A Parallel Batch-Dynamic Maximum Flow Algorithm

Jewon Im and Topher Pankow

Yale University

May 7, 2025

1 Introduction

Although much research has been done on both static and parallel algorithms and methods for solving graph problems (including the Maximum Flow/Minimum Cut problem) in many applications, graphs are constantly subjected to dynamic updates, and with large batch updates, recomputation of updates as new graphs becomes extremely costly. By utilizing parallel computation methods, we examine processing batch-dynamic updates to calculate the maximum flow of a network graph, basing on the Push-Relabel algorithm and recent work in parallel dynamic updates, parallel graph processing, and batch-dynamic graph frameworks.

2 Maximum Flow

The maximum flow problem, conceptually, handles the traversal of resources through a restricted graph. The problem handles a directed graph $G = (V, E)$ with two distinguished nodes, s, t, and a capacity function for the edges ($c(e) > 0, \forall e \in E$). Conceptually, resources move from s to t in the graph along directed edges such that resources are conserved as they move through nodes and such that no edge carries more resources (f_e) than its capacity specifies. To satisfy these conceptual needs precisely, the state of the graph is expressed as a flow: f. A flow defines a load on each edge (f_e) such that the following qualities are maintained:

Conservation of flow:

$$\forall v \in V \setminus \{s, t\} : \sum_{e=(w,v)\in E} f_e = \sum_{e=(v,w)\in E} f_e \tag{1}$$

Respect of Capacity:

$$\forall e \in E : f_e \leq c(e) \tag{2}$$

A maximum flow is then the flow $argmax_f \sum_{e=(s,w)\in E} f_e$. The end of the maximum flow problem is to find such a flow.

3 Exact Maximum Flow

Classic approaches to the maximum flow problem like Ford-Fulkerson and the Push-Relabel algorithm pursue exact solutions. Strong approaches exist to parallelize such algorithms, and some works have examined

dynamic approaches in the exact maximum-flow setting. In an ideal world, we would produce a parallel batch dynamic algorithm for exact maximum flow. However, some problems do arise. To begin, we walk through the underlying approaches and challenges of exact maximum flow.

3.1 Sequential Exact Maximum Flow

Most parallel exact maximum flow algorithms build on Goldberg and Tarjan's "push/relabel" algorithm [5]. Their algorithm pushes flow out from the source in quantities that may exceed the max flow of the network, and then provides mechanisms to allow excess flow to be pushed back to the source. The critical element enabling this versatile push operation is a height label, $h(v) \geq 0$, for all vertices. The network is initialized with the source labeled for the greatest height ($h(s) \leftarrow n$) and other nodes with $h(v) \leftarrow 0$. Additionally, outgoing edges from the source are initialized to their maximum capacity (this configuration is referred to as the pre-flow). Flow only moves from higher nodes to lower nodes, so flow moves out from the source. When a node receives more flow than it sends (an excess – $e(v) > 0$), it becomes active. Goldberg and Tarjan's algorithm iterates over all active nodes until none remain.

At any step, it will send flow to nodes at a height exactly one below the target node (if the edges to those nodes are under capacity), or, if there are none such nodes, the active node will set its own height to be one greater than that of its lowest neighbor capable of accepting more flow. In this way, it will iterate up through the heights of its neighbors to finds an edge that is under capacity and send flow along. Since upstream nodes are initialized higher to begin with, flow will move outward preferentially and send back to the source only when no more flow can move downstream. Consequently, flow propagates forward through the network until it reaches bottlenecks and then excess flow returns to the source, respecting capacity and ultimately ensuring all nodes have symmetric flows.

Each of n node is limited to relabeling $2n - 1$ times at most (allowing flow to pass the length of the longest possible network once in each direction). This allows at most $O(n^2)$ total relabeling operations. And at each level, a given node can affect at most m edges. Consequently, the algorithm obtains a complexity of $O(n^2m)$. $O(n^2m)$ wins out to be the overall complexity. (Goldberg and Tarjan also present more complex methods that improve the sequential time bound even further, but they not relevant to most parallel extensions of the algorithm.)

Goldberg and Tarjan additionally present an parallel application of their algorithm. In lieu of expounding on that here, we cover a different parallel extension of their algorithm in 3.2 .

Critically, Goldberg and Tarjan's algorithm operates on individual edges, with dependencies only to immediate neighbors. This characteristic proves particularly well suited for parallelism, as updates only require locking small regions of the total network, with the majority of other edges and nodes free for other threads to process. Goldberg and Tarjan outline their algorithm in a sequential context but other have parallelized it.

3.2 Parallel Exact Maximum Flow

Anderson and Setubal presented an early parallel enhancement to Goldberg and Tarjan's push-relabel algorithm [1]. Their implementation of the core parallel elements is quite straightforward. When they relabel a vertex they only lock that vertex itself (observe that they do not need to lock any neighboring vertices as in any connected pair of vertices, only one can increase due to the other at a given time). As for the push operation, they only need lock the active node and the node it is pushing to, as the operation is independent of other nodes.

However, one other key element underlies their parallelism. Goldberg and Tarjan's original paper outlined an additional labeling scheme that more intentionally maps the heights of the nodes to produce 'downhill' paths to the sink. These methods improve the immediate efficiency of the algorithm but introduce

overhead to maintain. Anderson and Setubal found a version of these secondary methods useful in practice to increase the efficiency of the algorithm [1]. To implement and maintain the more accurate labeling, they perform BFS in waves through network, labeling nodes with their distance from the source. They used locking to perform the updates themselves and had nodes track the wave on which they were updated to prevent nodes from different waves interacting.

3.3 Classic Dynamic Solution

Kumar and Gupta presented an early algorithm for exact incremental (adding an edge) dynamic updates to a maximum flow graph [6]. Their approach is perhaps the most natural, but it highlights the difficulties of such an algorithm. For every edge added, they track a set of affected vertices. They find this set by searching out from the added edge, both forward and backward in the network using BFS. This search does locate all of the vertices that may be affected by a new edge, as edges without a flow between them cannot directly exchange flow. Then, essentially a full maximum-flow algorithm is run on those affected vertices to push an excess flow through the new edge. The approach works and gains efficiency by not recomputing the whole graph. Additionally, starting from a pre-computed maximum-flow can frequently reduce the complexity of the calculation.

However, in a well connected graph, a search like BFS could ultimately include a substantial subset of the total nodes. Additionally, were we to try batch parallelizing such an algorithm, this problem would be amplified dramatically as each tree would increase the portion of the graph needing to be recomputed. And, depending on implementation, it may not be possible to handle the affected subsets in parallel (though, ideally they could all be processed at once). For larger batch sizes, the approach would regress toward a full graph recomputation.

3.4 Motivation for Approximation Algorithms

Exact maximum-flow solutions are significant, but recent results have found them difficult to maintain dynamically. Additionally, in practical applications, an exact solution may not be strictly necessary. Instead, a asymptotic complexity of a lower order might justify a proportionally approximation of the exact solution. The computational overhead to maintain exact maximum-flow may be unsustainable to begin with, and even when not, more frequent updates to maximize a given network could ultimately give a higher utilization than exact, less frequent updates that allow the network to spend longer periods further from maximum-flow. In this light, much of recent literature on dynamic maximum-flow focuses on approximation algorithms. We examine the extension of such results into a parallel batch-dynamic setting.

4 Approximate Maximum-Flow

4.1 Interior Points Methods

Many modern approaches to maximum-flow, including those below (we are drawing on various papers from Brand et al. here), avoid strict the pitfalls of exact maximum-flow by converting the problem into a convex optimization problem [3] [7]. Instead of directly approaching the maximum flow problem directly, they connect s to t with an infinite capacity link so that s and t possess no special properties. They define a potential function for the circulation that has barrier terms encouraging flow symmetry and capacity respect.

They examine this network for circulations (flow looping through the $s - t$ edge) that minimize the gradient of the potential function (minimize in the sense of negative high magnitude). These circulations can then be added to the state of the network to minimize the potential function, bringing it closer to a maximum flow. This addition operation works since the properties of circulations (capacity for low circulations and

flow symmetry) are respected under linear combination. These minimum circulations are approximated by a series of recursive data structures (principally spanners which approximate a larger flow network with a smaller one) to allow quick generation. These improvements on the potential function are applied iteratively in an interior points optimization. Then with extended analyses, the potential function can be guaranteed to reach an acceptable approximation within an efficient number of steps.

4.2 Incremental Approximate Min-Cost Flow

The recent work by Chen et al. introduces an almost-linear total time algorithm for the incremental thresholded min-cost flow problem, which generalizes the classical minimum-cost flow to dynamic graphs undergoing edge insertions [4]. In this setting, a flow must meet vertex demands while minimizing the total cost of routing, and the algorithm must detect the first moment at which a flow of cost at most F becomes feasible. A $(1+\varepsilon)$-approximate version of the problem is also addressed.

The key technical tool enabling these results is the introduction of a dynamic min-ratio cycle oracle. This oracle identifies negative cycles under a dynamic edge-weighting based on current flow and cost gradients. Building on the ℓ_1-interior point method (IPM) introduced in earlier work, the authors reduce incremental min-cost flow to a sequence of calls to this oracle. The challenge addressed in this work is to develop an oracle that can operate against adaptive adversaries—necessary in dynamic settings.

The algorithm maintains an ℓ_1-oblivious routing that allows extraction of approximate min-ratio cycles, which are used to perform multiplicative-weights updates to both primal and dual variables. As a result, the incremental setting admits a total time of $m^{1+o(1)}$, with deterministic guarantees and no dependence on knowing the total number of future insertions.

This framework immediately yields improvements for several related problems via reductions, such as incremental $(1+\varepsilon)$-approximate max-flow, s–t shortest path (even with negative edge weights), and weighted bipartite matching. Notably, this is achieved with optimal ε^{-1} dependence under the OMv conjecture, improving upon prior bounds of ε^{-2} or worse in similar problems.

4.3 Decremental Approximate Min-Cost Flow

In parallel, Brand et al. extend these techniques to decremental graphs–those undergoing edge deletions, cost increases, and capacity reductions—achieving a similar almost-linear update time guarantee [2].

Unlike the incremental setting, decremental updates threaten the feasibility of any existing flow solution. To address this, the authors perform a duality transformation, reducing the min-cost flow problem to a transshipment problem and then solving its dual. This dual, expressed in vertex potential space, preserves feasibility under deletions and cost increases, making it naturally suited to the decremental setting.

As in the incremental setting, the core approach leverages an ℓ_1-interior point method, but in this case it operates directly on the dual problem. The dynamic updates affect the matrix B and cost vector c, but if the dual constraints remain feasible, the dual solution continues to improve. This monotonicity of the dual objective allows the algorithm to sidestep many difficulties of recomputing flows in the primal space after deletions.

To enable fast updates, the authors develop efficient decremental data structures for approximating min-ratio cuts, which play a role analogous to the min-ratio cycle oracle in the incremental case. These structures rely on dynamically maintained tree cut sparsifiers that efficiently capture the cut structure of the underlying graph.

The resulting algorithm supports decremental thresholded min-cost flow in total time $(m+Q)m^{o(1)}$ for Q updates, under the assumption of bounded integer demands and costs. In addition to min-cost flow, this approach yields new decremental algorithms for approximate single-source reachability, strongly connected components, and s–t distances—several of which had no prior near-linear solutions.

5 Methods

We propose a parallel batch-dynamic framework for maintaining approximate maximum flows in capacitated, directed graphs. Our approach unifies the incremental cycle-based IPM framework developed by Brand and Liu with the decremental cut-maintenance structures described by van den Brand et al. and Chen et al. This unified system supports efficient parallel batch processing of both edge insertions and deletions, and dynamically maintains a near-optimal solution without full recomputation. Our implementation is designed for shared-memory systems using OpenMP and C++17 concurrency.

5.1 Problem Setting and Objectives

Let $G = (V, E)$ be a directed graph with nonnegative capacities $c : E \to \mathbb{R}_{>0}$, a designated source node s, and a sink node t. Our goal is to maintain a $(1 - \epsilon)$-approximate maximum s–t flow value as the graph undergoes dynamic updates in the form of batches of edge insertions and deletions. Each update batch modifies the residual network and potentially alters both the primal flow and the dual weights used in the interior-point formulation.

Unlike fully dynamic recomputation methods which reprocess the entire graph after each update, we seek to repair only the affected parts of the solution using dual-aware oracles and local data structures, thus achieving asymptotically lower work and span.

5.2 Interior-Point Framework for Batch Updates

Our implementation adopts a primal-dual formulation where three main quantities are maintained:

- f_e: The primal flow on edge e.
- ϕ_e: A dual weight associated with cycles (used in the incremental subroutine).
- ψ_e: A dual weight associated with cuts (used in the decremental subroutine).

We define the residual slack as $s_e = c_e - f_e$, and the gradient as $g_e = \pm\tau/s_e$ for a global barrier parameter τ. In each batch, the algorithm executes a single outer iteration of an ℓ_1-style interior point method (IPM). The key step is to compute a direction $c \in \mathbb{R}^m$ that reduces the duality gap, using one of two oracles:

- An incremental cycle oracle (used after insertions) that returns an approximate min-ratio circulation.
- A decremental cut oracle (used after deletions) that returns an approximate min-ratio cut.

Each direction is scaled and applied using a multiplicative weights update on the dual variables and a primal update on the flow. The new flow is then rounded and the slacks are recomputed.

5.3 Incremental Oracle: RoutingDS

To support efficient incremental updates, we construct a dynamic forest structure that tracks cycles introduced by off-tree edges. Initially, a union-find structure identifies tree edges; newly inserted edges that form cycles are tracked separately. We implemented a Heavy-Light Decomposition (HLD) system to traverse these cycles and compute aggregate metrics like path length and gradient. Each cycle query enumerates the path induced by the off-tree edge and computes the ratio ℓ/∇, where ℓ is the total residual slack and ∇ is the total gradient.

To improve efficiency and support future deletions, we later replaced this structure with a Link-Cut Tree (LCT) that supports dynamic tree updates in $\tilde{O}(\log n)$ time and provides path aggregates via splay operations. This change enabled a unified interface for both cycle detection and cut queries.

5.4 Decremental Oracle: TreeCutDS

For deletions, the data structure must dynamically maintain connectivity and support min-ratio cut queries on the surviving forest. We used an Euler Tour Tree (ETT) backed by treaps to store the current spanning tree and allow dynamic cut and link operations. Each edge stores two values: `len` (the slack) and `grad` (the gradient), and nodes maintain subtree minima of the ratio ℓ_e/∇_e.

To support path queries for min-ratio edges, we implemented a minimal Link-Cut Tree system based on splay trees. This allows us to find the bottleneck edge along any root-to-leaf path in $O(\log n)$ time, even as the tree is modified by deletions.

5.5 Batch Parallelism and Edge Update Handling

Edge insertions and deletions are applied in parallel. For insertions, each edge is added with its forward and reverse residual arcs to the adjacency list using atomic updates. The RoutingDS structure is updated by classifying the edge as a tree edge or an off-tree edge using union-find. All insertions are processed concurrently using OpenMP.

For deletions, the code traverses adjacency lists to locate and mark arcs as inactive. Tree edges are removed from TreeCutDS by cutting the corresponding path in the Euler Tour Tree. Insertions and deletions are handled independently using asynchronous futures (via std::async), allowing both phases to proceed in parallel.

Following update application, the algorithm invokes the incremental and decremental repair routines. These functions run in parallel and update the flow and dual weights by computing a direction and applying the interior-point method update rule.

5.6 Theoretical Complexity and Work Analysis

Let $n = |V|$ and $m = |E|$ denote the number of nodes and edges respectively. Let k denote the size of each batch.

- Each edge insertion/deletion affects only $O(\log n)$ paths in the LCT, and each path operation takes $\tilde{O}(\log n)$ time.
- Each oracle call (cycle or cut) takes $\tilde{O}(\log^2 n)$ time to compute a direction.
- Primal-dual updates are parallelizable and take $O(m)$ work, but only $O(\log n)$ span.
- Total work per batch is $\tilde{O}(m + k \log^2 n)$.

Compared to recomputation (Push–Relabel with $O(m\sqrt{n})$), our framework achieves near-optimal speedup for sparse graphs.

6 Results

We lay out a mock implementation of a parallel batch-dynamic approximate maximum flow framework in C++17 using OpenMP for shared-memory parallelism. This mock integrates cycle-based oracles for insertions and cut-based oracles for deletions, each paired with an interior-point method (IPM) style update rule. Table 1 summarizes a detailed pseudocode-style descriptions and a high-level execution pipeline for this proposal.

Component	Description
`BatchDynamicIPM`	Main orchestrating class. Manages the graph, flow vector f, dual weights ϕ, ψ, and barrier parameter τ. Handles batch updates and coordinates cycle/cut repair.
`processBatch(batch)`	Splits batch into insertions and deletions. Launches `parallelInsert` and `parallelDelete` concurrently using `std::async`. Then invokes `repairCycles` and `repairCuts` in parallel.
`repairCycles(S)`	Runs up to 40 IPM-style iterations. Computes slack $s\_e = c\_e - f\_e$ and gradients $g\_e = -\tau/s\_e$. Queries `RoutingDS` for an approximate min-ratio cycle. Updates f_e and ϕ_e multiplicatively using computed direction.
`repairCuts(S)`	Symmetric to `repairCycles`. Uses $g\_e = \tau/s\_e$ and queries `TreeCutDS` for min-ratio cut. Updates f_e and ψ_e. Fall back to push-relabel if no valid direction is found.
`RoutingDS`	Implements cycle oracle using a dynamic spanning forest. Off-tree edges are stored separately. Uses either HLD or LCT to enumerate cycles. Each edge maintains pointers `head`, `tail`. Queries scan all off-tree edges to return one minimizing ℓ/∇.
`TreeCutDS`	Implements decremental min-ratio cut oracle using Link-Cut Trees. Each tree arc stores `len`, `grad` and is linked into the forest using LCT primitives. Supports `deleteEdges` and `queryMinRatioCut`.
`LCT::link(u,v), cut(u,v)`	Maintains dynamic tree connectivity. Each LCT node maintains subtree sums for $\sum \ell\_e$ and $\sum \nabla\_e$. Push/pull logic ensures reversals and updates are lazy and correct.
`LCT::pathAgg(u,v)`	Performs root-to-root splay traversal and returns $\sum \ell\_e, \sum \nabla\_e$ for the path. Used to compute cycle/cut direction.

Table 1: Proposal of core components.

7 Conclusion

We focused this project as a reading-intensive exploration of dynamic and parallel algorithms for maximum flow. We aimed to identify and understand the structural barriers to efficient dynamic update processing in flow networks and unify multiple research directions into a single, practically-oriented implementation framework. Our focus centered around one central question: how would we build a parallel batch-dynamic approximate maximum flow algorithm?

To this end, we reviewed a broad body of work spanning from classical exact algorithms such as Goldberg and Tarjan's push-relabel method, to recent conditional lower bounds under SETH, and ultimately to modern IPM-style approaches. This culminated in a deep dive into two recent papers: the decremental duality-based approach of van den Brand et al. [2] and the incremental cycle oracle framework of Chen et al. [4]. These works demonstrated nearly-linear time algorithms for their respective settings but stopped short of addressing mixed dynamic batches or parallelism.

To bridge this gap, we implemented an IPM-based parallel architecture that supports both insertions and deletions in shared-memory environments. We examined replacing existing heavy structures like spanners with dynamic trees (HLD and later LCTs) to support general updates. We adapted cycle detection and cut recovery to batch-parallel logic and constructed a dual-aware primal-dual update pipeline to implement a hybrid of the two theoretical frameworks.

While the implementation remains incomplete in some technical details (e.g., fully functional decremental tree surgery), the framework is structurally sound and extensible. The core primitives—parallel update application, oracle integration, dual updates—are explored, and the experience offers guidance for batch-dynamic max flow in practice.

Given the project's short time frame, we do not benchmark extensively. However, we expect the total work to scale as $\tilde{O}(m + k \log^2 n)$ with $O(\log n)$ span per update, providing a marked improvement over full recomputation even under existing parallel push-relabel variants. More broadly, this work provides a roadmap for future implementation efforts and opens the door to more advanced data structures such as dynamic low-stretch trees or parallel sparsifiers.

References

[1] Richard Anderson and João C. Setubal. "A Parallel Implementation for the Push-Relabel Algorithm". In: *Journal of Parallel and Distributed Computing* 29 (1995), pp. 17–26.

[2] Jan van den Brand et al. *Almost-Linear Time Algorithms for Decremental Graphs: Min-Cost Flow and More via Duality*. 2024. arXiv: 2407.10830 [cs.DS]. URL: https://arxiv.org/abs/2407.10830.

[3] Jan van den Brand et al. *Incremental Approximate Maximum Flow on Undirected Graphs in Subpolynomial Update Time*. Nov. 6, 2023. DOI: 10.48550/arXiv.2311.03174. arXiv: 2311.03174[cs]. URL: http://arxiv.org/abs/2311.03174 (visited on 05/03/2025).

[4] Li Chen et al. *Almost-Linear Time Algorithms for Incremental Graphs: Cycle Detection, SCCs, s-t Shortest Path, and Minimum-Cost Flow*. 2023. arXiv: 2311.18295 [cs.DS]. URL: https://arxiv.org/abs/2311.18295.

[5] Andrew V. Goldberg and Robert E. Tarjan. "A new approach to the maximum-flow problem". In: *Journal of the ACM* 35.4 (Oct. 1988), pp. 921–940. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/48014.61051. URL: https://dl.acm.org/doi/10.1145/48014.61051 (visited on 05/06/2025).

[6] S Kumar and P Gupta. "An Incremental Algorithm for the Maximum Flow Problem". In: ().

[7] Jan Van Den Brand, Yang P. Liu, and Aaron Sidford. “Dynamic Maxflow via Dynamic Interior Point Methods”. In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*. STOC ’23: 55th Annual ACM Symposium on Theory of Computing. Orlando FL USA: ACM, June 2, 2023, pp. 1215–1228. ISBN: 978-1-4503-9913-5. DOI: 10.1145/3564246.3585135. URL: https://dl.acm.org/doi/10.1145/3564246.3585135 (visited on 05/03/2025).