INSIGHTS

3.1 Parallelization Challenges

As data volumes grow and graphs reach ever larger scales, the shift from sequential execution to parallel processing is increasingly necessary for computing MWPM in bipartite graphs. However, the intricate execution pattern of the Hungarian algorithm impedes parallelization and poses fundamental challenges in developing an efficient parallel computation framework.

Challenge #1: The search tree expansion process is inherently sequential. In the primal phase, the tree is built to find an augmenting path in the equality graph, so its expansion is restricted to tight edges. When no eligible tight edges are available, the algorithm selects the frontier edge with the minimum slack value and performs a dual label update to make it tight, thereby enabling the next expansion. As a result, tree growth is incremental, and typically only one frontier edge becomes tight. Each expansion depends on the current minimum slack and the corresponding label update, which enforces inherently sequential progress and limits parallelism.

Challenge #2: Repeated switching between the primal phase and the dual phase induces a strongly interleaved control flow, where execution in one phase depends on the outcome of the other. Moreover, each dual phase acts as a hard synchronization barrier, as the primal search can proceed only after the dual labels are updated consistently. Frequent dual phases lead to significant computational overhead, since each tree expansion step may require a label update over all vertices in the tree.

Challenge #3: Only one augmenting path can be produced in each search iteration, so the matching size increases by at most one per iteration. Consequently, obtaining an MWPM in a graph with millions of vertices requires millions of search iterations, and each iteration depends on the updated matching and dual labels from the previous iteration, creating strong data dependencies across iterations. This single-path-per-iteration approach is inefficient at scale and makes the algorithm time-consuming on large graphs.

These structural issues severely constrain scalable parallelization of the Hungarian algorithm, and to the best of our knowledge, there is still no parallel solver for MWPM on general bipartite graphs. Existing parallel efforts [32, 60, 66, 82] instead focus on the linear assignment problem (LAP), where the input is typically modeled as a complete bipartite graph. These methods parallelize execution only within each phase by expanding tight edges concurrently in the primal phase and updating labels concurrently in the dual phase, while leaving the interleaved primal–dual control flow unchanged and retaining repeated phase switching, failing to expose sufficient parallelism on general instances. The most recent parallel LAP solver is HyLAC [66]. As shown in Figure 5, for a balanced random bipartite graph with $|L| = |R| = 20{,}000$ and 1,000,000 edges, HyLAC treats unspecified edges as zero-weight edges and is 9.2x slower than the basic sequential Hungarian implementation. Moreover, a time breakdown of the basic Hungarian method shows that, on the same 40,000-vertex, 1,000,000-edge graph, frequent dual label updates account for 56.1% of the total runtime, indicating that the dual phase constitutes a major performance bottleneck.

Developing a fast and scalable parallel computation framework for solving MWPM in general bipartite graphs requires systematically resolving all these critical issues. Although highly challenging,

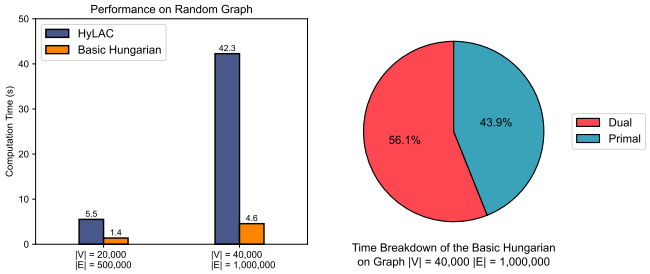


Figure 5: HyLAC [66] (RTX 3090 GPU) vs. Basic Hungarian (the basic sequential implementation on CPU) on two random graphs, with a runtime breakdown of Basic Hungarian.

first-principles thinking can help us uncover insights and reveal opportunities embedded in these difficulties.

3.2 Tightness and Feasibility in Label Update

Role of Dual Phase: *The essence of the dual phase is to create new tight edges while preserving the tightness of existing tree edges, and maintain label feasibility.*

In the dual phase, Δ is set to the minimum slack value among all frontier edges that connect an L-side tree vertex to an R-side vertex outside the tree. To make a frontier edge tight, the label of its L-side endpoint, which lies in the current tree, must decrease by Δ. However, updating only that vertex could cause other existing tree edges to become non-tight. Therefore, the algorithm applies a coordinated update in which it decreases the labels of all L-side tree vertices by Δ and increases the labels of all R-side tree vertices by Δ, ensuring that all previously tight tree edges remain tight after the update and the new tight edge can be added to the tree. Moreover, the feasibility condition requires $y_u + y_v \geq w(u, v)$ for all $(u, v) \in E$. In the dual phase, labels of L-side tree vertices decrease, reducing $y_u + y_v$ on their incident edges. If the update step were larger than the minimum slack, some L-side labels would decrease too much, causing $y_u + y_v$ to fall below $w(u, v)$ on one or more frontier edges and violating feasibility. Understanding the principles behind augmenting path search and dual label update is essential for finding the most effective way to parallelize the Hungarian algorithm.

4 PHASE-DECOUPLED METHOD

In this section, we propose a sequential phase-decoupled Hungarian algorithm that breaks the fine-grained data dependencies induced by the interleaved execution of the primal and dual phases and eliminates repeated phase switching, thus addressing **Challenge #1** and **Challenge #2** and providing an essential basis for the parallel framework presented in the next section.

4.1 Sequential Phase-Decoupled Algorithm

Both Challenge #1 and Challenge #2 arise from the fine-grained data dependencies in the interleaved execution of the primal and dual phases. The primal search and dual label update are strongly coupled. In the primal phase, the tree can expand only along tight edges, and which frontier edges become tight depends on the label update applied in the preceding dual phase. Conversely, the next dual update value is determined by the current tree and its frontier, since it is set by the minimum slack value over frontier edges. This mechanism creates strong mutual dependencies between the two phases, forcing the algorithm to repeatedly switch between tree expansion and label update in each iteration, which impedes parallel execution.

To enable phase decoupling, it is crucial to examine the dual phase in terms of its cumulative effect on vertex labels. In each dual update, the labels of all L-side tree vertices decrease by Δ, while the labels of all R-side tree vertices increase by Δ. Since the root is an L-side vertex and remains in the tree throughout the search, its label decreases in every dual phase. Consequently, within one search iteration, the root's total decrease equals the sum of the Δ values over all dual phases in that iteration. More generally, for any L-side tree vertex, its label decreases by the cumulative sum of Δ values from the time it is added to the tree until the end of iteration.

These findings lead us to develop a sequential phase-decoupled Hungarian algorithm (Algorithm 2) that separates the primal search from the dual label update and removes the interleaved primal–dual execution flow. In each iteration, the phase-decoupled algorithm performs exactly one primal phase to search an augmenting path, followed by a single dual label update that applies the consolidated changes, **without** repeatedly switching between the two phases.

In the primal phase, it still builds a search tree from an unmatched L-side vertex. However, tree expansion is **no longer** restricted to tight edges but may proceed along all edges, including non-tight edges with positive slack. Each tree vertex maintains an accumulated slack value that represents the sum of edge slack values along the path from the root to that vertex. At each expansion step, the tree grows by adding one unmatched edge followed by one matched edge. If an expansion reveals a path with smaller accumulated slack to a vertex already in the tree, the vertex is reparented to the new predecessor and its accumulated slack is updated accordingly (lines 20-22). If an unmatched edge connects an L-side tree vertex to an unmatched R-side vertex, an augmenting path is obtained, and the total accumulated slack along this path is recorded (lines 15-17). If multiple augmenting paths are discovered, the algorithm keeps only the one with the minimum total accumulated slack. The tree expansion continues until every branch in the tree has accumulated slack greater than this minimum value, ensuring that all vertices reachable within the current slack threshold have been explored.

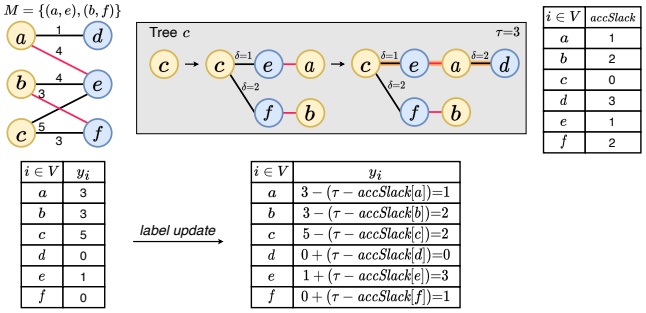
When the primal phase finishes, the algorithm enters the dual phase to update vertex labels. For each L-side tree vertex, its dual label is decreased by the gap between the augmenting path slack and its current accumulated slack; similarly, for each R-side tree vertex, its dual label is increased by the same gap (line 32). This consolidated update restores the tightness for tree edges and maintains label feasibility. The algorithm then returns the augmenting path, applies it to increase the matching size, and continues to the next iteration. It terminates when no unmatched L-side vertex remains.

Figure 6 shows the same example as the third row of Figure 4 and illustrates the execution of the phase-decoupled algorithm. The algorithm selects c as the root to build a search tree. It then expands two new branches together, one via the unmatched edge (c, e) and the matched edge (e, a), and the other via the unmatched edge (c, f) and the matched edge (f, b). The two unmatched edges have slacks $\delta(c, e) = y_c + y_e - w(c, e) = 1$ and $\delta(c, f) = y_c + y_f - w(c, f) = 2$, as annotated in the figure. Matched edges are tight and have zero

Algorithm 2 The Sequential Phase-Decoupled Hungarian Algorithm

Input: A weighted bipartite graph $G = (L \cup R, E, w)$ that admits a perfect matching
Output: A maximum weight perfect matching M

$M \leftarrow \emptyset$, $y \leftarrow$ Initialize-Dual-Labels
for each unmatched $i \in L$ **do**
  $P \leftarrow$ Phase-Decoupled-Find-Augmenting-Path(G, M, i, y)
  augment M with P
end for
procedure Phase-Decoupled-Find-Augmenting-Path(G, M, i, y)
  construct a single-node tree rooted at i
  $q_1 \leftarrow \{i\}$; $q_2 \leftarrow \emptyset$ ▷ frontier set
  $accSlack[\cdot] \leftarrow \infty$; $accSlack[i] \leftarrow 0$ ▷ min accumulated slack from root
  $\tau \leftarrow \infty$ ▷ total slacks on augmenting path
  while $q_1 \neq \emptyset$ **do** ▷ primal
    for each unmatched edge (u, v) with $u \in q_1$ **do**
      $newSlack \leftarrow accSlack[u] + y_u + y_v - w(u, v)$ ▷ tentative slack
      if $newSlack \leq \tau$ **then**
        if v is unmatched **then** ▷ augmenting path
          set the parent of v to u
          $\tau \leftarrow newSlack$; $accSlack[v] \leftarrow newSlack$; $P \leftarrow$ path$(i, \ldots, v)$
        else
          $x \leftarrow$ the vertex matched with v in M
          if $newSlack < accSlack[v]$ **then**
            set the parent of v to u and the parent of x to v
            $accSlack[v]$, $accSlack[x] \leftarrow newSlack$; add x to q_2
          end if
        end if
      end if
    end for
    $q_1 \leftarrow q_2$; $q_2 \leftarrow \emptyset$
  end while
  for each vertex z in the tree **do** ▷ dual
    $\Delta \leftarrow \tau - accSlack[z]$
    if $\Delta < 0$ **then** delete z from the tree **continue**
    if $z \in L$ **then** $y_z \leftarrow y_z - \Delta$ **else** $y_z \leftarrow y_z + \Delta$
  end for
  return P
end procedure



$i \in V$	$accSlack$
a	1
b	2
c	0
d	3
e	1
f	2

$i \in V$	y_i
a	3
b	3
c	5
d	0
e	1
f	0

$i \in V$	y_i
a	$3-(\tau - accSlack[a])=1$
b	$3-(\tau - accSlack[b])=2$
c	$5-(\tau - accSlack[c])=2$
d	$0+(\tau - accSlack[d])=0$
e	$1+(\tau - accSlack[e])=3$
f	$0+(\tau - accSlack[f])=1$

Figure 6: The execution of the phase-decoupled algorithm that eliminates the primal–dual interleaving

slack, so the slacks of (e, a) and (f, b) are omitted. The algorithm maintains an accumulated slack $accSlack[v]$ for each tree vertex v, defined as the sum of slacks along the tree path from the root to v. Hence, $accSlack[e] = accSlack[a] = 1$ and $accSlack[f] = accSlack[b] = 2$. In the next expansion, the unmatched edge (a, d) connects the L-side tree vertex a to an unmatched R-side vertex d with $\delta(a, d) = 2$, giving $accSlack[d] = accSlack[a] + \delta(a, d) = 3$. An augmenting path (c, e, a, d) is thus found, with total accumulated slack along this path $\tau = 3$. The algorithm then performs a single dual update for each tree vertex based on the gap $\tau - accSlack[v]$. For example, a is an L-side tree vertex, so y_a is decreased by $\tau - accSlack[a] = 2$, while e is an R-side tree vertex, so y_e is increased by $\tau - accSlack[e] = 2$. This produces the same final labels as the corresponding update in Figure 4.

4.2 Correctness and Complexity

Lemma 1. *The dual label update in the phase-decoupled Hungarian algorithm makes every edge in the search tree tight.*

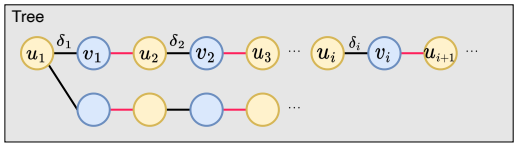


Figure 7: A search tree T rooted at an unmatched vertex u_1

Proof. Let T be a search tree rooted at an unmatched L-side vertex u_1. Consider an arbitrary unmatched tree edge $(u_i, v_i) \in T$ with $u_i \in L$ and $v_i \in R$. Let $(v_i, u_{i+1}) \in T$ be the matched edge incident to v_i, where $u_{i+1} \in L$ is the subsequent tree vertex. Let δ_i denote the slack of edge (u_i, v_i), i.e.,

$$\delta_i = \delta(u_i, v_i) = y_{u_i} + y_{v_i} - w(u_i, v_i)$$

and the matched edge (v_i, u_{i+1}) is tight under the current dual labels

$$\delta(v_i, u_{i+1}) = y_{v_i} + y_{u_{i+1}} - w(v_i, u_{i+1}) = 0$$

Let τ denote the total slack value along the augmenting path found in the primal phase. When the algorithm enters the dual phase, it decreases the label of each L-side tree vertex by the gap between τ and its accumulated slack, and increases the label of each R-side tree vertex by the same gap. After the update, the labels will be

$$y'_{u_i} = y_{u_i} - \Big(\tau - \sum_{k=1}^{i-1} \delta_k\Big),\ y'_{v_i} = y_{v_i} + \Big(\tau - \sum_{k=1}^{i} \delta_k\Big),\ y'_{u_{i+1}} = y_{u_{i+1}} - \Big(\tau - \sum_{k=1}^{i} \delta_k\Big)$$

The slack of the unmatched edge (u_i, v_i) becomes

$$y'_{u_i} + y'_{v_i} - w(u_i, v_i) = (y_{u_i} + y_{v_i} - w(u_i, v_i)) + \sum_{k=1}^{i-1} \delta_k - \sum_{k=1}^{i} \delta_k = 0$$

The slack of the matched edge (v_i, u_{i+1}) becomes

$$y'_{v_i} + y'_{u_{i+1}} - w(v_i, u_{i+1}) = (y_{v_i} + y_{u_{i+1}} - w(v_i, u_{i+1})) = 0$$

Thus, every edge in T is tight and Lemma 1 is proved. □

Lemma 2. *The dual label update in the phase-decoupled Hungarian algorithm maintains label feasibility.*

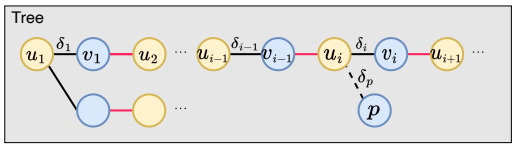


Figure 8: A search tree T and p is an vertex outside T

Proof. Label feasibility requires $y_u + y_v \geq w(u, v)$ for every edge $(u, v) \in E$, i.e., $\delta(u, v) \geq 0$. In the dual update, the labels of R-side tree vertices are increased; therefore, the feasibility condition on edges incident to these vertices cannot be violated. Moreover, by Lemma 1, all edges in the tree are tight after the update. Therefore, it suffices to consider edges outside the tree that are incident to L-side tree vertices, since only their labels are decreased.

Let T be a search tree, and let (u_i, v_i) be an arbitrary unmatched tree edge in T, where $u_i \in L$ and $v_i \in R$. Let τ denote the total slack along the augmenting path found in the primal phase. Let p be an arbitrary R-side vertex outside the tree such that $(u_i, p) \notin T$, and let δ_p denote its slack. In the primal phase, the tree contains all vertices reachable from the root with an accumulated slack no greater than τ. Since $p \notin T$, its accumulated slack must exceed τ: $\sum_{k=1}^{i-1} \delta_k + \delta_p > \tau$. After the dual update, the label of u_i becomes

$$y'_{u_i} = y_{u_i} - \Big(\tau - \sum_{k=1}^{i-1} \delta_k\Big)$$

and y_p is unchanged, so the slack of edge (u_i, p) after the update is

$$y'_{u_i} + y_p - w(u_i, p) = y_{u_i} + y_p - w(u_i, p) - \Big(\tau - \sum_{k=1}^{i-1} \delta_k\Big) = \delta_p - \tau + \sum_{k=1}^{i-1} \delta_k > 0$$

Thus, all edges remain feasible and Lemma 2 is proved. □

Theorem 1. *For any bipartite graph that admits a perfect matching, the phase-decoupled Hungarian algorithm returns an MWPM.*

Proof. The phase-decoupled algorithm repeatedly finds augmenting paths and updates the matching until no unmatched vertices remain. Hence, it returns a perfect matching M^*. By Lemma 1, the dual update makes all tree edges tight, including the edges on the augmenting path; hence all matched edges created by flipping along the path are tight. By Lemma 2, the dual update maintains label feasibility, i.e., $y_u + y_v \geq w(u, v)$ for all $(u, v) \in E$ throughout the algorithm. Because the labels are feasible, for any existing perfect matching M, we have

$$\sum_{(u,v)\in M} w(u, v) \leq \sum_{(u,v)\in M} (y_u + y_v) = \sum_{u\in L} y_u + \sum_{v\in R} y_v.$$

Moreover, every edge in M^* is tight, so the total weight of M^* is

$$\sum_{(u,v)\in M^*} w(u, v) = \sum_{(u,v)\in M^*} (y_u + y_v) = \sum_{u\in L} y_u + \sum_{v\in R} y_v.$$

Hence M^* attains the maximum possible weight among all perfect matchings, i.e., M^* is an MWPM. □

The worst-case running time complexity of the sequential phase-decoupled Hungarian algorithm is $O(|V|^2|E|)$. Each iteration finds one augmenting path and increases the matching size by one, so the algorithm performs $|V|/2$ iterations in total. Every iteration consists of one primal phase and one dual phase. In the primal phase, the algorithm grows a search tree and maintains the accumulated slack for each tree vertex. A vertex may be inserted into the frontier multiple times in the worst case, so the primal phase may examine at most $O(|V||E|)$ edges, with $O(1)$ cost per edge. The dual phase scans all tree vertices and updates their labels in $O(|V|)$ time. Therefore, the total running time is $|V|/2 \cdot \big(O(|V||E|) + O(|V|)\big) = O(|V|^2|E|)$.

The sequential phase-decoupled algorithm serves as a crucial foundation for the subsequent parallel framework. By decoupling the primal search from the dual label update, it removes the interleaved primal–dual execution that drives the algorithm's inherently sequential control flow. In the primal phase, the tree is allowed to expand beyond tight edges. This relaxation removes the incremental expansion pattern and thus resolves Challenge #1. Also, the algorithm avoids repeated phase switching and performs only one dual label update per path, thereby addressing Challenge #2. In practice, this phase decoupling delivers consistent performance gains. More importantly, it breaks the intertwined primal–dual dependencies and exposes the key opportunities for parallel execution.

5 PARALLEL FRAMEWORK

We then extend the above sequential algorithm to obtain a parallel phase-decoupled Hungarian algorithm that can concurrently find multiple disjoint augmenting paths in a single iteration to tackle **Challenge #3**. Moreover, it incorporates an adaptive strategy that dynamically changes the search direction to mitigate path conflicts, yielding significant additional speedups.

5.1 Parallel Phase-Decoupled Algorithm

Challenge #3 stems from the fact that each iteration can return only one augmenting path, making the search inefficient and incurring substantial overhead, especially on large graphs. In addition, updating the matching and dual labels after each search introduces strong dependencies across iterations, which complicates efforts to exploit parallelism. With the primal search and the dual label update already decoupled, we gain the essential flexibility to search for multiple augmenting paths in parallel and, at the end of each search iteration, apply a single label update to maintain label feasibility.

We propose a parallel phase-decoupled Hungarian algorithm (Algorithm 3) that enables finding multiple disjoint augmenting paths within a search iteration. The key idea is to build multiple trees rooted at different unmatched L-side vertices in parallel, and then search for an augmenting path in each tree concurrently. To ensure that the returned augmenting paths are disjoint, we introduce an efficient synchronization mechanism to resolve conflicts among concurrent searches. After a batch of disjoint augmenting paths is obtained, the algorithm performs a parallel dual label update over all trees to adjust vertex labels and maintain label feasibility.

Compared with the sequential phase-decoupled algorithm, each search iteration now returns a batch of disjoint augmenting paths (line 33), and the algorithm executes a single dual phase per batch rather than per path (line 36). This optimization is independent of parallel execution. Even in a single-thread setting, multi-path batching can substantially reduce dual label update overhead.

5.2 Disjoint Paths and Concurrent Label Update

When multiple augmenting paths are found in one iteration, only disjoint paths can be applied together because augmenting along overlapping paths would result in an invalid matching. Theorem 2 implies an additional advantage of the parallel phase-decoupled algorithm: any two augmenting paths found in the current iteration are either disjoint or share the same endpoint. This means that disjointness can be verified by comparing only the endpoints of the paths, instead of scanning all vertices on each path.

Theorem 2. *In the parallel phase-decoupled Hungarian algorithm, if two augmenting paths found in the current iteration are not disjoint, then they must share the same unmatched endpoint.*

Proof. Let $P_a = (a, \ldots, p, \ldots, m)$ denote the augmenting path found by the search tree rooted at a, where m is the unmatched endpoint and P_a has the minimum total slack τ_a. We claim that m

Algorithm 3 The Parallel Phase-Decoupled Hungarian Algorithm

Input: A weighted bipartite graph $G = (L \cup R, E, w)$ that admits a perfect matching
Output: A maximum weight perfect matching M

1: $M \leftarrow \emptyset$, $y \leftarrow$ Initialize-Dual-Labels
2: **while** M is not perfect **do**
3: $P_{\text{total}} \leftarrow$ Parallel-Phase-Decoupled-Find-Augmenting-Path(G, M, y)
4: augment M with all paths in P_{total}
5: **end while**
6: **procedure** Parallel-Phase-Decoupled-Find-Augmenting-Path(G, M, y)
7: $\mathcal{I} \leftarrow \{ i \in L \mid i$ is unmatched$\}$
8: **parallel for** each $i \in \mathcal{I}$ **do** ▷ multiple trees concurrently
9: construct a single-node tree T_i rooted at i
10: $q_{1,i} \leftarrow \{i\}$; $q_{2,i} \leftarrow \emptyset$ ▷ frontiers for T_i
11: $accSlack_i[\cdot] \leftarrow \infty$; $\tau_i \leftarrow \infty$; $accSlack_i[i] \leftarrow 0$; $end_i \leftarrow \infty$
12: **while** $q_{1,i} \neq \emptyset$ **do**
13: **for** each unmatched edge (u, v) with $u \in q_{1,i}$ **do**
14: $newSlack \leftarrow accSlack_i[u] + y_u + y_v - w(u, v)$
15: **if** $newSlack \leq \tau_i$ **then**
16: **if** v is unmatched **then**
17: set the parent of v to u in T_i; $accSlack_i[v] \leftarrow newSlack$
18: **if** $newSlack < \tau_i \ \vee \ (newSlack = \tau_i \wedge v < end_i)$ **then**
19: $\tau_i \leftarrow newSlack$; $end_i \leftarrow v$ ▷ break ties
20: **end if**
21: **else**
22: $x \leftarrow$ the vertex matched with v in M
23: **if** $newSlack < accSlack_i[v]$ **then**
24: set the parent of v to u and the parent of x to v in T_i
25: $accSlack_i[v], accSlack_i[x] \leftarrow newSlack$; add x to $q_{2,i}$
26: **end if**
27: **end if**
28: **end if**
29: **end for**
30: $q_{1,i} \leftarrow q_{2,i}$; $q_{2,i} \leftarrow \emptyset$
31: **end while**
32: **if** $end_i \neq \infty$ **then**
33: $P_i \leftarrow$ Parallel-Disjoint-AugPath(T_i, end_i); add P_i to P_{total}
34: **end if**
35: **end parallel for**
36: Parallel-Dual-Label-Update($\{T_i\}_{i\in\mathcal{I}}, \{\tau_i\}_{i\in\mathcal{I}}, \{accSlack_i[\cdot]\}_{i\in\mathcal{I}}, y$)
37: **return** P_{total}
38: **end procedure**

is the unmatched endpoint reachable from p that minimizes the accumulated slack among all unmatched endpoints. Otherwise, there would exist another unmatched endpoint r and a suffix $(p, \ldots, r)$ with smaller accumulated slack than $(p, \ldots, m)$ in P_a. Splicing this suffix onto the prefix $(a, \ldots, p)$ would form an augmenting path with its total slack smaller than τ_a, a contradiction. Thus, m is the unmatched endpoint with the minimum accumulated slack reachable from p. Now let $P_b = (b, \ldots, p, \ldots, n)$ be another augmenting path that intersects P_a at p. Applying the same splicing argument to P_b shows that n is also the unmatched endpoint with the minimum accumulated slack reachable from p. By the algorithm's deterministic tie-breaking rule, $m = n$, completing the proof. □

We thus design an efficient synchronization mechanism for all search trees (Algorithm 4). Each unmatched vertex maintains an atomic flag that records whether it has already been claimed as an augmenting path endpoint. During the parallel search, when a thread discovers an augmenting path ending at an unmatched vertex, it attempts to claim the endpoint via an atomic Compare-And-Swap (CAS) operation (line 3). A successful CAS guarantees exclusive ownership of that endpoint and prevents other trees from returning a conflicting augmenting path. The mechanism is lock-free, and no thread blocks. Upon a failed CAS attempt, the thread discards the candidate path and continues searching in other trees.

Algorithm 4 Find Multiple Disjoint Augmenting Paths in Parallel

1: select_endpoint[$\cdot$]$\leftarrow 0$ ▷ initialize atomic array to 0
2: **procedure** Parallel-Disjoint-AugPath(T_i, end_i)
3: **if** $atomicCAS$(select_endpoint[end_i], 0, 1) = 0 **then**
4: return path$(i, \ldots, end_i)$ on T_i
5: **end if**
6: return $\emptyset$
7: **end procedure**

Another difficulty is data races during dual label updates. All search trees are built in parallel, so different trees may overlap and share common branches. For a vertex that appears in multiple trees, each tree may propose a different update to its dual label. By Theorem 3, maintaining label feasibility requires resolving these conflicts via a maximum rule. For any shared R-side vertex, its label must increase by the maximum value proposed by the overlapping trees. Similarly, for any shared L-side vertex, its label must decrease by the maximum value, as shown in Algorithm 5.

Theorem 3. *When multiple search trees overlap, maintaining label feasibility requires applying the maximum update amount proposed by these trees at each shared vertex.*

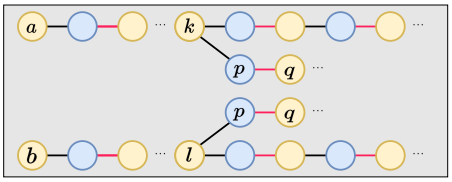


Figure 9: Two search trees share a common branch $(p, q, \cdots)$

Proof. Let T_a and T_b be two search trees that share a common branch $(p, q, \ldots)$, where $p \in R$ and $q \in L$. Let k be the parent of p in T_a and l be the parent of p in T_b. Let $y'_{p,a}$ and $y'_{p,b}$ denote the label values for p after applying only the update induced by T_a and T_b, respectively. Then (k, p) is tight under T_a, i.e., $y'_k + y'_{p,a} = w(k, p)$, and (l, p) is tight under T_b, i.e., $y'_l + y'_{p,b} = w(l, p)$. Assume $y'_{p,a} < y'_{p,b}$. If we set the final label to $y'_p = y'_{p,a}$ while keeping y'_l as produced by T_b, then the slack of (l, p) becomes

$$y'_l + y'_p - w(l, p) = y'_l + y'_{p,a} - w(l, p) < y'_l + y'_{p,b} - w(l, p) = 0,$$

which violates the feasibility condition. Therefore, at any shared R-side vertex, the final label must take the maximum value proposed by the overlapping trees, e.g., $y'_p = \max\{y'_{p,a}, y'_{p,b}\}$. Moreover, to keep the overlap edges tight in the tree proposing this maximum amount, the shared L-side vertices on the overlap branch must be decreased by the same maximum update magnitude. □

Algorithm 5 Update Dual Labels in Parallel

1: **procedure** Parallel-Dual-Label-Update($\{T_i\}_{i\in\mathcal{I}}, \{\tau_i\}_{i\in\mathcal{I}}, \{accSlack_i[\cdot]\}_{i\in\mathcal{I}}, y$)
2: $shift[\cdot] \leftarrow 0$ ▷ label update for each vertex
3: **parallel for** each vertex z in each tree T_i **do**
4: $\Delta \leftarrow \tau_i - accSlack_i[z]$
5: **if** $\Delta > 0$ **then** $shift[z] \leftarrow Max(shift[z], \Delta)$ ▷ aggregate Δ over trees
6: **end parallel for**
7: **parallel for** each vertex v in the forest $\bigcup_{i\in\mathcal{I}} T_i$ **do**
8: **if** $v \in L$ **then** $y_v \leftarrow y_v - shift[v]$ **else** $y_v \leftarrow y_v + shift[v]$
9: **end parallel for**
10: **end procedure**

The parallel phase-decoupled algorithm does not introduce any additional steps with higher asymptotic cost so its work complexity remains $O(|V|^2|E|)$. Let p be the number of processors and let c be the cost of a single atomic CAS. Both the primal and dual phases are parallelized, and each vertex performs at most one atomic CAS, so the overall parallel running time is $O(\frac{|V|^2|E|}{p} + c|V|)$.

5.3 Adaptive Search

The algorithm performance is dominated by how many disjoint augmenting paths it can find per iteration. In the current implementation, each iteration grows trees from unmatched L-side vertices and searches for augmenting paths that end at unmatched R-side vertices ($L \to R$). A key insight is that the bipartition (L, R) is symmetric, so we can reverse the search direction and grow trees from unmatched R-side vertices toward unmatched L-side endpoints ($R \to L$), without changing the search or label update rules.

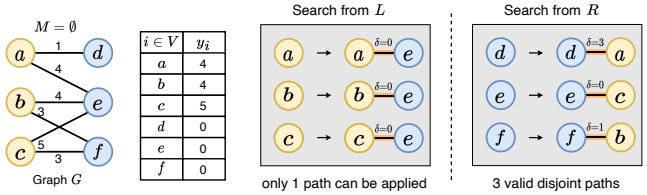


Figure 10: Search trees rooted at unmatched R-side vertices find three disjoint augmenting paths in one iteration

Figure 10 illustrates the same example as Figure 4. If we grow multiple trees from the unmatched L-side vertices a, b, and c, the search returns three augmenting paths (a, e), (b, e), and (c, e). However, only one can be applied because all three share the same endpoint e. In contrast, if we grow trees from the unmatched R-side vertices d, e, and f, each tree finds an augmenting path, and the resulting paths (d, a), (e, c), and (f, b) are disjoint. They can thus be applied together, increasing the matching size by 3 in a single iteration.

Building on this, we develop an adaptive search strategy that dynamically selects the search direction to mitigate path conflicts. We define the *search efficiency* η as the number of disjoint augmenting paths found in an iteration divided by the iteration time. The algorithm tracks the most recent efficiencies for both $L \to R$ and $R \to L$. At the beginning of each iteration, it chooses the direction with the higher recorded efficiency: if $\eta_{L\to R} < \eta_{R\to L}$, it grows trees from unmatched R-side vertices; otherwise, from unmatched L-side vertices. We evaluate three search strategies on random graphs: $L \to R$, $R \to L$, and adaptive search, as shown in Table 1. Compared with searching in only one fixed direction, the adaptive search substantially reduces the number of path conflicts by up to 7.04x. It also reduces the average search tree size by up to 3.11x. Selecting the more effective search direction can lead to a more balanced tree structure and avoid unnecessarily deep or broad tree growth.

Table 2 quantifies the cumulative benefits of the three proposed approaches in the single-threaded setting on the three random graphs. Compared with the basic Hungarian method, phase decoupling (PD) reduces the dual phase time by an average of 3.41x and speeds up the primal phase by up to 1.56x. Based on this phase-decoupled framework, multi-path batching (MP) further reduces the dual phase time by up to 29.1x, since it returns a batch of multiple disjoint augmenting paths per search and performs only one dual label update for each batch. By adopting adaptive search (AS), the primal phase time is further reduced by up to 7.66x, leading to a substantial overall speedup. These show that the three proposed approaches are highly effective even without parallel execution.

Table 1: Performance comparison of search strategies

Random Graph	Search Method	# of Path Conflicts	Avg. Tree Size	Time (ms)
$\|V\| = 40{,}000$ $\|E\| = 1{,}000{,}000$	$L \to R$	21408	238	1173
	$R \to L$	15085	291	912
	Adaptive Search	**4490**	**105**	**182**
$\|V\| = 60{,}000$ $\|E\| = 1{,}800{,}000$	$L \to R$	39084	333	3614
	$R \to L$	35856	305	3197
	Adaptive Search	**5552**	**165**	**475**
$\|V\| = 100{,}000$ $\|E\| = 2{,}500{,}000$	$L \to R$	21778	427	5568
	$R \to L$	25593	475	6324
	Adaptive Search	**6437**	**153**	**1231**

Table 2: Runtime breakdown of the benefits of phase decoupling (PD), multi-path batching (MP), and adaptive search (AS) in the single-threaded setting on three random graphs

Random Graph	Method	Primal (ms)	Dual (ms)	Total (ms)
$\|V\| = 40{,}000$ $\|E\| = 1{,}000{,}000$	Basic Hungarian	2005	2559	4564
	PD	1278	784	2062
	PD+MP	1139	34	1173
	PD+MP+AS	**177**	**5**	**182**
$\|V\| = 60{,}000$ $\|E\| = 1{,}800{,}000$	Basic Hungarian	4635	5656	10291
	PD	3865	1754	5619
	PD+MP	3539	75	3614
	PD+MP+AS	**462**	**13**	**475**
$\|V\| = 100{,}000$ $\|E\| = 2{,}500{,}000$	Basic Hungarian	9973	17827	27800
	PD	8035	4777	12812
	PD+MP	5404	164	5568
	PD+MP+AS	**1199**	**32**	**1231**

6 PERFORMANCE EVALUATION

6.1 Platform Setting

6.1.1 Baselines. We have implemented X-Wim, a general-purpose parallel solver for bipartite MWPM on multicore platforms, and evaluated it against three well-known sequential baselines across various datasets. OR-Tools [92] is a widely used optimization suite from Google with high-performance C++ implemented solvers for weighted matching. Blossom-V [69] is a fast MWPM solver implemented in C++ and serves as a strong classical baseline. LEMON [35] is a well-recognized, highly efficient C++ library that provides optimized routines for weighted matching problems. MCFClass [9] is an efficient C++ solver specialized for minimum cost network flow problems and serves as a flow-based baseline. The time complexity of OR-Tools is $O(|V|^3)$, and that of Blossom-V is $O(|V|^2|E|)$. Both LEMON and MCFClass have the time complexity $O(|V||E| \log |V|)$.

6.1.2 Experiment Environment. All experiments are tested on a server with dual Intel Xeon CPU Max 9470 processors (104 cores in total) and 512 GB of RAM. All baselines are used in their latest versions: OR-Tools (v9.14), Blossom V (v2.05), and LEMON (v1.3.1). All code is compiled with −O3 optimizations using GCC 13.2.0.

6.1.3 Datasets. Both real-world and synthetic datasets are included in the evaluation. Table 3 summarizes a diverse set of real-world graphs collected from the SNAP datasets [75] and the Network

Repository [96], including collaboration graphs (IMDB, Email, MathSciNet), recommendation networks (Yahoo, Epinions), social graphs (YouTube, Dating), and interaction networks (Twitter, Wikipedia). These graphs cover different domains and vary widely in vertex and edge scales, enabling a comprehensive performance study.

Table 3: A summary of real-world graphs

Graph	$\|V\|$	$\|E\|$	Description
Wikipedia [46]	185,848	331,612	Edit network on Wikipedia
Yahoo [96]	273,472	49,907,041	Music recommendation network on Yahoo
Dating [17]	337,584	17,528,006	User rating network on a dating platform
Email [74]	450,818	645,427	Email graph of a European research institution
MathSciNet [91]	622,568	1,131,918	Collaboration graph on MathSciNet
Twitter [33]	846,396	1,156,843	User interaction network on Twitter
Epinions [85]	1,651,320	15,211,614	Product recommendation network on Epinions
IMDB [89]	1,792,604	4,678,762	Actor–movie collaboration graph on IMDB
Youtube [122]	2,268,280	4,121,763	Social network on YouTube

We also generate Erdős–Rényi random graphs [90] and d-regular random graphs [15] as synthetic datasets. In an Erdős–Rényi random graph, edges are selected independently at random to achieve a specified edge density. In a d-regular random graph, each vertex has exact degree d, and the d neighbors are assigned randomly subject to this degree constraint.

6.1.4 Evaluation Method. All graphs are stored in the Compressed Sparse Row (CSR) format [29], including their edge weights. In each random graph category, we generate 10 instances for each specified density (Erdős–Rényi random graphs) or degree (d-regular random graphs). For real-world graphs without edge weights and synthetic graphs, we assign a weight to each edge by sampling uniformly at random from $[0, |V|]$. Each data point is averaged over 20 runs.

6.2 Overall Performance

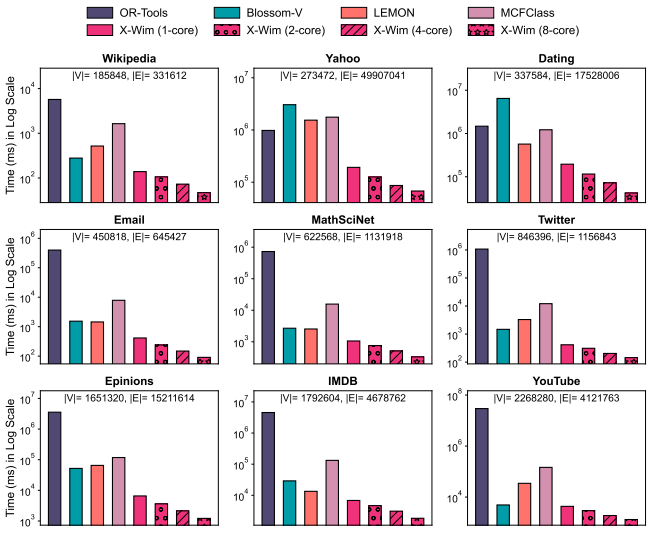


Figure 11: Performance on real-world datasets (in log scale)

Figure 11 presents the performance of all implementations on real-world graphs, with runtime shown on a **log** scale. X-Wim is evaluated by using 1 core (sequential) to 8 cores. The experiments show that X-Wim consistently outperforms all baselines across the datasets. Even when using a single core, X-Wim achieves up to a 9.93x speedup over the state-of-the-art baseline LEMON. Moreover, the log scale plots indicate that, on several large graphs (e.g., Yahoo, Dating, and Epinions), X-Wim is faster than OR-Tools, Blossom-V, and MCFClass by orders of magnitude. These results demonstrate that the phase-decoupled approach, together with finding multiple augmenting paths per iteration and the adaptive search strategy, delivers substantial benefits even without parallel execution.

As the core count increases, X-Wim exhibits steady runtime reductions. Scaling from 1 to 8 cores decreases runtime on all graphs, yielding a speedup of up to 5.4x. With 8 cores, X-Wim achieves up to a 53.6x speedup over LEMON, up to a 153x speedup over Blossom-V, and up to a 110x speedup over MCFClass. The incremental gains are also stable. Across the three scaling steps (from 1 to 2 cores, from 2 to 4 cores, and from 4 to 8 cores), the average speedup on each graph ranges from 1.44x to 1.75x. This trend suggests that X-Wim effectively exploits parallelism by growing multiple search trees concurrently and returning a batch of disjoint paths per iteration. Overall, the robust scaling behavior across diverse real-world graphs indicates that the parallel framework generalizes well and efficiently leverages multicore execution for practical acceleration.

6.3 Random Graphs with Specific Density

In the evaluation on synthetic datasets, Erdős–Rényi random graphs are generated using four different densities from sparse to dense, with vertex count ranging from 200,000 to 800,000. Figure 12 shows the performance of all implementations on these random graphs.

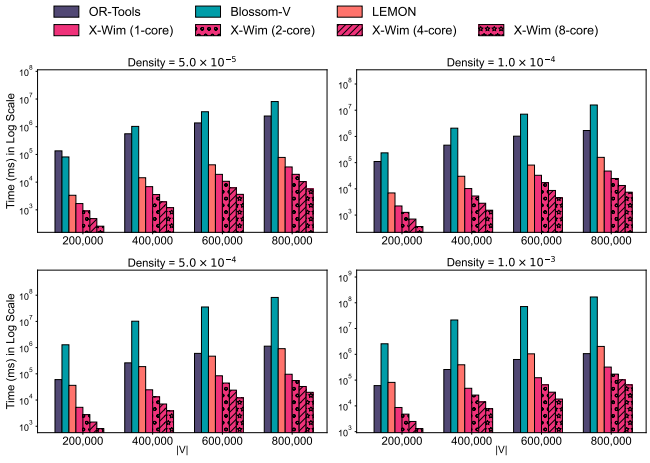


Figure 12: Performance on random graphs at specific density (in log scale), with $|V|$ ranging from 200,000 to 800,000

When the vertex count and the density increase, the 8-core X-Wim achieves stronger performance. The maximum speedup reaches 62.5x and 524x compared to LEMON and OR-Tools, respectively. It also shows that OR-Tools is slower than LEMON in sparse cases but faster in dense cases, indicating a crossover. In contrast, X-Wim consistently surpasses all baselines across all densities and graph sizes, and its advantage becomes more pronounced as the graphs scale. These results highlight X-Wim's efficiency and scalability under different graph densities.

6.4 Random Graphs with Specific Degree

Another part of the synthetic datasets consists of d-regular random graphs. Graphs are generated with four degree settings from 4 to 16 and the vertex count is varied from 1,000,000 to 4,000,000. Figure 13 presents the performance of all implementations except Blossom-V, which is is far slower than OR-Tools.

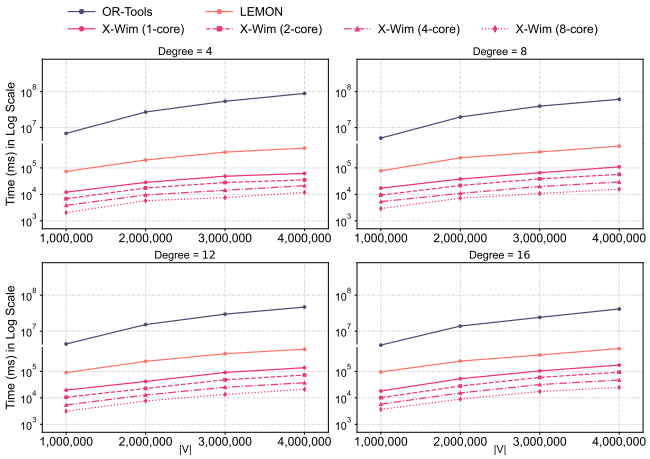


Figure 13: Performance on random graphs at specific degree (in log scale), with $|V|$ ranging from 1,000,000 to 4,000,000

The results show that X-Wim remains the fastest approach across all graphs, and its runtime continues to decrease when the core count increases. With 8 cores, X-Wim is up to 52.4x faster than LEMON and achieves an average 5.98x speedup compared with its 1-core configuration. OR-Tools is orders of magnitude slower than both LEMON and X-Wim. Although LEMON performs much better than OR-Tools, it is still substantially slower than X-Wim even on a single core. As the vertex count grows, the runtime gaps widen, and X-Wim's parallel benefits persist across different degree settings.

6.5 Robustness of Proposed Techniques

We evaluate the effectiveness of the three techniques in X-Wim on graphs with different structures, as shown in Figure 14. The left panel shows results on random graphs with 200,000 vertices at four different densities. We also generate random graphs whose degrees follow a gamma distribution, with 200,000 vertices and a fixed mean degree of 8. The right panel considers four such distributions with shape parameters ranging from 8 to 1. A smaller shape parameter indicates greater degree variation and thus a more irregular graph.

In the left panel, as graph density increases, the speedups of phase decoupling and multi-path batching grow from 1.57x to 2.61x and from 1.85x to 3.65x, respectively, while the benefit of adaptive search remains stable with an average speedup of 8.8x. In the right panel, even when the graphs become more irregular, the performance gains of all three techniques are fairly consistent.

The results across all experiments indicate that the benefits of the sequential X-Wim become more pronounced on larger and denser graphs, since both phase decoupling and multi-path batching can effectively reduce the cost of dual label updates. As the vertex count increases, X-Wim achieves better parallel performance because

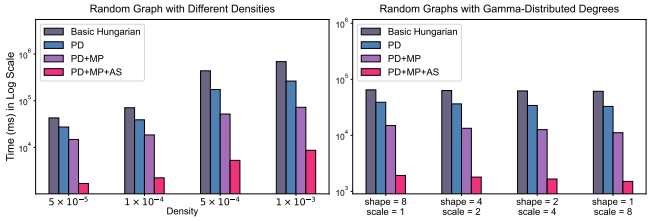


Figure 14: Effectiveness of phase decoupling (PD), multi-path batching (MP), and adaptive search (AS) under different densities and degree skews in random graphs with |V|=200,000

more disjoint augmenting paths can be found concurrently within a single search iteration, which exposes greater parallelism. In addition, X-Wim maintains robust benefits even when degree skew increases and the graph becomes more irregular.

6.6 Memory Footprint

We also compare the peak memory footprints of X-Wim and other baselines on random graphs with two densities and vertex counts ranging from 200,000 to 800,000 in Figure 15. All baselines show similar overall memory usage, while X-Wim uses less memory than others. The savings become more evident as graph size increases. Compared with OR-Tools and LEMON, X-Wim reduces the memory footprint by up to 1.58x and 1.37x, respectively. The additional accumulated slack array in X-Wim incurs negligible overhead relative to graph storage. Without maintaining a global set of tight edges, X-Wim further reduces auxiliary memory usage.

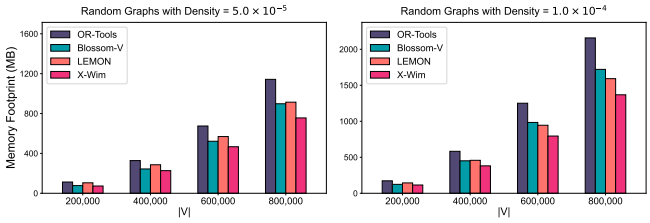


Figure 15: Memory footprint comparison on random graphs at specific density with $|V|$ ranging from 200,000 to 800,000

6.7 Scalability Test

We further conduct a strong scalability study of X-Wim by increasing the number of threads from 1 to 96. Figure 16 reports its performance on four large random graphs with a degree of 4 and the vertex count ranges from 7,000,000 to 10,000,000. Both axes are plotted on a log scale, where the x-axis represents the number of threads and the y-axis represents the computation time.

The four linear trends in the log–log plot intuitively show the strong scaling behavior for X-Wim. When the number of threads doubles, X-Wim achieves an average speedup of 1.64x to 1.76x across the four graphs, with an overall average of 1.70x; larger graphs tend to obtain higher gains. Importantly, this is enabled by the phase-decoupled framework, which eliminates the interleaved execution of the primal and dual phases and creates opportunities to find multiple augmenting paths concurrently.

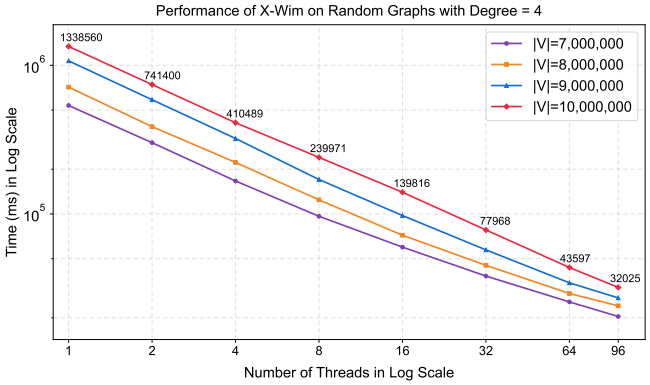


Figure 16: Computation time (in log scale) of X-Wim when the number of threads increases (in log scale)

The experimental server consists of 8 NUMA nodes. We further evaluate NUMA effects using 48 threads on two random graphs, as shown in Table 4. Restricting CPU and memory binding to the first four NUMA nodes yields only limited gains, whereas interleaving memory across these nodes increases the speedup to 2.08x and 2.14x over the baseline. This indicates that distributing memory pages uniformly across NUMA nodes can improve memory bandwidth utilization, thereby achieving higher performance.

Table 4: Impact of NUMA Binding Policies on Performance

Random Graph	# of Threads	CPU Policy	Memory Policy	Time (ms)
$\|V\| = 7{,}000{,}000$ $\|E\| = 28{,}000{,}000$	48	Default	Default	29841
		CPU bind 0–3	Default	28081
		CPU bind 0–3	Mem bind 0–3	25744
		CPU bind 0–3	Interleave 0–3	14316
$\|V\| = 10{,}000{,}000$ $\|E\| = 40{,}000{,}000$	48	Default	Default	54958
		CPU bind 0–3	Default	52798
		CPU bind 0–3	Mem bind 0–3	49020
		CPU bind 0–3	Interleave 0–3	25648

7 RELATED WORK

Bipartite weighted matching is a fundamental graph problem with broad applications, including in database systems [13, 18, 53]. The Hungarian algorithm [71, 88] is the first modern polynomial-time method for solving bipartite MWPM, building on the earlier work of Hungarian mathematicians König and Egerváry. Since then, substantial research [7, 26, 50, 52] has improved its sequential performance through better data structures (e.g., heaps) [47, 65, 112] and specialization to particular settings (e.g., integer-weight instances) [39, 99, 111]. Some studies [37, 100, 115] also explore machine learning or approximation approaches to improve runtime efficiency. The Hungarian algorithm also has an optimized implementation with time complexity $O(|V|^3)$ that incrementally maintains slack values for each candidate vertex to avoid repeated computation of tight edges in each iteration [112]. However, this method still requires selecting the tight edge with the minimum slack value for tree expansion, and thus still incurs frequent dual label update overhead. Moreover, the interleaved data dependencies between the primal and dual phases persist and hinder parallel execution.

There has been limited research on parallel solutions to the Hungarian algorithm [6, 11]. Existing parallel efforts [32, 60, 66, 82] focus on the linear assignment problem, where the input is modeled as a complete graph. All these methods parallelize computations only within each phase, and do not address the dependencies induced by the interleaved primal–dual execution, preventing them from being a practical solution for general bipartite MWPM instances. The most recent work, HyLAC [66], is evaluated in Section 3.1 and only supports graphs with at most 83,000 vertices.

The bipartite MWPM can also be solved using network flow algorithms [41, 45, 70]. A standard approach is to repeatedly find the shortest path in the residual graph and send additional flow along that path until the required flow is reached. After each flow augmentation, the residual capacities and reverse edges must be modified, introducing strong data dependencies. Even when multiple paths are available, their updates must still be coordinated on the shared residual network to preserve flow validity, which further constrains its parallelism. Therefore, this paper focuses on parallelizing the Hungarian algorithm rather than flow-based methods.

Graph data management systems [8, 30, 34, 51, 54, 55, 72, 78, 110, 114, 120, 132] have been widely studied due to their practical importance. Advances in graph algorithms [1, 43, 61, 81, 84, 87, 124, 130, 131] directly benefit database systems by enabling faster and more scalable graph analytics and query processing. However, efficient graph processing is often difficult due to irregular computation and data-dependent control flow [44, 63, 67, 103, 105, 133]. Prior works [5, 20, 23, 24, 62, 94, 107, 116, 123, 126, 129] have addressed a range of bottlenecks, such as synchronization overhead [22, 101, 125], load imbalance [19, 104], and memory footprint [21, 102, 119, 121]. MWPM in general graphs is also an active research area with many algorithmic solutions [38, 40, 48, 49, 93]. Other related topics include subgraph matching [59, 64, 68, 73, 76, 108, 109], unweighted matching [4, 42, 117], and maximal independent set [14, 36, 80], which are distinct from the bipartite MWPM addressed in this paper.

8 CONCLUSION

We have developed X-Wim, a massively parallel framework for solving MWPM in bipartite graphs, along with its implementation on multicore processors. Three structural bottlenecks have been identified from the Hungarian algorithm's intricate execution pattern: inherently sequential tree expansion, frequent phase switching, and the single-path-per-iteration search constraint. These critical issues impede parallelization, yet have not been systematically addressed. By breaking the fine-grained data dependencies between the primal search and the dual label update, we propose a sequential phase-decoupled Hungarian algorithm that eliminates the interleaved primal–dual execution and removes repeated phase switching, providing a crucial basis for parallel computing. Building on this, we develop a phase-decoupled parallel framework that concurrently finds multiple disjoint augmenting paths within each iteration, exploiting massive parallelism. In addition, we introduce an adaptive search strategy that automatically selects the search direction to mitigate path conflicts and further enhance performance. These designs also yield substantial performance benefits in the single-core setting. Extensive experiments demonstrate that X-Wim consistently surpasses existing MWPM solvers. X-Wim, together with its open-source implementation, provides an efficient and scalable solution for computing MWPM on large-scale bipartite graphs.

REFERENCES

[1] Hiba Abu Ahmad and Hongzhi Wang. 2020. Automatic weighted matching rectifying rule discovery for data repairing: Can we discover effective repairing rules automatically from dirty data? *The VLDB Journal* 29, 6 (June 2020), 1433–1447. https://doi.org/10.1007/s00778-020-00617-6

[2] Itai Ashlagi, Maximilien Burq, Patrick Jaillet, and Amin Saberi. 2018. Maximizing Efficiency in Dynamic Matching Markets. *CoRR* abs/1803.01285 (2018). arXiv:1803.01285 http://arxiv.org/abs/1803.01285

[3] Ariful Azad, Aydin Buluç, Xiaoye S. Li, Xinliang Wang, and Johannes Langguth. 2020. A Distributed-Memory Algorithm for Computing a Heavy-Weight Perfect Matching on Bipartite Graphs. *SIAM Journal on Scientific Computing* 42, 4 (2020), C143–C168. https://doi.org/10.1137/18M1189348 arXiv:https://doi.org/10.1137/18M1189348

[4] Ariful Azad, Mahantesh Halappanavar, Sivasankaran Rajamanickam, Erik G. Boman, Arif Khan, and Alex Pothen. 2012. Multithreaded Algorithms for Maximum Matching in Bipartite Graphs. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 860–872. https://doi.org/10.1109/IPDPS.2012.82

[5] Oana Balalau, Francesco Bonchi, T-H. Hubert Chan, Francesco Gullo, Mauro Sozio, and Hao Xie. 2024. Finding Subgraphs with Maximum Total Density and Limited Overlap in Weighted Hypergraphs. *ACM Trans. Knowl. Discov. Data* 18, 4, Article 95 (Feb. 2024), 21 pages. https://doi.org/10.1145/3639410

[6] Egon Balas, Donald Miller, Joseph Pekny, and Paolo Toth. 1991. A parallel shortest augmenting path algorithm for the assignment problem. *J. ACM* 38, 4 (Oct. 1991), 985–1004. https://doi.org/10.1145/115234.115349

[7] M. L. Balinski and R. E. Gomory. 1964. A Primal Method for the Assignment and Transportation Problems. *Management Science* 10, 3 (1964), 578–593. https://doi.org/10.1287/mnsc.10.3.578 arXiv:https://doi.org/10.1287/mnsc.10.3.578

[8] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive query compilation in graph databases. *Distributed and Parallel Databases* 41, 3 (2023), 359–386.

[9] A. Bertolini, A. Frangioni, and C. Gentile. 2011. The MCFClass Project. https://frangio68.github.io/Min-Cost-Flow-Class/.

[10] Dimitri P. Bertsekas. 1992. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications* 1, 1 (1992), 7–66. https://doi.org/10.1007/BF00247653

[11] Dimitri P. Bertsekas and David A. Castañon. 1993. Parallel Asynchronous Hungarian Methods for the Assignment Problem. *ORSA Journal on Computing* 5, 3 (1993), 261–274. https://doi.org/10.1287/ijoc.5.3.261 arXiv:https://doi.org/10.1287/ijoc.5.3.261

[12] Dimitris Bertsimas, Vivek F. Farias, and Nikolaos Trichakis. 2013. Fairness, Efficiency, and Flexibility in Organ Allocation for Kidney Transplantation. *Operations Research* 61, 1 (2013), 73–87. https://doi.org/10.1287/opre.1120.1138 arXiv:https://doi.org/10.1287/opre.1120.1138

[13] A. Bilke and F. Naumann. 2005. Schema matching using duplicates. In *21st International Conference on Data Engineering (ICDE'05)*. 69–80. https://doi.org/10.1109/ICDE.2005.126

[14] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 308–317. https://doi.org/10.1145/2312005.2312058

[15] Béla Bollobás. 2001. Random Graphs. Cambridge University Press, Chapter 2.4. Random Regular Graphs.

[16] Angela Bonifati, M. Tamer Özsu, Yuanyuan Tian, Hannes Voigt, Wenyuan Yu, and Wenjie Zhang. 2024. The Future of Graph Analytics. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 544–545. https://doi.org/10.1145/3626246.3658369

[17] Lukas Brozovsky and Vaclav Petricek. 2007. Recommender system for online dating service. *arXiv preprint cs/0703042* (2007).

[18] Peter Buneman and Sławek Staworko. 2016. RDF graph alignment with bisimulation. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1149–1160. https://doi.org/10.14778/2994509.2994531

[19] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2024. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware. *Proc. ACM Manag. Data* 2, 3, Article 161 (May 2024), 25 pages. https://doi.org/10.1145/3654964

[20] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2023. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 657–670. https://doi.org/10.14778/3636218.3636223

[21] Matteo Ceccarello, Andrea Pietracaprina, Geppino Pucci, and Eli Upfal. 2015. Space and time efficient parallel graph decomposition, clustering, and diameter approximation. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 182–191.

[22] Matteo Ceccarello, Andrea Pietracaprina, Geppino Pucci, and Eli Upfal. 2016. A practical parallel algorithm for diameter approximation of massive weighted graphs. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 12–21.

[23] Xuhao Chen. 2019. GraphCage: Cache Aware Graph Processing on GPUs. arXiv:1904.02241 [cs.DC] https://arxiv.org/abs/1904.02241

[24] Zi Chen, Bo Feng, Long Yuan, Xuemin Lin, and Liping Wang. 2023. Fully dynamic contraction hierarchies with label restrictions on road networks. *Data Science and Engineering* 8, 3 (2023), 263–278.

[25] Zhanzhan Cheng, Jing Lu, Baorui Zou, Liang Qiao, Yunlu Xu, Shiliang Pu, Yi Niu, Fei Wu, and Shuigeng Zhou. 2021. FREE: A Fast and Robust End-to-End Video Text Spotter. *IEEE Transactions on Image Processing* 30 (2021), 822–837. https://doi.org/10.1109/TIP.2020.3038520

[26] J. Cheriyan and K. Mehlhorn. 1996. Algorithms for dense graphs and networks on the random access computer. *Algorithmica* 15, 6 (1996), 521–549. https://doi.org/10.1007/BF01940880

[27] Smriti Chopra, Giuseppe Notarstefano, Matthew Rice, and Magnus Egerstedt. 2017. A Distributed Version of the Hungarian Method for Multirobot Assignment. *IEEE Transactions on Robotics* 33, 4 (2017), 932–947. https://doi.org/10.1109/TRO.2017.2693377

[28] Vassilis Christophides, Vasilis Efthymiou, Themis Palpanas, George Papadakis, and Kostas Stefanidis. 2020. An Overview of End-to-End Entity Resolution for Big Data. *ACM Comput. Surv.* 53, 6, Article 127 (Dec. 2020), 42 pages. https://doi.org/10.1145/3418896

[29] Wikipedia contributors. 2025. Sparse matrix. https://en.wikipedia.org/wiki/Sparse_matrix. Accessed: 2025-01-19.

[30] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph views for efficient graph analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 193–204.

[31] Jonas Dann, Tobias Goetz, Daniel Ritter, Jana Giceva, Holger Fröning, and Gustavo Alonso. 2025. GraphMatch: Subgraph Query Processing on Steroids. *Proc. ACM Manag. Data* 3, 6, Article 332 (Dec. 2025), 26 pages. https://doi.org/10.1145/3769797

[32] Ketan Date and Rakesh Nagi. 2016. GPU-accelerated Hungarian algorithms for the Linear Assignment Problem. *Parallel Comput.* 57 (2016), 52–72. https://doi.org/10.1016/j.parco.2016.05.012

[33] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. 2013. The Anatomy of a Scientific Rumor. *Scientific Reports* 3, 1 (2013), 2980. https://doi.org/10.1038/srep02980

[34] Fernando de Meer Pardo, Claude Lehmann, Dennis Gehrig, Andrea Nagy, Stefano Nicoli, Misheva Branka Hadji, Martin Braschler, and Kurt Stockinger. 2025. Gralmatch: matching groups of entities with graphs and language models. In *28th International Conference on Extending Database Technology (EDBT), Barcelona, Spain, 25-28 March 2025*. Open Proceedings, 1–12.

[35] Balázs Dezső, Alpár Jüttner, and Péter Kovács. 2011. LEMON – an Open Source C++ Graph Template Library. *Electronic Notes in Theoretical Computer Science* 264, 5 (2011), 23–45. https://doi.org/10.1016/j.entcs.2011.06.003 Proceedings of the Second Workshop on Generative Technologies (WGT) 2010.

[36] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1, Article 4 (April 2021), 70 pages. https://doi.org/10.1145/3434393

[37] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. 2021. Faster matchings via learned duals. In *Proceedings of the 35th International Conference on Neural Information Processing Systems (NIPS '21)*. Curran Associates Inc., Red Hook, NY, USA, Article 795, 14 pages.

[38] Ran Duan, Seth Pettie, and Hsin-Hao Su. 2018. Scaling Algorithms for Weighted Matching in General Graphs. *ACM Trans. Algorithms* 14, 1, Article 8 (Jan. 2018), 35 pages. https://doi.org/10.1145/3155301

[39] Ran Duan and Hsin-Hao Su. 2012. A scaling algorithm for maximum weight matching in bipartite graphs. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms* (Kyoto, Japan) *(SODA '12)*. Society for Industrial and Applied Mathematics, USA, 1413–1424.

[40] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of mathematics* 17 (1965), 449–467.

[41] Jack Edmonds and Richard M Karp. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)* 19, 2 (1972), 248–264.

[42] Dayi Fan, Rubao Lee, and Xiaodong Zhang. 2025. X-Blossom: Massive Parallelization of Graph Maximum Matching. *Proc. VLDB Endow.* 18, 10 (June 2025), 3339–3353. https://doi.org/10.14778/3748191.3748199

[43] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 1233–1244. https://doi.org/10.14778/2994509.2994538

[44] Fabian Fier and Johann-Christoph Freytag. 2022. Parallelizing filter-and-verification based exact set similarity joins on multicores. *Information Systems* 108 (2022), 101912. https://doi.org/10.1016/j.is.2021.101912

[45] L. R. Ford and D. R. Fulkerson. 1958. Constructing Maximal Dynamic Flows from Static Flows. *Oper. Res.* 6, 3 (June 1958), 419–433. https://doi.org/10.1287/

opre.6.3.419
[46] Wikimedia Foundation. 2010. Wikimedia Downloads. http://dumps.wikimedia.org/.
[47] Michael L. Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (July 1987), 596–615. https://doi.org/10.1145/28869.28874
[48] Harold N. Gabow. 1985. A scaling algorithm for weighted matching on general graphs. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 90–100. https://doi.org/10.1109/SFCS.1985.3
[49] Harold N. Gabow, Zvi Galil, and Thomas H. Spencer. 1989. Efficient implementation of graph algorithms using contraction. *J. ACM* 36, 3 (July 1989), 540–572. https://doi.org/10.1145/65950.65954
[50] Harold N. Gabow and Robert E. Tarjan. 1989. Faster Scaling Algorithms for Network Problems. *SIAM J. Comput.* 18, 5 (1989), 1013–1036. https://doi.org/10.1137/0218069 arXiv:https://doi.org/10.1137/0218069
[51] Sainyam Galhotra, Donatella Firmani, Barna Saha, and Divesh Srivastava. 2018. Robust Entity Resolution using Random Graphs. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3183713.3183755
[52] Harold N. Garbow. 1985. Scaling algorithms for network problems. *J. Comput. System Sci.* 31, 2 (1985), 148–168. https://doi.org/10.1016/0022-0000(85)90039-X
[53] Sudipto Guha, Nick Koudas, Amit Marathe, and Divesh Srivastava. 2004. Merging the results of approximate match operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 636–647.
[54] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proc. VLDB Endow.* 14, 11 (July 2021), 2491–2504. https://doi.org/10.14778/3476249.3476297
[55] Mohamed S. Hassan, Walid G. Aref, and Ahmed M. Aly. 2016. Graph Indexing for Shortest-Path Finding over Dynamic Sub-Graphs. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1183–1197. https://doi.org/10.1145/2882903.2882933
[56] Xiangnan He, Ming Gao, Min-Yen Kan, and Dingxian Wang. 2017. BiRank: Towards Ranking on Bipartite Graphs. *IEEE Transactions on Knowledge and Data Engineering* 29, 1 (2017), 57–71. https://doi.org/10.1109/TKDE.2016.2611584
[57] Oscar Higgott. 2022. Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching. *ACM Transactions on Quantum Computing* 3, 3 (2022), 1–16.
[58] Gunter J. Hitsch, Ali Hortaçsu, and Dan Ariely. 2010. Matching and Sorting in Online Dating. *American Economic Review* 100, 1 (March 2010), 130–63. https://doi.org/10.1257/aer.100.1.130
[59] Wenzhe Hou, Xiang Zhao, and Bo Tang. 2025. OptMatch: An Efficient and Generic Neural Network-Assisted Subgraph Matching Approach. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 4079–4091. https://doi.org/10.1109/ICDE65448.2025.00304
[60] Cheng Huang, Alexander Mathiasen Graphcore, Josef Dean Graphcore, Davide Mottin, and Ira Assent. 2024. HUNIPU: Efficient Hungarian Algorithm on IPUs. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. 388–394. https://doi.org/10.1109/ICDEW61823.2024.00062
[61] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1311–1322. https://doi.org/10.1145/2588555.2610495
[62] Rana Hussein, Alberto Lerner, Andre Ryser, Lucas David Bürgi, Albert Blarer, and Philippe Cudre-Mauroux. 2023. GraphINC: Graph Pattern Mining at Network Speed. *Proc. ACM Manag. Data* 1, 2, Article 184 (June 2023), 28 pages. https://doi.org/10.1145/3589329
[63] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2018. Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2018), 659–672. https://doi.org/10.1109/TPDS.2017.2763951
[64] Xun Jian, Zhiyuan Li, and Lei Chen. 2023. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* 16, 7 (March 2023), 1699–1711. https://doi.org/10.14778/3587136.3587144
[65] Donald B. Johnson. 1977. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. ACM* 24, 1 (Jan. 1977), 1–13. https://doi.org/10.1145/321992.321993
[66] Samiran Kawtikwar and Rakesh Nagi. 2024. HyLAC: Hybrid linear assignment solver in CUDA. *J. Parallel and Distrib. Comput.* 187 (2024), 104838. https://doi.org/10.1016/j.jpdc.2024.104838
[67] Xiangyu Ke, Arijit Khan, and Leroy Lim Hong Quan. 2019. An in-depth comparison of s-t reliability algorithms over uncertain graphs. *Proc. VLDB Endow.* 12, 8 (April 2019), 864–876. https://doi.org/10.14778/3324301.3324304
[68] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2023. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB journal* 32, 2 (2023), 343–368.
[69] Vladimir Kolmogorov. 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1, 1 (2009), 43–67. https://doi.org/10.1007/s12532-009-0002-8
[70] Péter Kovács. 2015. Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software* 30, 1 (2015), 94–127.
[71] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.
[72] Michael Lan, Xiaoying Wu, and Dimitri Theodoratos. 2022. Answering Graph Pattern Queries using Compact Materialized Views.. In *DOLAP*. 51–60.
[73] Yukyoung Lee, Kyoungmin Kim, Wonseok Lee, and Wook-Shin Han. 2024. In-depth Analysis of Continuous Subgraph Matching in a Common Delta Query Compilation Framework. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–27.
[74] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data* 1, 1 (March 2007), 2–es. https://doi.org/10.1145/1217299.1217301
[75] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. Accessed: 2026-01-12.
[76] Qiyan Li, Jeffrey Xu Yu, and Zongyan He. 2025. Subgraph Matching: A New Decomposition Based Approach. *Proc. VLDB Endow.* 18, 11 (July 2025), 4282–4294. https://doi.org/10.14778/3749646.3749693
[77] Xin Li and Hsinchun Chen. 2013. Recommendation as link prediction in bipartite graphs: A graph kernel-based machine learning approach. *Decision Support Systems* 54, 2 (2013), 880–890. https://doi.org/10.1016/j.dss.2012.09.019
[78] Chunbin Lin, Jianguo Wang, and Yannis Papakonstantinou. 2017. GQFast: Fast graph exploration with context-aware autocompletion. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1389–1390.
[79] Qingmin Liu. 2020. Stability and Bayesian Consistency in Two-Sided Markets. *American Economic Review* 110, 8 (August 2020), 2625–66. https://doi.org/10.1257/aer.20181186
[80] Yu Liu, Jiaheng Lu, Hua Yang, Xiaokui Xiao, and Zhewei Wei. 2015. Towards maximum independent sets on massive graphs. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2122–2133. https://doi.org/10.14778/2831360.2831366
[81] Yu Liu, Bolong Zheng, Xiaodong He, Zhewei Wei, Xiaokui Xiao, Kai Zheng, and Jiaheng Lu. 2017. Probesim: scalable single-source and top-k simrank computations on dynamic graphs. *arXiv preprint arXiv:1709.06955* (2017).
[82] Paulo A.C. Lopes, Satyendra Singh Yadav, Aleksandar Ilic, and Sarat Kumar Patra. 2019. Fast block distributed CUDA implementation of the Hungarian algorithm. *J. Parallel and Distrib. Comput.* 130 (2019), 50–62. https://doi.org/10.1016/j.jpdc.2019.03.014
[83] Lingzhi Luo, Nilanjan Chakraborty, and Katia Sycara. 2015. Distributed Algorithms for Multirobot Task Assignment With Task Deadline Constraints. *IEEE Transactions on Automation Science and Engineering* 12, 3 (2015), 876–888. https://doi.org/10.1109/TASE.2015.2438032
[84] Ruiyao Ma, Yifan Zhu, Baihua Zheng, Lu Chen, Congcong Ge, and Yunjun Gao. 2024. GTI: Graph-Based Tree Index with Logarithm Updates for Nearest Neighbor Search in High-Dimensional Spaces. *Proc. VLDB Endow.* 18, 4 (Dec. 2024), 986–999. https://doi.org/10.14778/3717755.3717760
[85] Paolo Massa and Paolo Avesani. 2005. Controversial users demand local trust metrics: an experimental study on Epinions.com community. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1* (Pittsburgh, Pennsylvania) *(AAAI'05)*. AAAI Press, 121–126.
[86] Aranyak Mehta. 2013. Online Matching and Ad Allocation. *Foundations and Trends® in Theoretical Computer Science* 8, 4 (2013), 265–368. https://doi.org/10.1561/0400000057
[87] Arpit Merchant, Aristides Gionis, and Michael Mathioudakis. 2022. Succinct graph representations as distance oracles: an experimental evaluation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2297–2306. https://doi.org/10.14778/3551793.3551794
[88] James Munkres. 1957. Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics* 5, 1 (1957), 32–38.
[89] Wouter de Nooy, Andrej Mrvar, and Vladimir Batagelj. 2011. *Exploratory social network analysis with Pajek*. Cambridge university press.
[90] Erdős P. and Rényi A. 1959. On random graphs I. *Publicationes Mathematicae* 6, 290-297 (1959), 18.
[91] Gergely Palla, Illés J Farkas, Péter Pollner, Imre Derényi, and Tamás Vicsek. 2008. Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics* 10, 12 (2008), 123026.
[92] Laurent Perron and Vincent Furnon. 2025. *OR-Tools*. Google. https://developers.google.com/optimization/
[93] Eric C Peterson and Peter J Karalekas. 2022. A distributed blossom algorithm for minimum-weight perfect matching. *arXiv preprint arXiv:2210.14277* (2022).
[94] Abdul Quamar, Amol Deshpande, and Jimmy Lin. 2016. NScale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal* 25, 2 (April 2016), 125–150. https://doi.org/10.1007/s00778-015-0405-2

[95] Quazzafi Rabbani, Aamir Khan, and Abdul Quddoos. 2019. Modified Hungarian method for unbalanced assignment problem with multiple jobs. *Appl. Math. Comput.* 361 (2019), 493–498. https://doi.org/10.1016/j.amc.2019.05.041

[96] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 4292–4293.

[97] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431. https://doi.org/10.1145/3186728.3164139

[98] Arezoo Samiei and Liang Sun. 2024. Distributed Matching-By-Clone Hungarian-Based Algorithm for Task Allocation of Multiagent Systems. *IEEE Transactions on Robotics* 40 (2024), 851–863. https://doi.org/10.1109/TRO.2023.3335656

[99] Piotr Sankowski. 2006. Weighted Bipartite Matching in Matrix Multiplication Time. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 274–285.

[100] Madan Sathe. 2012. *Parallel graph algorithms for finding weighted matchings and subgraphs in computational science.* Ph.D. Dissertation. Verlag nicht ermittelbar.

[101] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1906–1917.

[102] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. 2020. Memory-Aware Framework for Efficient Second-Order Random Walk on Large Graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1797–1812. https://doi.org/10.1145/3318464.3380562

[103] Yanyan Shen, Gang Chen, H. V. Jagadish, Wei Lu, Beng Chin Ooi, and Bogdan Marius Tudor. 2014. Fast failure recovery in distributed graph processing systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 437–448. https://doi.org/10.14778/2735496.2735506

[104] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. 2019. Realtime top-k personalized pagerank over large graphs on GPUs. *Proc. VLDB Endow.* 13, 1 (Sept. 2019), 15–28. https://doi.org/10.14778/3357377.3357379

[105] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6, Article 81 (Jan. 2018), 35 pages. https://doi.org/10.1145/3128571

[106] Kyujin Shim, Kangwook Ko, Yujin Yang, and Changick Kim. 2025. Focusing on Tracks for Online Multi-Object Tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11687–11696.

[107] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. Scan++ efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1178–1189.

[108] Konstantinos Skitsas, Davide Mottin, and Panagiotis Karras. 2025. Pilos: Scalable Large-Subgraph Matching by Online Spectral Filtering. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE, 1180–1193.

[109] Siwoo Song, Wonseok Shin, Kunsoo Park, Giuseppe F Italiano, Zhengyi Yang, and Wenjie Zhang. 2025. Efficient Hypergraph Pattern Matching via Match-and-Filter and Intersection Constraint. *arXiv preprint arXiv:2512.10621* (2025).

[110] Georgios Theodorakis, Hugo Firth, James Clarkson, Natacha Crooks, and Jim Webber. 2025. TuskFlow: An Efficient Graph Database for Long-Running Transactions. *Proc. VLDB Endow.* 18, 12 (Aug. 2025), 4777–4790. https://doi.org/10.14778/3750601.3750603

[111] Mikkel Thorup. 2003. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* (San Diego, CA, USA) *(STOC '03)*. Association for Computing Machinery, New York, NY, USA, 149–158. https://doi.org/10.1145/780542.780566

[112] N. Tomizawa. 1971. On some techniques useful for solution of transportation network problems. *Networks* 1, 2 (1971), 173–194. https://doi.org/10.1002/net.3230010206

[113] Kun Tu, Bruno Ribeiro, David Jensen, Don Towsley, Benyuan Liu, Hua Jiang, and Xiaodong Wang. 2014. Online dating recommendations: matching markets and learning preferences. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14 Companion)*. Association for Computing Machinery, New York, NY, USA, 787–792. https://doi.org/10.1145/2567948.2579240

[114] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[115] Yansheng Wang, Yongxin Tong, Cheng Long, Pan Xu, Ke Xu, and Weifeng Lv. 2019. Adaptive Dynamic Bipartite Graph Matching: A Reinforcement Learning Approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1478–1489. https://doi.org/10.1109/ICDE.2019.00133

[116] Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. 2024. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. *Proceedings of the VLDB Endowment* 17, 9 (2024), 2307–2320.

[117] Michael M. Wu and Michael C. Loui. 1990. An efficient distributed algorithm for maximum matching in general graphs. *Algorithmica* 5, 1–4 (June 1990), 383–406. https://doi.org/10.1007/BF01840395

[118] Yue Wu, Namitha Liyanage, and Lin Zhong. 2025. *Micro Blossom: Accelerated Minimum-Weight Perfect Matching Decoding for Quantum Error Correction.* Association for Computing Machinery, New York, NY, USA, 639–654. https://doi.org/10.1145/3676641.3716005

[119] Qian Xu, Juan Yang, Feng Zhang, Zheng Chen, Jiawei Guan, Kang Chen, Ju Fan, Youren Shen, Ke Yang, Yu Zhang, et al. 2024. Improving graph compression for efficient resource-constrained graph analytics. *Proceedings of the VLDB Endowment* 17, 9 (2024), 2212–2226.

[120] Da Yan, James Cheng, M. Tamer Özsu, Fan Yang, Yi Lu, John C. S. Lui, Qizhen Zhang, and Wilfred Ng. 2016. A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.* 9, 7 (March 2016), 564–575. https://doi.org/10.14778/2904483.2904488

[121] Boyu Yang, Weiguo Zheng, Xiang Lian, and Lingfei Zheng. 2025. Space-Efficient Compact Representations for Graph Analytics. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 1–14. https://doi.org/10.1109/ICDE65448.2025.00255

[122] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics* (Beijing, China) *(MDS '12)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2350190.2350193

[123] Renchi Yang, Yidu Wu, Xiaoyang Lin, Qichen Wang, Tsz Nam Chan, and Jieming Shi. 2024. Effective Clustering on Large Attributed Bipartite Graphs. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Barcelona, Spain) *(KDD '24)*. Association for Computing Machinery, New York, NY, USA, 3782–3793. https://doi.org/10.1145/3637528.3671764

[124] Kaiqiang Yu and Cheng Long. 2023. Maximum k-Biplex Search on Bipartite Graphs: A Symmetric-BK Branching Approach. *Proc. ACM Manag. Data* 1, 1, Article 49 (May 2023), 26 pages. https://doi.org/10.1145/3588729

[125] Zihao Yu, Ningyi Liao, and Siqiang Luo. 2024. GENTI: GPU-Powered Walk-Based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs. *Proc. VLDB Endow.* 17, 9 (May 2024), 2269–2278. https://doi.org/10.14778/3665844.3665856

[126] Yuanyuan Zeng, Yixiang Fang, Chenhao Ma, Xu Zhou, and Kenli Li. 2024. Efficient Distributed Hop-Constrained Path Enumeration on Large-Scale Graphs. *Proc. ACM Manag. Data* 2, 1, Article 22 (March 2024), 25 pages. https://doi.org/10.1145/3639277

[127] Stefanos A. Zenios, Glenn M. Chertow, and Lawrence M. Wein. 2000. Dynamic Allocation of Kidneys to Candidates on the Transplant Waiting List. *Operations Research* 48, 4 (2000), 549–569. https://doi.org/10.1287/opre.48.4.549.12418 arXiv:https://doi.org/10.1287/opre.48.4.549.12418

[128] Peng Zhang, Juntao Ma, and Xiaorong Sun. 2008. Intelligent delivery of interactive advertisement content. *Bell Labs Technical Journal* 13, 3 (2008), 143–158. https://doi.org/10.1002/bltj.20330

[129] Zhuowei Zhao, Junhao Gan, Jianzhong Qi, and Zhifeng Bao. 2024. Efficient Example-Guided Interactive Graph Search. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 342–354.

[130] Weiguo Zheng, Lei Zou, Lei Chen, and Dongyan Zhao. 2017. Efficient SimRank-Based Similarity Join. *ACM Trans. Database Syst.* 42, 3, Article 16 (July 2017), 37 pages. https://doi.org/10.1145/3083899

[131] Weiguo Zheng, Lei Zou, Xiang Lian, Dong Wang, and Dongyan Zhao. 2013. Graph similarity search with edit distance constraint in large graph databases. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 1595–1600. https://doi.org/10.1145/2505515.2505723

[132] Yunjia Zheng, Charlotte Sacré, Mohanna Shahrad, Owen Lipchitz, Yu Ting Gu, and Bettina Kemme. 2025. G-View: View Management for Graph Databases. *Proc. VLDB Endow.* 18, 6 (Feb. 2025), 1730–1742. https://doi.org/10.14778/3725688.3725702

[133] Chaoji Zuo and Dong Deng. 2023. ARKGraph: All-Range Approximate K-Nearest-Neighbor Graph. *Proc. VLDB Endow.* 16, 10 (June 2023), 2645–2658. https://doi.org/10.14778/3603581.3603601