Weighted Matching in a Poly-Streaming Model

Ahammed Ullah
Purdue University, West Lafayette, IN, USA

S M Ferdous
Pacific Northwest National Laboratory, Richland, WA, USA

Alex Pothen
Purdue University, West Lafayette, IN, USA

Abstract

We introduce the *poly-streaming model*, a generalization of streaming models of computation in which k processors process k data streams containing a total of N items. The algorithm is allowed $\mathcal{O}(f(k) \cdot M_1)$ space, where M_1 is either $o(N)$ or the space bound for a sequential streaming algorithm. Processors may communicate as needed. Algorithms are assessed by the number of passes, per-item processing time, total runtime, space usage, communication cost, and solution quality.

We design a *single-pass* algorithm in this model for approximating the *maximum weight matching* (MWM) problem. Given k edge streams and a parameter $\varepsilon > 0$, the algorithm computes a $(2+\varepsilon)$-approximate MWM. We analyze its performance in a shared-memory parallel setting: for any constant $\varepsilon > 0$, it runs in time $\widetilde{\mathcal{O}}(L_{max} + n)$, where n is the number of vertices and L_{max} is the maximum stream length. It supports $\mathcal{O}(1)$ per-edge processing time using $\widetilde{\mathcal{O}}(k \cdot n)$ space. We further generalize the design to hierarchical architectures, in which k processors are partitioned into r groups, each with its own shared local memory. The total intergroup communication is $\widetilde{\mathcal{O}}(r \cdot n)$ bits, while all other performance guarantees are preserved.

We evaluate the algorithm on a shared-memory system using graphs with trillions of edges. It achieves substantial speedups as k increases and produces matchings with weights significantly exceeding the theoretical guarantee. On our largest test graph, it reduces runtime by nearly two orders of magnitude and memory usage by five orders of magnitude compared to an offline algorithm.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Mathematics of computing → Approximation Algorithms

Keywords and phrases Streaming Algorithms, Matchings, Graphs, Parallel Algorithms

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.15

Supplementary Material *Software*: `https://github.com/ahammed-ullah/algodyssey`

Funding *Ahammed Ullah*: Research supported by grant SC-0022260 of the Advanced Scientific Computing Research program of the U. S. Department of Energy.
S M Ferdous: Laboratory Directed Research and Development Program at PNNL.
Alex Pothen: Research supported by grant SC-0022260 of the Advanced Scientific Computing Research program of the U. S. Department of Energy.

1 Introduction

Data-intensive computations arise in data science, machine learning, and science and engineering disciplines. These datasets are often massive, generated dynamically, and, when stored, kept in distributed formats on disks, making them amenable to processing as multiple data streams. The modular feature of these datasets can be exploited by streaming algorithms designed for tightly-coupled shared-memory and distributed-memory multiprocessors to efficiently solve large problem instances that offline algorithms cannot handle due to their high memory requirements. However, the design of parallel algorithms that process multiple data streams concurrently has not yet received much attention.

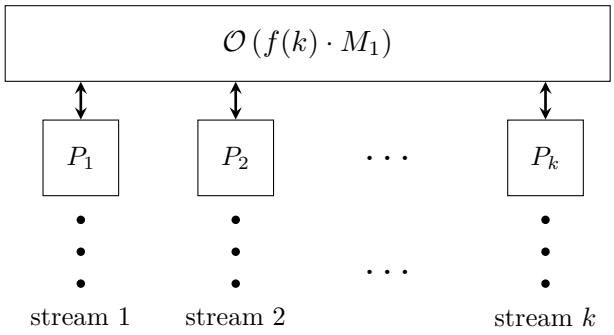

Figure 1 A schematic diagram of the *poly-streaming model* for shared-memory parallel computers. Processors $\{P_l\}_{l \in [k]}$ have access to $\mathcal{O}(f(k) \cdot M_1)$ memory collectively, depicted with the rectangle connected to the processors.

Current multicore shared-memory processors consist of up to a few hundred cores, organized hierarchically to share caches and memory controllers. These cores compute in parallel to achieve speedups over serial execution. With multiple memory controllers, I/O operations can also proceed in parallel, and this feature can be used to process multiple data streams concurrently. These I/O capabilities and the limitations of offline algorithms motivate a model of computation, illustrated in Figure 1 and discussed next.

The streaming model of computation allows $o(N)$ space for a data stream of size N [2, 13]. For graphs, the *semi-streaming model* permits $\mathcal{O}(n \cdot \text{polylog } n)$ space for a graph with n vertices and an edge stream of arbitrary length [8]. Building on these space-constrained models, we introduce the *poly-streaming model*. The key aspects of our model are as follows.

We consider k data streams that collectively contain N items. An algorithm has access to k (abstract) processors, and is allowed $\mathcal{O}(f(k) \cdot M_1)$ total space, where M_1 is either $o(N)$ or the space permitted to a single-stream algorithm. In each pass, each stream is assigned to one of the processors, and each processor independently reads one item at a time from its stream and processes it. Processors may communicate as needed, either via shared or remote memory access. Algorithms are assessed on several metrics: space complexity, number of passes, per-item processing time, total runtime, communication cost, and solution quality.

In the poly-streaming model, we address the problem of approximating a *maximum weight matching* (MWM) in an edge-weighted graph, where the goal is to find a set of vertex-disjoint edges with maximum total weight. We design an algorithm for approximating an MWM when the graph is presented as multiple edge streams. Our design builds on the algorithm of [20] and adds support for handling multiple streams concurrently. We also generalize the design to NUMA (non-uniform memory access) multiprocessor architectures.

We summarize our contributions to the MWM problem as follows. Let L_{max} and L_{min} denote the maximum and minimum lengths of the input streams, respectively, and let n denote the number of vertices in a graph G. For any realization of the CREW PRAM model (such as in Figure 1), we have the following result.

▶ **Theorem 1.** *For any constant $\varepsilon > 0$, there exists a* single-pass *poly-streaming algorithm for the maximum weight matching problem that achieves a $(2+\varepsilon)$-approximation. It admits a CREW PRAM implementation with $\widetilde{\mathcal{O}}(L_{max} + n)$ runtime.*[1] *If $L_{min} = \Omega(n)$, the algorithm achieves $\mathcal{O}(\log n)$ amortized per-edge processing time using $\widetilde{\mathcal{O}}(k + n)$ space. For arbitrarily*

[1] $\widetilde{\mathcal{O}}(\cdot)$ hides polylogarithmic factors.

balanced streams, it uses either $\widetilde{\mathcal{O}}(k+n)$ *space and* $\widetilde{\mathcal{O}}(n)$ *per-edge processing time, or* $\widetilde{\mathcal{O}}(k \cdot n)$ *space and* $\mathcal{O}(1)$ *per-edge processing time.*

In NUMA architectures, memory access costs depend on a processor's proximity to the target memory. We generalize the algorithm in Theorem 1 to account for these cost differences. In particular, we show that when k processors are partitioned into r groups, each with its own shared local memory, the total number of global memory accesses across all groups is $\widetilde{\mathcal{O}}(r \cdot n)$. This generalization preserves all other performance guarantees from Theorem 1, except that the $\widetilde{\mathcal{O}}(k+n)$ space bound becomes $\widetilde{\mathcal{O}}(k+r \cdot n)$. These results are formalized in Theorem 18 in Section 4. This design gives a memory-efficient algorithm for the NUMA shared memory multiprocessors, on which we report empirical results.

We have evaluated our algorithm on a NUMA machine using graphs with billions to trillions of edges. For most of these graphs, our algorithm uses space that is orders of magnitude smaller than that required by offline algorithms. For example, storing the largest graph in our evaluation would require more than 91,600 GB (≈ 90 TB), whereas our algorithm used less than 1 GB. Offline matching algorithms typically require even more memory to accommodate their auxiliary data structures.

We employ approximate dual variables that correspond to a linear programming relaxation of MWM to obtain *a posteriori* upper bounds on the weights of optimal matchings. These bounds allow us to compare the weight of a matching produced by our algorithm with the optimal weight. Thus, we show that our algorithm produces matchings whose weights significantly exceed the approximation guarantee.

For $k = 128$, our algorithm achieves runtime speedups of 16–83 across all graphs in our evaluation, on a NUMA machine with only 8 memory controllers. This is significant scaling for a poly-streaming algorithm, given that 8 memory controllers are not sufficient to serve the concurrent and random access requests of 128 processors without delays. Nevertheless, these speedups demonstrate the effectiveness of our design, which accounts for a processor's proximity to the target memory. A metric less influenced by memory latency suggests that the algorithm would achieve even better speedups on architectures with more efficient memory access.

Note that Theorem 1 and Theorem 18 both guarantee $\widetilde{\mathcal{O}}(L_{max}+n)$ runtime. For $L_{max} = \Omega(n)$, this is tight up to polylogarithmic factors. However, by using $\widetilde{\mathcal{O}}(k \cdot n)$ space and $\mathcal{O}(1)$ per-edge processing time, we can even achieve $\widetilde{\mathcal{O}}(L_{max}+n/k)$ runtime, which becomes polylogarithmic for large values of k (see the arXiv version).

Organization. Section 2 introduces necessary background. Section 3 presents the design and analyses of our algorithm in Theorem 1. In Section 4, we extend the design to NUMA architectures. Section 5 summarizes the evaluation results. We conclude in Section 6 with a discussion of future research directions.

2 Preliminaries

For a graph $G = (V, E)$, let $n := |V|$ and $m := |E|$ denote the number of vertices and edges, respectively. We denote an edge $e := \{u, v\}$ by the unordered pair of its endpoints. For a weighted graph, let w_e denote the weight of edge e, and for any subset $A \subseteq E$, define $w(A) := \sum_{e \in A} w_e$. For $\ell \in [k]$, let E^ℓ be the set of edges received in the ℓth stream. Define $L_{max} := \max_{\ell \in [k]} |E^\ell|$ and $L_{min} := \min_{\ell \in [k]} |E^\ell|$.

A *matching* $\mathcal{M}$ in a graph is a set of edges that do not share endpoints. A *maximum weight matching* (MWM) $\mathcal{M}^*$ is a matching with maximum total weight; that is, $w(\mathcal{M}^*) \geq w(\mathcal{M})$ for all matchings $\mathcal{M} \subseteq E$.

Primal LP

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} w_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(u)} x_e \leq 1, \quad \text{for all } u \in V \\ & x_e \geq 0, \quad \text{for all } e \in E \end{aligned}$$

Dual LP

$$\begin{aligned} \text{minimize} \quad & \sum_{u \in V} y_u \\ \text{subject to} \quad & \sum_{u \in e} y_u \geq w_e, \quad \text{for all } e \in E \\ & y_u \geq 0, \quad \text{for all } u \in V \end{aligned}$$

Figure 2 The linear programming (LP) relaxations of the MWM problem and its dual.

A ρ-approximation algorithm computes a solution whose value is within a factor ρ of optimal. The factor ρ is called the (worst-case) *approximation ratio*. We assume $\rho \geq 1$ for both maximization and minimization problems. Thus, for maximization, a ρ-approximation guarantees a solution whose value is at least $\frac{1}{\rho}$ times the optimal.

We use the linear programming (LP) relaxation of the MWM problem, and its dual, shown in Figure 2. In the primal LP, each variable x_e is 1 if edge e is in the matching and 0 otherwise. Each y_u is a dual variable, and $\delta(u)$ denotes the set of edges incident on a vertex u. Let $\{x_e\}_{e \in E}$ and $\{y_u\}_{u \in V}$ be feasible solutions to the primal and dual LPs, respectively. By weak LP duality, we have $\sum_{e \in E} w_e x_e \leq \sum_{u \in V} y_u$. If $\{x_e\}_{e \in E}$ is an optimal solution to the primal LP, then $w(\mathcal{M}^*) \leq \sum_{e \in E} w_e x_e \leq \sum_{u \in V} y_u$. The first inequality holds because the primal LP is a relaxation of the MWM problem.

3 Algorithms for Uniform Memory Access Cost

In this section, we present the design and analyses of our algorithm in Theorem 1 that assumes a uniform memory access cost.

3.1 The Algorithm

Several semi-streaming algorithms have been designed for the MWM problem [3, 4, 6, 8, 10, 11, 18, 20, 23] (see the arXiv version for brief descriptions of these algorithms). In this paper, we focus exclusively on the single-pass setting in the *poly-streaming* model. Our starting point is the algorithm of Paz and Schwartzman [20], which computes a $(2+\varepsilon)$-approximation of MWM. This is currently the best known guarantee in the single-pass setting under arbitrary or adversarial ordering of edges.[2] We extend a primal-dual analysis by Ghaffari and Wajc [11] to analyze our algorithm.

The algorithm of Paz and Schwartzman [20] proceeds as follows. Initialize an empty stack S and set $\alpha_u = 0$ for each vertex $u \in V$. For each edge $e = \{u, v\}$ in the edge stream, skip e if $w_e < (1+\varepsilon)(\alpha_u + \alpha_v)$. Otherwise, compute $g_e = w_e - (\alpha_u + \alpha_v)$, push e onto the stack S, and increase both α_u and α_v by g_e. After processing all edges, compute a matching $\mathcal{M}$ greedily by popping edges from S.

Note that for each edge pushed onto the stack, the increment $g_e = w_e - (\alpha_u + \alpha_v)$ satisfies $g_e \geq \varepsilon(\alpha_u + \alpha_v)$. This ensures that both α_u and α_v increase by a factor of $1+\varepsilon$. Hence, the number of edges in the stack incident to any vertex is at most $\log_{1+\varepsilon}(W) = \mathcal{O}\left(\frac{\log W}{\varepsilon}\right)$,

[2] No single-pass algorithm can achieve an approximation ratio better than $1 + \ln 2 \approx 1.7$; see [15].

PS-MWM(V, ℓ, ε)**:**

/* each processor executes this algorithm concurrently */
1. In parallel initialize $lock_u$, and set α_u and $mark_u$ to 0 for all $u \in V$
   /* processor ℓ initializes or sets $\Theta(n/k)$ locks/variables */
2. $S^\ell \leftarrow \varnothing$ /* initialize an empty stack */
3. for each edge $e = \{u, v\}$ in ℓth stream do
   a. Process-Edge(e, S^ℓ, ε)
4. wait for all processors to complete execution of Step 3 /* a barrier */
5. $\mathcal{M}^\ell \leftarrow$ Process-Stack(S^ℓ)
6. return $\mathcal{M}^\ell$

Figure 3 A poly-streaming matching algorithm.

where W is the (normalized) maximum edge weight. Therefore, the total number of edges in the stack is $\mathcal{O}\left(\frac{n \log W}{\varepsilon}\right) = \mathcal{O}\left(\frac{n \log n}{\varepsilon}\right)$.[3]

To design a poly-streaming algorithm, we begin with a simple version and then refine it. All k processors share a global stack and a set of variables $\{\alpha_u\}_{u \in V}$, and each processor runs the above sequential streaming algorithm on its respective stream. To complete and adapt this setup for efficient execution across multiple streams, we must address two interrelated issues: (1) concurrent edge arrivals across streams may lead to contention for the shared stack or variables, and (2) concurrent updates to the shared variables may lead to inconsistencies in their observed values.

A natural approach to addressing these issues is to enforce a fair sequential strategy, where processors access shared resources in a round-robin order. While this ensures progress, it incurs $\mathcal{O}(k)$ per-edge processing time, which scales poorly with increasing k. Instead, we adopt fine-grained contention resolution that avoids global coordination by allowing processors to operate asynchronously. However, under the initial setup, this leads to $\widetilde{\mathcal{O}}(n/\varepsilon)$ per-edge processing time: a processor may be blocked from accessing shared resources until the stack has accumulated its $\widetilde{\mathcal{O}}(n/\varepsilon)$ potential edges. We address these limitations with the following design choices.

For the first issue, we observe that a global ordering of edges, as used in the single-stack solution, is not necessary; local orderings within multiple stacks suffice. In particular, we can identify a subset of edges (later referred to as *tight edges*) for which maintaining local orderings is sufficient to compute a $(2+\varepsilon)$-approximate MWM. Hence, we can localize computation using k stacks, assigning one stack to each processor exclusively during the streaming phase. This design eliminates the $\widetilde{\mathcal{O}}(n/\varepsilon)$ contention associated with a shared stack.

However, contention still arises when updating the variables $\{\alpha_u\}_{u \in V}$. It is unclear how to resolve this contention without using additional space. Hence, we consider two strategies for processing edge streams that illustrate the trade-off between space and per-edge processing

[3] Throughout the paper, we assume $W = \mathcal{O}(poly(n))$. For arbitrary weights on edges, we can skip any edge whose weight is less than $\frac{\varepsilon W_{max}}{2(1+\varepsilon)n^2}$, where W_{max} denotes the maximum edge weight observed so far in the stream. This ensures that the (normalized) maximum weight the algorithm sees is $\mathcal{O}\left(n^2/\varepsilon\right)$, while maintaining a $2(1+\mathcal{O}(\varepsilon))$ approximation ratio (see [11] for details).

Process-Edge$(e = \{u, v\}, S^\ell, \varepsilon)$**:**

/* Assumes access to global variables $\{\alpha_u\}_{u \in V}$ and locks $\{lock_u\}_{u \in V}$ */

1. if $w_e \leq (1+\varepsilon)(\alpha_u + \alpha_v)$ then return
2. repeatedly try to acquire $lock_u$ and $lock_v$ in lexicographic order of u and v as long as $w_e > (1+\varepsilon)(\alpha_u + \alpha_v)$
3. if $w_e > (1+\varepsilon)(\alpha_u + \alpha_v)$ then
   a. $g_e \leftarrow w_e - (\alpha_u + \alpha_v)$
   b. increment α_u and α_v by g_e
   c. add e to the top of S^ℓ along with g_e
4. release $lock_u$ and $lock_v$, and return

Figure 4 A subroutine used in algorithms PS-MWM and PS-MWM-LD.

time. In the first, which we call the *non-deferrable strategy*, the decision to include an edge in a stack is made immediately during streaming. In the second, which we call the *deferrable strategy*, this decision may be deferred to post-processing. The latter strategy requires more space but achieves $\mathcal{O}(1)$ per-edge processing time.

To address the second issue, which concerns the potential for inconsistencies due to concurrent updates to the variables $\{\alpha_u\}_{u \in V}$, we observe that the variables are monotonically increasing and collectively require only $\widetilde{\mathcal{O}}(n/\varepsilon)$ updates. Thus, for most edges that are not eligible for the stacks, decisions can be made by simply reading the current values of the relevant variables. However, for the $\widetilde{\mathcal{O}}(n/\varepsilon)$ edges that are included in the stacks, we must update the corresponding variables. To ensure consistency of these updates, we associate a lock with each variable in $\{\alpha_u\}_{u \in V}$. We maintain $|V|$ exclusive locks and allow a variable to be updated only after acquiring its corresponding lock.[4]

We now outline the non-deferrable strategy of our poly-streaming algorithm for the MWM problem (for the deferrable strategy see the arXiv version). For simplicity, we assume that if a processor attempts to release a lock it did not acquire, the operation has no effect. We also assume that any algorithmic step described with the "in parallel" construct includes an implicit barrier (or synchronization primitive) at the end, synchronizing the processors participating in that step.

The non-deferrable strategy is presented in Algorithm PS-MWM, with two subroutines used by PS-MWM described in Process-Edge (Figure 4) and Process-Stack (Figure 5). In PS-MWM, Steps 1–2 form the preprocessing phase, Steps 3–4 the streaming phase, and Step 5 the post-processing phase. Each processor $\ell \in [k]$ executes PS-MWM asynchronously, except that all processors begin the post-processing phase simultaneously (due to Step 4) and then resume asynchronous execution.

In the subroutine Process-Edge, Step 2 ensures that all edges are processed using the non-deferrable strategy: a processor repeatedly attempts to acquire the locks corresponding to the endpoints of an edge $e = \{u, v\}$ until it succeeds, or the edge becomes ineligible for inclusion in a stack. As a result, a processor executing Step 3, has a consistent view of the variables α_u and α_v. In Step 3(c), we store the *gain* g_e of an edge e along with the edge itself for use in the post-processing phase.

[4] This corresponds to the concurrent-read exclusive-write (CREW) paradigm of the PRAM model.

Process-Stack(S^ℓ)**:**

/* Assumes access to global variables $\{\alpha_u\}_{u \in V}$; and $\{mark_u\}_{u \in V}$, initialized in Algorithm PS-MWM (or PS-MWM-LD). */

1. $\mathcal{M}^\ell \leftarrow \varnothing$
2. while $S^\ell \neq \varnothing$ do
   a. remove the top edge $e = \{u, v\}$ of S^ℓ
   b. if $w_e + g_e < \alpha_u + \alpha_v$ then wait for e to be a tight edge
      /* e is a *tight edge* if $w_e + g_e = \alpha_u + \alpha_v$ */
   c. if both $mark_u$ and $mark_v$ are set to 0 then
      /* no locking is needed since e is a tight edge */
      i. $\mathcal{M}^\ell \leftarrow \mathcal{M}^\ell \cup \{e\}$
      ii. set $mark_u$ and $mark_v$ to 1
   d. decrement α_u and α_v by g_e
3. return $\mathcal{M}^\ell$

Figure 5 A subroutine used in algorithms PS-MWM and PS-MWM-LD.

When all k processors are ready to execute Step 5 of PS-MWM, the k stacks collectively contain all the edges needed to construct a $(2+\varepsilon)$-approximate MWM, which can be obtained in several ways. In the subroutine Process-Stack, we outline a simple approach based on local edge orderings. We define an edge $e = \{u, v\}$ in a stack to be a *tight edge* if $w_e + g_e = \alpha_u + \alpha_v$. Equivalently, an edge is tight if and only if all of its neighboring edges that were included after it in any stack have already been removed. Any set of tight edges can be processed concurrently, regardless of their positions in the stacks. In Process-Stack, we simultaneously process the tight edges that appear at the tops of the stacks.

3.2 Analyses

We now formally characterize several correctness properties of the algorithm and analyze its performance. These correctness properties include the absence of deadlock, livelock, and starvation. The performance metrics are space usage, approximation ratio, per-edge processing time, and total runtime.

To simplify the analysis, we assume that processors operate in a quasi-synchronous manner. In particular, to analyze Step 3 of Algorithm PS-MWM, we define an algorithmic *superstep* as a unit comprising a constant number of elementary operations.

▶ **Definition 2** (Superstep). *A processor takes one* superstep *for an edge if it executes Process-Edge with at most one iteration of the loop in Step 2 (i.e., without contention), requiring $\mathcal{O}(1)$ elementary operations. Each additional iteration of Step 2 due to contention adds one superstep, with each such iteration also requiring $\mathcal{O}(1)$ operations.*

▶ **Definition 3** (Effective Iterations). Effective iterations *is the maximum number of supersteps taken by any processor during the execution of Step 3 of Algorithm PS-MWM.*

Note that for $k = 1$, the effective iterations is equal to the number of edges in the stream. Using this notion, we align the supersteps of different processors and define the following directed graph.

▶ **Definition 4** ($G^{(t)}$). *For the* tth *effective iteration, consider the set of edges processed across all* k *streams. Let* $e_\ell = (u_\ell, v_\ell)$ *denote an edge processed in* ℓ*th stream, where* u_ℓ *precedes* v_ℓ *in the lexicographic ordering of the vertices. If processor* ℓ *is idle in the* tth *iteration, then* $e_\ell = \varnothing$. *Define* $G^{(t)} := \left(V^{(t)}, E^{(t)}\right)$, *where* $E^{(t)} := \{e_\ell \mid \ell \in [k]\}$ *and* $V^{(t)} := \bigcup_{(u_\ell, v_\ell) \in E^{(t)}} \{u_\ell, v_\ell\}$.

The following property of $G^{(t)}$ is straightforward to verify.

▶ **Proposition 5.** *$G^{(t)}$ is a directed acyclic graph.*

We show that Algorithm PS-MWM is free from deadlock, livelock, and starvation. *Deadlock* occurs when a set of processors forms a cyclic dependency, with each processor waiting for a resource held by another. *Livelock* occurs when a set of processors repeatedly form such a cycle, where each processor continually acquires and releases resources without making progress. *Starvation* occurs when a processor waits indefinitely for a resource because other processors repeatedly acquire it first. The following lemma shows that the streaming phase of Algorithm PS-MWM is free from deadlock, livelock, and starvation.

▶ **Lemma 6.** *The concurrent executions of the subroutine Process-Edge are free from deadlock, livelock, and starvation.*

Proof. Since the variables $\{\alpha_u\}_{u \in V}$ are updated only while holding their corresponding locks, we treat the locks $\{lock_u\}_{u \in V}$ as the only shared resources in Process-Edge.

Let $G^{(t)}$ be the graph defined in Definition 4. By Proposition 5, $G^{(t)}$ is a directed acyclic graph (DAG), and hence each of its components is also a DAG.

To reason about cyclic dependencies, we focus on components of $G^{(t)}$ involving processors executing Step 2 of Process-Edge. Every DAG contains at least one vertex with no outgoing edges. Thus, each such component includes an edge $e_\ell = (u_\ell, v_\ell)$ such that only processor ℓ requests $lock_{v_\ell}$. This precludes the possibility of cyclic dependencies; that is, the concurrent executions of Process-Edge is free from deadlock and livelock.

To show that starvation does not occur, suppose an edge appears in every effective iteration $t \in [a, b]$, that is, $e_\ell = (u_\ell, v_\ell) \in \bigcap_{t \in [a,b]} E^{(t)}$. We show that $b - a = \widetilde{\mathcal{O}}\left(n/\varepsilon\right)$, which bounds the number of supersteps that processor ℓ may spend attempting to acquire locks for e_ℓ. Step 2 requires one superstep per iteration, while all other steps collectively require at most one. For each $t \in (a, b]$, the component of $G^{(t-1)}$ containing e_ℓ has at least one vertex with no outgoing edge. This guarantees that at least one edge in that component acquires its locks and completes Step 3 during the $(t-1)$th effective iteration. Since Step 3 can increment the values in $\{\alpha_u\}_{u \in V}$ for at most $\mathcal{O}\left(n \log_{1+\varepsilon} W\right) = \widetilde{\mathcal{O}}\left(n/\varepsilon\right)$ edges over the entire execution, the number of iterations for which e_ℓ may remain blocked is also bounded by $\widetilde{\mathcal{O}}\left(n/\varepsilon\right)$. ◀

To analyze Step 5 of Algorithm PS-MWM, we adopt the same simplification: processors are assumed to operate in a quasi-synchronous manner. Accordingly, we define $\mathcal{U}^{(t)}$ as the set of edges present in the stacks $\bigcup_{\ell \in [k]} S^\ell$ at the beginning of iteration t of Step 2 in Process-Stack. The following definition is useful for characterizing tight edges via an equivalent notion.

▶ **Definition 7** (Follower). *An edge* $e_j \in \mathcal{U}^{(t)}$ *is a* follower *of an edge* $e_i \in \mathcal{U}^{(t)}$ *if* $e_i \cap e_j \neq \varnothing$ *and* e_j *is added to some stack* S^j *after* e_i *is added to some stack* S^i. *We denote the set of followers of an edge* e *by* $\mathcal{F}(e)$.

The proofs of the following four lemmas are included in the arXiv version. The fourth lemma establishes that the post-processing phase of PS-MWM is free from deadlock, livelock, and starvation.

▶ **Lemma 8.** *An edge e is a tight edge if and only if $\mathcal{F}(e) = \varnothing$.*

▶ **Lemma 9.** *Let $\mathcal{T}^{(t)}$ be the set of top edges in the stacks at the beginning of iteration t of Step 2 of Process-Stack. Then $\mathcal{T}^{(t)}$ contains at least one tight edge.*

▶ **Lemma 10.** *The set of tight edges in $\mathcal{U}^{(t)}$ is vertex-disjoint.*

▶ **Lemma 11.** *The concurrent executions of the subroutine Process-Stack are free from deadlock, livelock, and starvation.*

We now analyze the performance metrics of the algorithm.

▶ **Lemma 12.** *For any constant $\varepsilon > 0$, the space complexity and per-edge processing time of Algorithm PS-MWM are $\mathcal{O}(k + n \log n)$ and $\mathcal{O}(n \log n)$, respectively. Furthermore, for $L_{min} = \Omega(n)$, the amortized per-edge processing time of the algorithm is $\mathcal{O}(\log n)$.*

Proof. The claimed space bound follows from three components: $\mathcal{O}(n)$ space for the variables and locks, $\mathcal{O}(n \log n)$ space for the stacked edges, and $\mathcal{O}(1)$ space per processor.

The worst-case per-edge processing time follows from the second part of the proof of Lemma 6.

Processor ℓ processes $|E^\ell|$ edges, each requiring at least one distinct effective iteration (see Definition 3). Additional iterations may arise when it repeatedly attempts to acquire locks in Step 2 of Process-Edge. From the second part of the proof of Lemma 6, the total number of such additional iterations is bounded by $\mathcal{O}(n \log n)$. This implies that to process $|E^\ell|$ edges, a processor ℓ uses $\mathcal{O}\left(|E^\ell| + n \log n\right)$ supersteps. Therefore, the amortized per-edge processing time is

$$\mathcal{O}\left(\frac{|E^\ell| + n \log n}{|E^\ell|}\right) = \mathcal{O}\left(\frac{n \log n}{|E^\ell|}\right) = \mathcal{O}\left(\frac{n \log n}{L_{min}}\right) = \mathcal{O}(\log n).$$

◀

Note that the amortized per-edge processing time is computed over the edges of an individual stream, not over the total number of edges across all streams. While both forms of amortization are meaningful for poly-streaming algorithms, our analysis is more practically relevant, as it reflects the cost incurred per edge arrival within a single stream.

▶ **Lemma 13.** *For any constant $\varepsilon > 0$, Algorithm PS-MWM takes $\mathcal{O}(L_{max} + n \log n)$ time.*

Proof. The preprocessing phase (Steps 1–2) takes $\Theta(n/k)$ time.

To process $|E^\ell|$ edges, processor ℓ takes $\mathcal{O}\left(|E^\ell| + n \log n\right)$ supersteps (see the proof of Lemma 12). Since $|E^\ell| \leq L_{max}$ for all $\ell \in [k]$, the time required for Step 3 is $\mathcal{O}(L_{max} + n \log n)$.

At the beginning of Step 5, the total number of edges in the stacks is $\mathcal{U}^{(1)} = \mathcal{O}(n \log n)$. By Lemma 9, iteration t of Process-Stack removes at least one edge from $\mathcal{U}^{(t)}$. Hence, the time required for Step 5 is $\mathcal{O}(n \log n)$.

The claim now follows by summing the time spent across all three phases. ◀

Now, using the characterizations of tight edges, we extend the duality-based analysis of [11] to our algorithm. Let Δ_α^e denote the change in $\sum_{u\in V}\alpha_u$ resulting from processing an edge $e \in E^\ell$ in Step 3 of Process-Edge. If an edge $e \in E^\ell$ is not included in a stack S^ℓ, then $\Delta_\alpha^e = 0$, either because it fails the condition in Step 1 or Step 3 of Process-Edge. It follows that $\sum_{e\in\bigcup_{\ell\in L} E^\ell}\Delta_\alpha^e = \sum_{u\in V}\alpha_u$. For an edge e that is included in some stack S^i, let $\mathcal{P}(e)$ denote the set of edges that share an endpoint with e and are included in some stack S^j no later than e including e itself. The following two results are immediate from Observation 3.2 and Lemma 3.4 of [11].

▶ **Proposition 14.** *Any edge e added to some stack S^ℓ satisfies the inequality*

$$w_e \geq \sum_{e'\in\mathcal{P}(e)} g_{e'} = \frac{1}{2}\left(\sum_{e'\in\mathcal{P}(e)}\Delta_\alpha^{e'}\right).$$

▶ **Proposition 15.** *After all processors complete Step 3 of Algorithm PS-MWM, the variables $\{\alpha_u\}_{u\in V}$, scaled by a factor of $(1+\varepsilon)$, form a feasible solution to the dual LP in Figure 2.*

▶ **Lemma 16.** *Let $\mathcal{M}^*$ be a maximum weight matching in G. The matching $\mathcal{M} := \bigcup_{l\in[k]}\mathcal{M}^\ell$ returned by Algorithm PS-MWM satisfies $w(\mathcal{M}) \geq \frac{1}{2(1+\varepsilon)}w(\mathcal{M}^*)$.*

Proof. We only process tight edges in Process-Stack. By Lemma 10 tight edges are vertex disjoint, and hence their independent processing does not interfere with their inclusion in $\mathcal{M}$.

By Lemma 8, an edge e included in $\mathcal{M}$ must satisfy $\mathcal{F}(e) = \varnothing$. Consider any edge $e' \in \mathcal{P}(e)\backslash\{e\}$. Since $e \in \mathcal{F}(e')$, we have $\mathcal{F}(e') \neq \varnothing$, which means e' is not a tight edge before e is processed. Thus, when e is selected for inclusion in $\mathcal{M}$, none of the edges in $\mathcal{P}(e)\backslash\{e\}$ is tight. Hence, all edges of $\mathcal{P}(e)$ are in the stacks when we are about to process e. Therefore, the total gain contributed by edges of $\mathcal{P}(e)$ can be attributed to the weight of e, and by Proposition 14, we have

$$\begin{aligned} w(\mathcal{M}) = \sum_{e\in\mathcal{M}} w_e &\geq \frac{1}{2}\left(\sum_{e\in\mathcal{M}}\sum_{e'\in\mathcal{P}(e)}\Delta_\alpha^{e'}\right) \geq \frac{1}{2}\left(\sum_{e\in\bigcup_{\ell\in[k]} S^\ell}\Delta_\alpha^e\right) \\ &= \frac{1}{2}\left(\sum_{e\in\bigcup_{\ell\in[k]} E^\ell}\Delta_\alpha^e\right) = \frac{1}{2}\left(\sum_{u\in V}\alpha_u\right). \end{aligned}$$

Let $\{x_e^*\}_{e\in E}$ be an optimal solution to the primal LP in Figure 2. By Proposition 15 and LP duality we have

$$w(\mathcal{M}^*) \leq \sum_{e\in E} w_e x_e^* \leq (1+\varepsilon)\left(\sum_{u\in V}\alpha_u\right) \leq 2(1+\varepsilon)w(\mathcal{M}).$$

◀

Algorithm PS-MWM uses only one pass over the streams. Theorem 1 now follows by combining the results in Lemma 12, Lemma 13, Lemma 16, and the analysis of the *deferrable strategy* sketched in the arXiv version.

PS-MWM-LD(V, l, j, ε)**:**

1. In parallel initialize $lock_u$, and set α_u and $mark_u$ to 0 for all $u \in V$
   /* processor ℓ initializes or sets $\Theta(n/k)$ locks/variables */
2. In parallel initialize $lock_u^j$, and set α_u^j to 0 for all $u \in V$
   /* processor ℓ initializes or sets $\Theta(n/(k/r))$ locks / variables */
3. In parallel initialize $glock^j$ /* one processor initializes for group j */
4. $S^\ell \leftarrow \varnothing$ /* initialize an empty stack */
5. for each edge $e = \{u, v\}$ in ℓth stream do
   a. Process-Edge-LD(e, S^ℓ, ε)
6. wait for all processors to complete execution of Step 4 /* a barrier */
7. $\mathcal{M}^\ell \leftarrow$ Process-Stack(S^ℓ)
8. return $\mathcal{M}^\ell$

Figure 6 A generalization of Algorithm PS-MWM using local dual variables.

4 Algorithms for Non-Uniform Memory Access Costs

In this section, we extend the algorithm from Section 3 to account for the non-uniform memory access (NUMA) costs present in real-world machines.

In a poly-streaming algorithm, each processor may receive an arbitrary subset of the input, making it difficult to maintain memory access locality. Modern shared-memory machines, as illustrated in Figure 1, have non-uniform memory access costs and far fewer memory controllers than processors [12]. As a result, memory systems with such limitations would struggle to handle the high volume of concurrent, random memory access requests generated by poly-streaming algorithms, leading to significant delays.

We now describe a generalization of the algorithm from Section 3 that localizes a significant portion of each processor's memory access to its near memory. This generalization applies to both edge-processing strategies introduced in Section 3.1. We focus on the *non-deferrable strategy*. (The deferrable strategy generalizes in the same way, following the same relationship between the two strategies as in the specialized case.)

The runtime of Process-Edge is dominated by the time to access the dual variables $\{\alpha_u\}_{u \in V}$. By assigning a dedicated stack to each processor, we have substantially localized accesses associated with edges in that stack. However, since a large fraction of edges is typically not included in the stacks, the runtime remains dominated by accesses to dual variables associated with these discarded edges. We therefore describe an algorithm that localizes these accesses to memory near the processor.

To localize accesses to the dual variables $\{\alpha_u\}_{u \in V}$, we observe that these variables increase monotonically during the streaming phase. This observation motivates a design in which a subset of processors maintains local copies of the variables and can discard a substantial number of edges without synchronizing with the global copy. When a processor includes an edge in its stack, it increments the corresponding dual variables in the global copy by the gain of the edge and synchronizes its local copy accordingly. As a result, some local copies may lag behind the global copy, but they can be synchronized when needed.

A general scheme for allocating dual variables is as follows. The set of k processors is partitioned into r groups. For simplicity, we assume that k is a multiple of r, so each group contains exactly k/r processors. For $r > 1$, in addition to a global copy of dual variables,

Process-Edge-LD$(e = \{u, v\}, S^\ell, \varepsilon)$**:**

/* Assumes access to $\{\alpha_u\}_{u\in V}$, $\{lock_u\}_{u\in V}$, $\{\alpha_u^j\}_{u\in V}$, $\{lock_u^j\}_{u\in V}$, and $glock^j$ */

1. if $w_e \leq (1+\varepsilon)(\alpha_u^j + \alpha_v^j)$ then return
2. repeatedly try to acquire $lock_u^j$ and $lock_v^j$ in lexicographic order of u and v as long as $w_e > (1+\varepsilon)(\alpha_u^j + \alpha_v^j)$
3. if $w_e \leq (1+\varepsilon)(\alpha_u^j + \alpha_v^j)$ then release $lock_u^j$ and $lock_v^j$, and return
4. repeatedly try to acquire $glock^j$
5. Process-Edge(e, S^ℓ, ε)
6. $\alpha_u^j \leftarrow \alpha_u$ and $\alpha_v^j \leftarrow \alpha_v$ /* synchronization of local and global dual variables */
7. release $lock_u^j$, $lock_v^j$, $glock^j$ and return

Figure 7 A subroutine used in Algorithm PS-MWM-LD.

we maintain r *local copies* $\{\alpha_u^j\}_{u\in V}$, one for each $j \in [r]$. Group j consists of the processors $\{\ell \in [k] \mid \lfloor \ell/(k/r) \rfloor = j\}$, and uses $\{\alpha_u^j\}_{u\in V}$ as its local copy of the dual variables. Algorithm PS-MWM is the special case $r = 1$, where all processors operate using only the global copy of the dual variables.

Algorithm PS-MWM-LD, along with its subroutine Process-Edge-LD, incorporates local dual variables in addition to global ones. In Step 2, processors in each group $j \in [r]$ collectively initialize their group's local copies of dual variables and locks, followed by initializing a group lock in Step 3. All other steps of the algorithm are identical to those in PS-MWM.

In the subroutine Process-Edge-LD, Step 5 implements the non-deferable strategy. Steps 1–3 and Step 6 enforce the localization of access to dual variables. Steps 2–3 ensure that, at any given time, each global dual variable is accessed by at most one processor per group; we refer to this processor as the *delegate* of the group for that variable. Thus, a processor executing Steps 4–6 serves as a delegate of its group for that time. In Step 6, after completing updates to the global variables, the delegate synchronizes its group's local copy in $\mathcal{O}(1)$ time. As a result, the waiting time on a local variable in Step 2 is bounded by the total time spent by the corresponding delegates, up to constant factors.

The delegates in each group handle vertex-disjoint edges, so concurrent executions of Step 6 would have been safe. However, the lock in Step 4 ensures that at most one delegate per group executes Step 5. Regardless of these design choices, the behavior of the delegates executing Step 5 concurrently mirrors that of processors competing for exclusive access to global dual variables in PS-MWM.

The following lemma highlights the benefit of using Algorithm PS-MWM-LD.

▶ **Lemma 17.** *For any constant $\varepsilon > 0$, in the streaming phase of Algorithm PS-MWM-LD, processors in all r groups collectively access global variables a total of $\mathcal{O}(r \cdot n \log n)$ times.*

In contrast to the result in Lemma 17, the streaming phase of Algorithm PS-MWM accesses global variables $\Omega(m)$ times or up to $\mathcal{O}(m + k \cdot n \log n)$ times.

Algorithm PS-MWM-LD, together with the generalization of the deferrable strategy, leads to the following result (a proof is included in the arXiv version).

▶ **Theorem 18.** *Let k processors be partitioned into r groups, each with its own shared local memory. For any constant $\varepsilon > 0$, there exists a* single-pass *poly-streaming algorithm for the maximum weight matching problem that achieves a $(2+\varepsilon)$-approximation. It admits a*

Table 1 Summary of datasets. Each collection contains eight graphs.

Graph Collection	# of Edges (in billions)
The *SSW graphs*	1.36 – 127.4
The *BA graphs*	4.64 – 550.1
The *ER graphs*	256 – 4096
The *UA-dv graphs*	275.2 – 550.1
The *UA graphs*	8.93 – 1100
The *ER-dv graphs*	32 – 4096

CREW PRAM implementation with $\widetilde{\mathcal{O}}(L_{max} + n)$ *runtime. If* $L_{min} = \Omega(n)$*, the algorithm achieves* $\mathcal{O}(\log n)$ *amortized per-edge processing time using* $\widetilde{\mathcal{O}}(k + r \cdot n)$ *space. For arbitrarily balanced streams, it uses either* $\widetilde{\mathcal{O}}(k + r \cdot n)$ *space and* $\widetilde{\mathcal{O}}(n)$ *per-edge processing time, or* $\widetilde{\mathcal{O}}(k \cdot n)$ *space and* $\mathcal{O}(1)$ *per-edge processing time. The processors collectively access the global memory* $\widetilde{\mathcal{O}}(r \cdot n)$ *times.*

5 Empirical Evaluation

This section summarizes our evaluation results for Algorithm PS-MWM-LD. Detailed datasets, experimental setup, and additional comparisons (including with PS-MWM) are provided in the arXiv version.

5.1 Datasets

Table 1 summarizes our datasets. Each collection consists of eight graphs, with edge counts ranging from one billion to four trillion. To the best of our knowledge, these represent some of the largest graphs for which matchings have been reported in the literature. Exact and approximate offline MWM algorithms (see [22]) would exceed available memory on the larger graphs. The first class (SSW) consists of six of the largest graphs from the SuiteSparse Matrix collection [5] and two from the Web Data Commons [19], which includes the largest publicly available graph dataset. Other classes include synthetic graphs generated from the Barabási–Albert (BA), Uniform Attachment (UA), and Erdős–Rényi (ER) models [1, 7, 21].

5.2 Experimental Setup

We ran all experiments on a community cluster called Negishi [17], where each node has an AMD Milan processor with 128 cores running at 2.2 GHz, 256–1024 GB of memory, and the Rocky Linux 8 operating system version 8.8. The cores are organized in a hierarchy: groups of eight cores constitute a core complex that share an L3 cache. Eight core complexes form a socket, and they share four dual-channel memory controllers; two sockets constitute a Milan node [12]. Memory access within a socket is approximately three times faster than across sockets.

We implemented the algorithms in `C++` and compiled the code using the *gcc* compiler (version 12.2.0) with the `-O3` optimization flag. For shared-memory parallelism, we used the OpenMP library (version 4.5). All experiments used $\varepsilon = 1e - 6$. Reported values are the average over five runs.

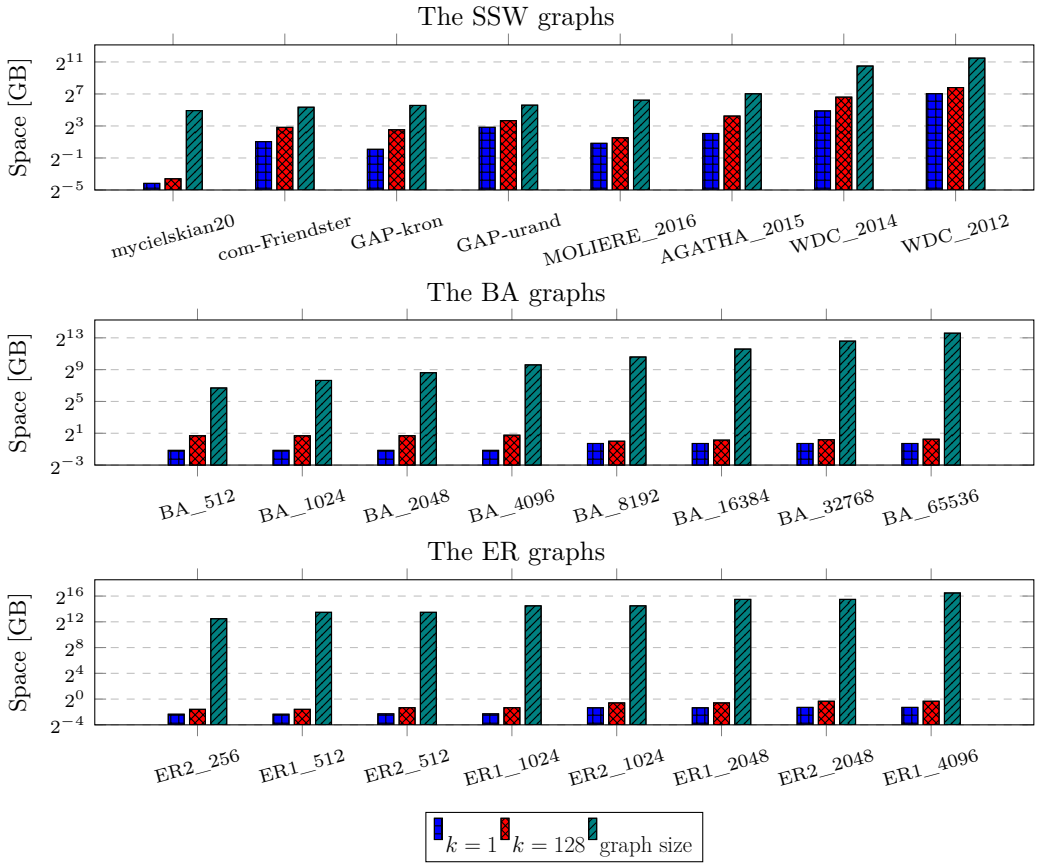

Figure 8 Memory used by the algorithm and the corresponding *graph size* (space needed to store the entire graph in CSR format). Note that the y-axes are in a logarithmic scale.

5.3 Space

Figure 8 summarizes the space usage of our algorithm. For $k = 1$, the algorithm of Paz and Schwartzman [20], we store one copy of the dual variables, stack, and matching. For $k > 1$, our algorithm stores $r + 1$ copies of the dual variables (global and local), stacks, matching, and locks. We choose the values of r based on the number of memory controllers and the number of streams.

The maximum space used by our algorithm is 223 GB, for the web graph WDC_2012. In comparison, storing this graph in compressed sparse row (CSR) format would require over 2800 GB. Storing the largest graph in our datasets (ER1_4096) in CSR would require more than 91,600 GB (89.45 TB), for which our algorithm used less than 0.8 GB.

5.4 Solution Quality

min-OPT percent. In the arXiv version, we describe different ways to get *a posteriori* upper bounds on the weight of a MWM $w(M^*)$, using the values of the dual variables. Let Y_{min} denote the minimum value of these upper bounds. If $\mathcal{M}$ is a matching in the graph returned by any algorithm, then we have $\frac{w(\mathcal{M})}{w(\mathcal{M}^*)} \geq \frac{w(\mathcal{M})}{Y_{min}}$. Hence, $\frac{w(\mathcal{M})}{Y_{min}} \times 100$ gives a lower bound on the percentage of the maximum weight $w(\mathcal{M}^*)$ obtained by $\mathcal{M}$. We use *min-OPT percent* to denote the fraction $\frac{w(\mathcal{M})}{Y_{min}} \times 100$.

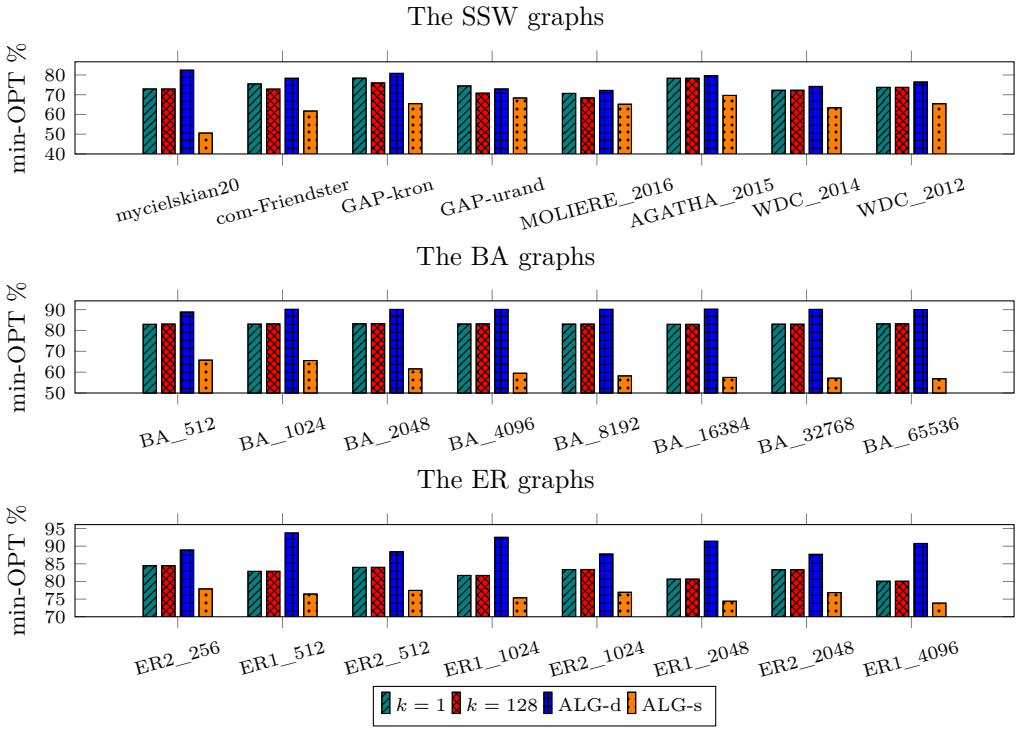

Figure 9 Comparisons of *min-OPT percent* obtained by different algorithms. *ALG-d* denotes the best results from four dual update rules, and *ALG-s* denotes the algorithm of Feigenbaum et al. [8].

Figure 9 shows min-OPT percent obtained by different algorithms. In the arXiv version, we describe four dual update rules as alternatives to the default rule used in Steps 3(a)–(b) of Process-Edge. The values under $k = 1$ and $k = 128$ use the default rule, and the values under *ALG-d* use the best result among the four new dual update rules. For perspective, we include min-OPT percent obtained by the sequential 6-approximate streaming algorithm of Feigenbaum et al. [8], denoted *ALG-s*.

The results under $k = 1$ and $k = 128$ show that, in terms of solution quality, our poly-streaming algorithm is on par with the single-stream algorithm of [20]. The values under *ALG-d* indicate further potential improvements using alternative dual update rules. The comparison with *ALG-s* supports our choice of the algorithm from [20] over other simple algorithms, such as that of [8]. The arXiv version contains comparisons with an offline algorithm and details on the dual update rules.

5.5 Runtime

We report runtime-based speedups, computed as the total time across all three phases of PS-MWM-LD (preprocessing, streaming, and post-processing). Figure 10 shows these speedups. For $k = 128$, we have speedups of $16 - 60$, $37 - 73$, and $68 - 83$ for the SSW graphs, the BA graphs, and the ER graphs, respectively.

Due to the significant memory bottlenecks (discussed in Section 4), we also report speedups w.r.t. *effective iterations* (Definition 3), which are less affected by such bottlenecks. The speedup w.r.t. effective iterations is the ratio of the metric for one stream to that for k streams. Now for $k = 128$, we obtain speedups of $112 - 127$, $121 - 127$, and $124 - 128$ for the SSW graphs, the BA graphs, and the ER graphs, respectively. These results indicate

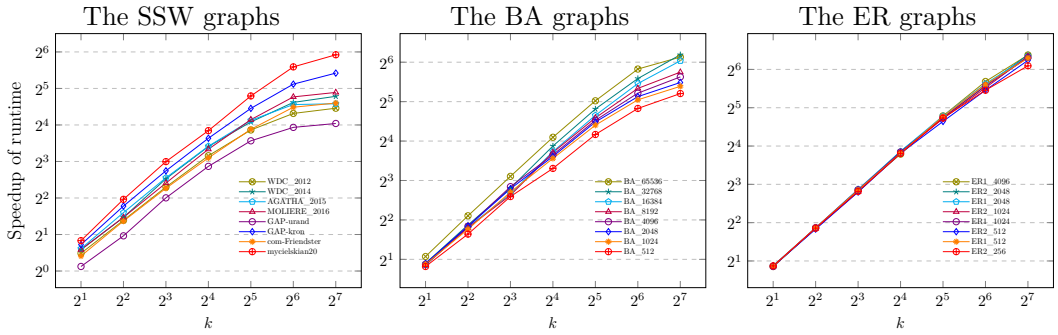

Figure 10 Speedup in runtime vs. k. Note that both axes are in a logarithmic scale.

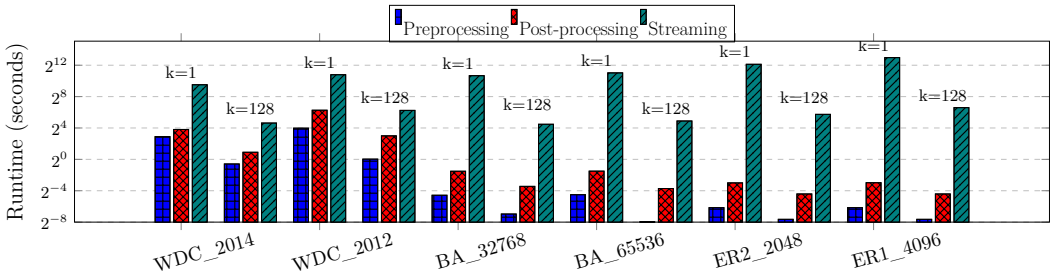

Figure 11 Breakdown of runtime into three phases for $k = 1$ and $k = 128$. Note that the y-axis is in a logarithmic scale.

that shared variable access incurs no noticeable contention among processors. As a result, we expect even better runtime improvements on systems with more memory controllers or better support for remote memory access.

Figure 11 shows the runtimes for different graphs, decomposed into three phases, for $k = 1$ and $k = 128$. The plots report the absolute time savings achieved by processing multiple streams concurrently. For $k = 1$ and $k = 128$, the geometric means of the runtimes for these graphs are over 2350 seconds and under 45 seconds, respectively. For the largest graph (ER1_4096), single-stream processing took over 8000 seconds, whereas poly-stream processing reduced the time to under 100 seconds.

6 Conclusion

While numerous studies have focused on optimizing time (in parallel computing) or space (in streaming algorithms) in isolation, the poly-streaming model offers a *practically relevant paradigm* for exploring how time and space can be jointly optimized. It fills a gap by providing a formal framework for analyzing algorithmic design choices and their associated time-space trade-offs. Our study of matchings illustrates both the benefits of this paradigm and its practical relevance.

The simplicity of our matching algorithm and its generalization reflect our choice to adopt the design of [20]. We believe this principle will inspire the development of other poly-streaming algorithms. To this end, we note that [20] has also motivated simple algorithms for related problems, such as matchings with submodular objectives [16], b-matchings [14], and collections of disjoint matchings [9].

References

1 Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002.

2 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, pages 20–29, 1996.

3 Sepehr Assadi. A simple (1—ε)-approximation semi-streaming algorithm for maximum (weighted) matching. In *2024 Symposium on Simplicity in Algorithms (SOSA)*, pages 337–354. SIAM, 2024.

4 Michael Crouch and Daniel M Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*, pages 96–104. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014.

5 Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

6 Leah Epstein, Asaf Levin, Julián Mestre, and Danny Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM Journal on Discrete Mathematics*, 25(3):1251–1265, 2011.

7 Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–60, 1960.

8 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

9 S M Ferdous, Bhargav Samineni, Alex Pothen, Mahantesh Halappanavar, and Bala Krishnamoorthy. Semi-streaming algorithms for weighted k-disjoint matchings. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, pages 53–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

10 Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 491–500, 2019.

11 Mohsen Ghaffari and David Wajc. Simplified and space-optimal semi-streaming $(2+\varepsilon)$-approximate matching. In *Symposium on Simplicity in Algorithms*, volume 69, 2019.

12 NASA High-End Computing Capability (HECC). AMD Milan Processors, 2024. Accessed: 2025-07-07. URL: `https://www.nas.nasa.gov/hecc/support/kb/amd-milan-processors_688.html`.

13 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External Memory Algorithms*, 50:107–118, 1998.

14 Chien-Chung Huang and François Sellier. Semi-streaming algorithms for submodular function maximization under b-matching, matroid, and matchoid constraints. *Algorithmica*, 86(11):3598–3628, 2024.

15 Michael Kapralov. Space lower bounds for approximating maximum matching in the edge arrival model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1874–1893. SIAM, 2021.

16 Roie Levin and David Wajc. Streaming submodular matching meets the primal-dual method. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1914–1933. SIAM, 2021.

17 Gerry McCartney, Thomas Hacker, and Baijian Yang. Empowering faculty: A campus cyberinfrastructure strategy for research communities. *Educause Review*, 2014.

18 Andrew McGregor. Finding graph matchings in data streams. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 170–181. Springer, 2005.

19 Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1, 2015.

20 Ami Paz and Gregory Schwartzman. A (2+ ε)-approximation for maximum weight matching in the semi-streaming model. *ACM Transactions on Algorithms (TALG)*, 15(2):1–15, 2018.

21 Erol A Peköz, Adrian Röllinn, and Nathan Ross. Total variation error bounds for geometric approximation. *Bernoulli*, 19(2):610–632, 2013.

22 Alex Pothen, S M Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

23 Mariano Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.