Strongly Polynomial Parallel Maximum Flow Revisited

Adam Karczmarz
University of Warsaw, Poland

Paweł Pilarski
University of Warsaw, Poland

Abstract

We study the maximum flow problem in directed networks with real capacities in the parallel setting. For a network with n vertices and m arcs, we show that a randomized parallel implementation of a variant of the strongly polynomial max-flow algorithm of Dadush, Orlin, Sidford, and Végh [8] runs in $\widetilde{O}(mn)$ work and $\widetilde{O}(m)$ depth. This improves upon the previously described tradeoffs between work and depth for strongly polynomial parallel maximum flow algorithms: earlier $\widetilde{O}(n^3)$-work algorithms have $\widetilde{O}(n^2)$ depth [15, 29], while the known $\widetilde{O}(m)$-depth approach uses $\widetilde{O}(mn^3)$ work [25].

2012 ACM Subject Classification Theory of computation → Network flows; Theory of computation → Shared memory algorithms

Keywords and phrases maximum flow, parallel algorithm, work-depth tradeoff, strongly polynomial

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.147

Related Version *Full Version*: https://arxiv.org/abs/2608.12171

Funding Supported by the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

Acknowledgements We thank the anonymous ESA reviewers for their helpful comments.

1 Introduction

Maximum flow is one of the most fundamental and well-studied polynomial-time problems on graphs. Given a graph $G = (V, E)$ with n vertices and m arcs with non-negative capacities, a source $s \in V$, and a sink $t \in V$, the problem asks to send as much flow from s to t as possible while maintaining flow conservation at every $v \in V \setminus \{s, t\}$ and not exceeding arc capacities.

Numerous max-flow algorithms have been described since the problem was first shown to be polynomial-time solvable [11, 10]. These early algorithms [11, 10] were *strongly polynomial*, meaning that (1) their running time, measured in the number of elementary arithmetic operations, depends only on the graph size parameters n and m, and (2) they use polynomial space on a Turing machine under the standard binary encoding of capacities. In contrast, the running time of *weakly polynomial* methods may also depend polynomially on the encoding length of the capacities. Typically, weakly polynomial max-flow algorithms are studied for integer capacities, and their efficiency is analyzed as a function of n, m, and $\log U$, where U denotes the maximum capacity in the instance.

Weakly polynomial methods are currently the fastest for integer capacities of subexponential magnitude. The first max-flow algorithm running in $\widetilde{O}((nm)^{1-\delta} \log U)$ time for some $\delta > 0$ appeared in the late 1990s [14]. Recent years brought striking improvements, including $m^{1+o(1)} \log U$-time algorithms [7, 34, 33], as well as even faster methods tailored to denser graphs [36, 5]. Notably, the interior-point based developments [7, 34, 33, 36] also solve the more general min-cost flow problem within similar bounds.

At the same time, there has been no polynomial improvement in strongly polynomial max-flow running time, as a function of n and m alone, since the 1980s, when $\widetilde{O}(nm)$-time algorithms were obtained [12, 30] by accelerating the blocking-flow method [10] with dynamic trees. More recent work on strongly polynomial max-flow [26, 8] focused on matching the $O(nm)$ bound, understanding the barrier behind it, and exploiting additional structure. In particular, [26, 8] give faster strongly polynomial algorithms for dense instances with a limited number of *capacitated* (that is, finite-capacity) arcs. If $m_c = O(m)$ denotes the number of vertices plus capacitated arcs, then one algorithm in [8] runs in $\widetilde{O}(n^{\omega-1}m_c)$ time, where $\omega < 2.372$ is the matrix multiplication exponent [2].

Parallel setting. Our focus is on parallel shared-memory algorithms for max flow. We use the standard *work-depth* viewpoint (e.g., [6, 19, 29]) when measuring the efficiency of a parallel algorithm. The *work* is the total number of operations performed, and the *depth* is the length of the longest chain of sequential dependencies. Since we care about polynomial speedups only, we state all work and depth bounds in $\widetilde{O}(\cdot)$ notation. At that level of granularity, there is no need to distinguish between standard shared-memory models such as EREW and CREW, because they are equivalent up to polylogarithmic overheads [19].

Parallel computation of max flow has also been studied extensively. The known algorithms, however, generally exhibit a trade-off between work and depth: lower depth is obtained at the price of higher work. This applies to both weakly and strongly polynomial algorithms. The recent almost-linear-time max-flow algorithms [7, 34, 33] are highly sequential. On the weakly polynomial side, a recent parallel min-cost flow algorithm of [35] yields improved bounds for max flow on sufficiently dense instances, achieving near-linear work there together with $\widetilde{O}(\sqrt{n}\,\mathrm{poly}(\log U))$ depth. Older randomized parallel results also give very low depth in capacity-dependent regimes, but only with high polynomial work and with running times that still depend explicitly on $\log U$ or on related encoding parameters [23, 27].

Strongly polynomial parallel max-flow received most attention in the 1980s. Blocking-flow based methods [10] generally perform $\Theta(n)$ blocking-flow augmentations in sequence. The $\widetilde{O}(m)$-time blocking-flow algorithm for acyclic networks [30] seems difficult to parallelize in a non-trivial way; however, one can compute a blocking flow in $\widetilde{O}(n^2)$ work and $\widetilde{O}(n)$ depth [29, 37], which yields $\widetilde{O}(n^3)$ work and $\widetilde{O}(n^2)$ depth overall [29]. A comparable work-depth pair is also achieved by the push-relabel framework [15]. A near-linear depth bound can be extracted from Orlin's strongly polynomial min-cost flow framework [25]: the algorithm is presented as a sequence of $\widetilde{O}(m_c)$ shortest-path computations, so plugging in a polylog-depth parallel shortest-path routine yields $\widetilde{O}(m_c)$ depth at the cost of $\widetilde{O}(m_c n^3)$ work. Parallel max-flow was also investigated in [28][1].

There is strong evidence that strongly polynomial $\widetilde{O}(1)$ depth may be out of reach. For binary-encoded capacities, max-flow is known to be P-complete under log-space reductions, and thus also under NC reductions [16]. The relevant reduction to a Circuit Value Problem (see also [17]) uses a sparse network with capacities of order $2^{\Theta(n)}$. Since arithmetic on polynomial-bit integers has polylogarithmic depth (see, e.g., [23]), a strongly polynomial $\widetilde{O}(1)$-depth max-flow algorithm would imply algorithms even for such exponential-capacity networks, proving P = NC. Achieving truly sublinear $O(n^{1-\epsilon})$ depth (where $\epsilon > 0$) for strongly polynomial max-flow would also constitute a breakthrough as no truly sublinear-depth parallel algorithms are known for Circuit Value Problems (cf. [17]).

[1] Peretz and Fischler [28] describe a relatively simple algorithm and argue that it can be implemented in $\widetilde{O}(n^4)$ work and $\widetilde{O}(n)$ depth. However, we were unable to verify the claimed $O(n)$-iteration bound as stated. We discuss this issue, and give a counterexample to the argument from [28], in the full version.

To sum up, some of the known strongly polynomial parallel algorithms achieve a natural near-linear depth barrier. This raises the question whether one can reach near-linear depth without giving up near-state-of-the-art (sequential) strongly polynomial work.

1.1 Our results

We answer this question in the affirmative. Our main result is the following.

▶ **Theorem 1.1.** *There is a Monte Carlo randomized strongly polynomial max-flow algorithm with $\widetilde{O}(mn)$ work and $\widetilde{O}(m_c)$ depth. The returned result is correct with high probability.*

By verifying the output and rerunning on failure, Theorem 1 also yields a Las Vegas algorithm with expected $\widetilde{O}(mn)$ work and expected $\widetilde{O}(m_c)$ depth. Verification checks output flow's feasibility and then tests if t is unreachable from s in the residual network. A naive reachability test requires $\widetilde{O}(m)$ work and $\widetilde{O}(n)$ depth, which is dominated by the stated bounds.

In the strongly polynomial regime, the algorithm behind Theorem 1.1 matches the best known work and the best known depth simultaneously, up to polylogarithmic factors. To prove Theorem 1.1, we describe a parallel implementation of a variant of the recent algorithm of Dadush, Orlin, Sidford, and Végh [8]. Along the way, we obtain an auxiliary result: a parallel batch-incremental data structure for transitive closure.

▶ **Theorem 1.2.** *Let G be a directed graph subject to batches of arc insertions. There is a Monte Carlo randomized data structure that explicitly maintains the reachability matrix of G within $\widetilde{O}(mn)$ total work, where m is the number of arcs in the final graph. Each batch is processed in $\widetilde{O}(n^{1/3})$ depth. The maintained matrix is correct with high probability.*

We believe Theorem 1.2 may be of independent interest. While there is a substantial body of work on batch-dynamic graph data structures, e.g., [1, 3, 24], most of it concerns undirected graphs and very low-depth regimes.

1.2 Roadmap

In Section 2 we set some notation. In Section 3 we give a high-level overview of the algorithm of [8] we build on. Building on that, in Section 4 we describe the challenges in parallelizing the framework of [8] and sketch how these challenges are addressed in our parallel implementation.

The remaining part of this extended abstract presents a more detailed view of our algorithm. Section 5 is devoted to the adjustments we make to the framework of [8]. We deal with parallel bottlenecks of the adjusted algorithm in Section 6. Finally, Section 7 discusses our incremental transitive closure data structure. Due to the space limit, many of the details and proofs are deferred to the attached full version.

2 Preliminaries

We follow the notation of [9].[2] Let $G = (V, E)$ be a directed graph where V is a set of vertices and E is a set of arcs. Let $n = |V|$, $m = |E|$. For an arc $e = (i, j)$ let $\overleftarrow{e} = (j, i)$ be a reverse arc of e. Let u be the capacity vector, so that $u_e \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ represents the capacity of an arc $e \in E$. Let m_c denote the number of arcs in E with finite capacities plus n.

[2] In the following, we often refer to the full version [9] of [8].

We will discuss maximum flow instances on different graphs and capacity vectors. We denote by (G, u) a maximum flow instance where $G = (V, E)$ is the underlying network and u is a capacity vector. The graph G has a distinguished source $s \in V$ and sink $t \in V$.

For a vector $f \in \mathbb{R}^E_{\geq 0}$, let the excess of a vertex $v \in V$ be defined as $\mathrm{ex}_f(v) = \sum_{x \in V} f_{(x,v)} - \sum_{x \in V} f_{(v,x)}$. f is a *feasible flow* or *flow* if for every arc e, it satisfies $0 \leq f_e \leq u_e$ and for every vertex $v \in V \setminus \{s, t\}$ it satisfies $\mathrm{ex}_f(v) = 0$. The *value* of flow f is defined as $\mathrm{val}(f) = -\mathrm{ex}_f(s) = \mathrm{ex}_f(t)$. The value of the optimal flow for instance (G, u) is denoted by $\nu(G, u)$. For a flow vector f, let u^f_e be the *residual capacity* of arc e, defined as $u^f_e = u_e - f_e + f_{\overleftarrow{e}}$.

As in [9], we assume in the input instance that G includes infinite-capacity arcs (v, s) and (t, v) for all $v \in V$ and that $n \geq 4$. While adding these auxiliary arcs does not change the maximum flow value $\nu(G, u)$, eventually obtaining a flow supported on the original arcs exclusively is non-trivial. In the full version we describe how this is achieved in $\widetilde{O}(m)$ work and $\widetilde{O}(n) = \widetilde{O}(m_c)$ depth. For a worse[3] $\widetilde{O}(m)$ depth bound, one can simply convert the flow into an acyclic one using the sequential algorithm [30].

Moreover, we assume $(s, t) \notin E$ and there is no $s \to t$ path in G consisting of infinite-capacity arcs only. Finally, for each $e \in E$, the reverse arc $\overleftarrow{e}$ is also in E. If initially $\overleftarrow{e} \notin E$, we set $u_{\overleftarrow{e}} = 0$. These assumptions are w.l.o.g., as justified in [9].

Acyclic basic flows

An important ingredient of the algorithm presented in [9] is the notion of an *acyclic basic flow*.

▶ **Definition 2.1** (Acyclic flow). *A feasible flow f is acyclic if the set of arcs e such that $f_e > 0$ forms an acyclic graph.*

▶ **Definition 2.2** (Basic flow). *A feasible flow f is basic if one of the following equivalent definitions holds:*

1. *The multiset of* undirected *arcs $\{u, v\}$ obtained from the arcs $e = uv$ satisfying $0 < f_e < u_e$ forms a forest where s and t are in different trees.*
2. *f is an extreme point of the feasible-flow polytope, i.e., there are no distinct feasible flows f_1, f_2 and $\alpha \in (0, 1)$ such that $f = (1 - \alpha) \cdot f_1 + \alpha \cdot f_2$.*

In [9], basic flow is defined using item 2 above only, and it is proven there that item 2 implies item 1. However, the two properties are in fact equivalent (see the full version).

3 Overview of the algorithm of Dadush, Orlin, Sidford, and Végh

In this section we give a high-level sketch of the $\widetilde{O}(nm)$-time algorithm from [9] with a special focus on the ingredients that need to be taken care of in the parallel implementation. An informal skeleton of the framework of [9] outlined in this section can be found in Algorithm 1. There, the parallel bottlenecks of the framework have been marked red.

The algorithm of [9] uses a *scaling* approach, i.e., it operates in iterations that gradually improve the approximation error of the constructed solution. Specifically, it maintains a feasible flow f along with an estimate ε on the upper bound on the flow left to be sent in the residual network (G, u^f). In each iteration the error ε either decreases by a factor at least Γ^{-3}, where $\Gamma := 4n^2$, or vanishes completely. This is achieved as follows.

[3] Recall that we might have $m \gg m_c$.

The arcs of (G, u^f) are divided into categories. First, *abundant* arcs are arcs with so large capacity that they can be treated like infinite-capacity arcs. More precisely, e is called abundant if $u_e^f \geq \Gamma\varepsilon$. If an arc becomes abundant at some point, it stays abundant in subsequent iterations. Note that, by the definition of ε, G does not contain an $s \to t$ path composed of abundant arcs. If both e and $\overleftarrow{e}$ are abundant, e is called *free*.

Abundant and free arcs define a partition of (G, u^f) into *free components*. The free component containing s is a set of vertices S reachable from s using paths composed of abundant arcs only. Symmetrically, the free component of t consists of vertices T that can reach t via abundant arcs. Recall that $S \cap T = \emptyset$. Finally, the free components of $V \setminus (S \cup T)$ are the same as connected components of the subgraph of free arcs induced on $V \setminus (S \cup T)$.

Define $E_{\mathcal{P}} \subseteq E$ to be the arcs that connect distinct free components. The arcs outside $E_{\mathcal{P}}$ and all but $O(n)$ abundant arcs can be effectively ignored, as each free component Y is "abundantly" strongly connected, that is, enough flow can be routed freely within $G[Y]$ via $O(|Y|)$ abundant arcs only. Now, the non-abundant arcs in $E_{\mathcal{P}}$ are partitioned into:

1. *Essential* arcs E_{ess} that satisfy $\Gamma\varepsilon > u_e^f \geq \Gamma^{-5}\varepsilon$.
2. *Small* arcs that satisfy $u_e^f < \Gamma^{-5}\varepsilon$.

Roughly speaking, in a single iteration, the flow f is augmented by a Γ^{-5}-approximate maximum flow in an *auxiliary graph* $\bar{G}$ obtained from the residual network (G, u^f) by:
(1) ignoring arcs $E \setminus E_{\mathcal{P}}$ and the small arcs,
(2) preserving the essential arcs, and
(3) compressing connections between s, t, and the endpoints of E_{ess} via abundant paths.
In [9] it is proven that this way, both the number of iterations and the total number of essential arcs through all iterations are $O(m_c)$.

For correctness and efficiency, the computed approximate maximum flow needs to satisfy additional requirements. One uses an *approximate flow solver* ApproxFlow defined as follows.

▶ **Definition 3.1.** *Approximate flow solver* ApproxFlow(G, u, M) *is a procedure that, for instance* (G, u) *and precision parameter* M*, computes an acyclic basic flow* f *and arc* e^* *such that*

$$\nu(G, u^f) \leq m\, u_{e^*}^f \leq m\nu(G, u^f) \leq \frac{\nu(G, u)}{M}.$$

Above, producing an arc e^* whose residual capacity approximates the error serves as a mechanism to help guarantee low bit complexity of the values computed by the algorithm.

As shown in [9], ApproxFlow can be implemented by solving a certain *exact* max-flow instance with integral capacities bounded by $\text{poly}(n, M)$ (obtained from (G, u) by scaling, rounding, and capping) followed by an $\widetilde{O}(m)$-time strongly polynomial step converting the flow into a basic acyclic one [30, 31]. If one uses the state-of-the-art weakly polynomial algorithm [7] for the former, ApproxFlow runs in $m^{1+o(1)}$ time.

Let us now discuss in more detail how abundant subgraph compression is performed in the auxiliary network $\bar{G}$ to guarantee the algorithm's efficiency. It is useful to see why the compression is necessary at all. In principle, in every iteration one could simply include all the abundant and essential arcs in $\bar{G}$. However, in each of $\Theta(m_c)$ iterations we could possibly have just a single essential arc and $\Theta(m)$ abundant arcs, which would make the total cost of ApproxFlow calls $\Omega(m_c m)$ even if a linear-time implementation of ApproxFlow was used.

In [9], the problem of abundant arc compression is reduced to the following *transitive cover* problem. Given a graph $G = (V, E)$ and subset $S \subseteq V$, a graph $H = (V', E')$ is a transitive cover of S with respect to G if $S \subseteq V' \subseteq V$ and pairwise reachability in G between

Algorithm 1 The high-level skeleton of the max-flow algorithm of Dadush et al. [9]. The main parallel bottlenecks – the iteration count, ApproxFlow, and updating the transitive closure – are highlighted in red.

```
Maintain a feasible flow f, an error bound ε, and an incremental set of abundant arcs;
Maintain the free components of (G, u^f) and the subset E_P;
Set up an incremental transitive closure data structure D for the abundant subgraph;
while ε > 0 do
    Identify the current iteration's essential arcs E_ess ⊆ E_P and their endpoints X_i;
    Compress the reachability via abundant arcs between X_i ∪ {s,t} into H using D;
    Build an auxiliary instance (Ḡ, ū) from essential arcs and H;
    f' ← ApproxFlow(Ḡ, ū, poly(n));
    Augment f by f' (potentially introducing auxiliary arcs to G);
    Decrease ε and identify new abundant arcs and free components;
    Insert the new abundant arcs to D;
end
Reroute flow through auxiliary arcs in f to original arcs of G;
return f;
```

vertices S is preserved in H. With that notion in hand, in iteration i with m_i essential arcs E_{ess} whose endpoint set is X_i, it would be enough if the auxiliary graph $\bar{G}_i$ contained E_{ess}, plus a transitive cover H_i of $X_i \cup \{s,t\}$ with respect to the abundant subgraph G^* of G.

To implement this approach, one needs an efficient way to compute possibly small transitive covers of subsets $X_i \cup \{s,t\}$. Note that the sets X_i are revealed online, and recall that the abundant subgraph changes in time: it is subject to $O(m)$ arc insertions throughout.

Dadush et al. [9] compute the transitive covers as follows. The graph G^* is maintained in an *incremental transitive closure* data structure [18] that supports arc insertions and $O(1)$-time reachability queries, and has $O(nm)$ total update time. To obtain a transitive cover of S, the algorithm returns either the entire graph G^*, or a graph (S, F), where F contains one arc uv for each pair $u, v \in S$ such that there exists a path $u \to v$ in G^* – *whichever is smaller*. Note that in the latter case, the graph has size $O(|S|^2)$ and can be obtained in $O(|S|^2)$ time using reachability queries. Observe that if m^* arcs are added to G^* eventually, and transitive cover queries are issued for sets of size $s_1, \ldots, s_k$, $\sum_{i=1}^{k} s_i = s^*$, then the total size of transitive covers produced is at most $\sum_{i=1}^{k} \min(s_i^2, m^*)$. By summing separately for $s_i < \sqrt{m^*}$ and $s_i \geq \sqrt{m^*}$, the sum can be bounded by $O(s^*\sqrt{m^*})$. In [8], the total size of auxiliary networks $\bar{G}_i$ produced over all iterations is $O(m_c\sqrt{m})$. This allows bounding the total time spent on ApproxFlow calls by $(m_c\sqrt{m})^{1+o(1)}$ which is $O(nm)$ for $m = O(n^{1.99})$.[4]

We stress that the above overview does not describe many important technical details of the algorithm of [9]. However, as we argue in the full version, these details do not pose a significant challenge from the parallel implementation standpoint.

4 High-level overview of the parallel implementation

In this section we discuss the main challenges in parallelizing the algorithm of [8] outlined in Section 3 and sketch how these challenges are addressed in our algorithm.

[4] For dense graphs $m = \Theta(n^2)$, using the dense solver [36] in place of [7] yields $\widetilde{O}(nm)$ time.

4.1 Challenges in parallel implementation

The framework of [8] does not parallelize well out of the box and would have $\widetilde{O}(nm)$ depth if implemented literally. The first obstacle is the number of iterations of the main loop (see Algorithm 1): the framework may perform $\Theta(m_c)$ of them, as each iteration processes a set of essential arcs, and $O(m_c)$ essential arcs arise in total [8]. Since the number of iterations is a lower bound on the depth of any parallel implementation, obtaining $\widetilde{O}(m_c)$ depth without reducing the iteration count would require each iteration to run in $\widetilde{O}(1)$ depth.

There are two main parallel bottlenecks inside an iteration. The first one is ApproxFlow. An implementation of ApproxFlow using the almost-linear weakly polynomial max-flow algorithm of [7] for the rounded instance would have depth linear in the number of arcs, which is too much because the total size of all auxiliary instances in [8] is $O(m_c\sqrt{m})$. While there are lower-depth weakly polynomial approaches [35, 23, 27], they come with higher work. Fortunately, the recent weakly polynomial parallel algorithm of [35] provides a good enough work-depth trade-off to fit in our desired work/depth budgets. There is another problem, though: the known efficient methods for enforcing the additional structural requirements of f' – acyclicity [30] and basicness [31] – are inherently sequential.

The second bottleneck is incremental transitive closure. Updating the transitive closure of a dense graph in $\widetilde{O}(1)$ depth per inserted arc is easy if one is willing to spend, say, $\widetilde{O}(n^2)$ work per update. The difficulty is to achieve the $\widetilde{O}(nm)$ total work bound needed here, or $\widetilde{O}(n)$ amortized work per inserted arc. The algorithm of [18] and other natural approaches detect new reachability relations by extending already-discovered paths one arc at a time. Such an approach can easily use $\Theta(n^2)$ depth when processing $O(n)$ insertions in sequence.

4.2 Adjustments to the framework of [8]

Our first modification is to reduce the number of iterations. Recall that the original main loop may perform $\Theta(m_c)$ iterations, each of which triggers one batch update to the transitive closure structure. Designing a structure with $\widetilde{O}(nm)$ total work and $\widetilde{O}(1)$ depth per batch proved challenging, and our incremental transitive closure data structure (Theorem 1.2) only gives $\widetilde{O}(n^{1/3})$ depth per batch without increasing the total work. To compensate for that, we incorporate an idea already present in Orlin's framework [26]: we ensure that iterations process a larger set of essential arcs at once so that the total number of iterations is reduced to $O(m_c/n^{1/3})$. The price is that the residual capacity range used to define the essential arcs in an iteration has to become wider. We show that this does not affect correctness.

However, the widened range is no longer guaranteed to be polynomially bounded in n as in [8], so it is unclear whether using a weakly polynomial max-flow algorithm such as [7, 35] could still be efficient enough inside the ApproxFlow subroutine (see Definition 3.1). In those iterations we therefore fall back to a low-depth (but high-work) exact strongly polynomial algorithm [25]. To prevent the high $\widetilde{O}(m_c n^3)$ work of Orlin's parallel algorithm [25] from becoming the sole work bottleneck, we ensure that whenever we use the strongly polynomial subroutine [25], the auxiliary network has $O(n^{1/3})$ vertices in the worst case and $O(n^{1/3})$ capacitated arcs on average.

4.3 Making an approximate flow acyclic and basic

The near-linear procedures [31, 30] used in [8] to turn an approximate max-flow obtained via scaling, rounding, and capping in the first step of ApproxFlow into an acyclic one and then into a basic one seem difficult to parallelize. Within the original framework they create an $\widetilde{O}(m_c\sqrt{m})$ depth bottleneck. We therefore avoid them altogether. Instead, we enforce the

required structure already on the rounded instances by means of an isolation argument for flows [13]. By sampling polynomially bounded positive integral arc costs, one can guarantee with high probability that the resulting minimum-cost flow routing a prescribed demand vector is unique. A unique minimum-cost flow is automatically acyclic and basic. Thus, the computed approximate flows satisfy the structural requirements for the scaled, rounded, and capped capacities.

The framework still requires these properties with respect to the original residual capacities. Acyclicity is preserved when we translate the rounded solution back to the original instance, but basicness may be lost. We show that the resulting flow is nevertheless close enough to basic that it can be repaired in depth linear in the number of capacitated arcs of the auxiliary network. The total number of such arcs over the entire algorithm is $O(m_c)$.

4.4 Incremental transitive closure with improved depth per batch insertion

To prove Theorem 1.2, we combine repeated matrix squaring with the shortcut-edges technique of Bernstein [4]. Related combinations of these ideas have previously proved useful in static parallel APSP computation [22] and in sequential approximate incremental APSP data structures [20].

Let $H \subseteq V$ be a random subset of size $\Theta(n^{2/3} \log n)$, called the *hitting set*. A standard fact [32] implies that, with high probability, H contains a vertex of every path in any fixed collection of $\mathrm{poly}(n)$ paths of length $n^{1/3}$. As long as H is chosen independently of the updates, this remains true for paths coming from any intermediate version of the evolving graph.

Our data structure maintains four interlocking components:

(1) Short paths: for every $v \in V$, it keeps a subset S_v of vertices currently reachable from v that includes all vertices within distance $n^{1/3}$ from v.

(2) $H \times H$ reachability: for pairs of hitting-set vertices, it maintains reachability in a dense auxiliary graph $G' = (H, E')$, where $uv \in E'$ iff $v \in S_u$.

(3) $V \times H$ reachability: equivalently, it maintains reachability from H to all of V in the reverse graph by adding shortcut arcs to the underlying incremental graph.

(4) $V \times V$ reachability: finally, it extends the same idea to all source vertices in V.

The first, third, and fourth components are implemented using a collection of parallel BFS runs resumed after each batch insertion and truncated after performing $\widetilde{O}(n^{1/3})$ steps; this guarantees $\widetilde{O}(n^{1/3})$ depth per batch. The dense graph G' on H can be updated in $\widetilde{O}(1)$ depth per batch with $\widetilde{O}(|H|^3) = \widetilde{O}(n^2)$ total work by using a deterministic repeated-squaring-based incremental transitive closure data structure. Altogether, this yields $\widetilde{O}(mn)$ total work and $\widetilde{O}(n^{1/3})$ depth per insertion batch.

A final issue is adaptivity. When plugged into the max-flow algorithm of [8], the randomness used by the data structure remains independent of the updates it receives, at least until the data structure errs, in which case we simply count this as failure of the overall Monte Carlo algorithm. This is because the max-flow algorithm only queries the maintained reachability relation during its main loop, and that relation is uniquely determined by the updates.

5 Adjustments to the algorithm of Dadush, Orlin, Sidford, and Végh

In this section we describe the adjustments we make to the algorithm [9] outlined in Section 3. The full version contains the pseudocode of that algorithm, reproduced verbatim, with our changes marked. We stress that we do not alter the overall framework of [8] in a significant

way and our changes do not influence the correctness or strong polynomiality of the algorithm. That being said, the full version contains a description of the changes to the lemmas and proofs in [9] that need to be made to account for our adjustments.

5.1 Definition changes

Our main change to the framework [9] outlined in Section 3 is a more general definition of an essential arc. This change is crucial to reducing the number of iterations of the main loop (see Algorithm 1).

▶ **Definition 5.1** (cf. [9, Definition 4.7]). *Let f be an ε-optimal flow in (G, u) for $\varepsilon > 0$ and let $k \in \mathbb{R} \cup \{\infty\}$ be defined as*

$$k = \min\{k \in \mathbb{R} : k \geq 3, |\{e \in E_{\mathcal{P}} : \Gamma^{-(k+2)}\varepsilon \leq u_e^f < \Gamma\varepsilon \text{ or } \Gamma^{-(k+2)}\varepsilon \leq u_{\overleftarrow{e}}^f < \Gamma\varepsilon\}| \geq 2n^{1/3}\}.$$

We say that k is an essential arc exponent. *We say that $e \in E$ is an* essential arc *with respect to k if $e \in E_{\mathcal{P}}$ and*

$$\Gamma^{-(k+2)}\varepsilon < u_e^f < \Gamma\varepsilon \quad \text{or} \quad \Gamma^{-(k+2)}\varepsilon < u_{\overleftarrow{e}}^f < \Gamma\varepsilon.$$

Additionally, if there are fewer than $2n^{1/3}$ essential arcs for $k < \infty$, we arbitrarily select some of the arcs that satisfy

$$u_e^f = \Gamma^{-(k+2)}\varepsilon \quad \text{or} \quad u_{\overleftarrow{e}}^f = \Gamma^{-(k+2)}\varepsilon.$$

We consider those arcs and their reversals essential and select enough of them so that the number of essential arcs is exactly $2\lceil n^{1/3}\rceil$.

The formula for k in Definition 5.1 guarantees that for $k < \infty$ there are at least $\Omega(n^{1/3})$ essential arcs in a single iteration. If $k > 3$, then $\Gamma^{-(k+2)}\varepsilon$ is the residual capacity of some arc, and if $k = \infty$, then this threshold is 0. It also allows us to prove there are $\widetilde{O}(m_c/n^{1/3})$ iterations.

Apart from changing the lower end of the interval for the essential arcs, for convenience, the lower bound is now a strict inequality (cf. Section 3). This alteration also influences the definition of small arcs so that they satisfy $u_e^f \leq \Gamma^{-(k+2)}\varepsilon$. The flow missed by ignoring the small arcs can still be bounded analogously as in [9] regardless of the strictness of the inequality. Naturally, we also need to adjust a bound in the definition of so-called valid successors [9, Definition 4.9.(A)] from $\varepsilon' \leq \Gamma^{-3}\varepsilon$ to $\varepsilon' \leq \Gamma^{-k}\varepsilon$.

5.2 Algorithm changes

Widening the essential arc capacity range could significantly increase the depth of ApproxFlow, whose first step is to compute max-flow in a scaled, rounded, and capped instance. This is why for large essential arc capacity ranges we sometimes resort to strongly polynomial max-flow solvers.

▶ **Definition 5.2** (Exact flow). *An* exact flow solver *is a procedure* ExactFlow(G, u) *that computes, for an instance (G, u), a maximum flow f that is also acyclic and basic.*

The algorithm changes in the following way. At the beginning of the iteration we check if the value k from Definition 5.1 equals $k = 3$. If this is the case, we use ApproxFlow to compute the approximate flow. Otherwise, there are less than $2n^{1/3}$ essential arcs, and we use ExactFlow instead of ApproxFlow.

5.3 Analysis

▶ **Lemma 5.3.** *The number of essential arcs in each iteration, except the last one, is at least* $2n^{1/3}$.

Proof. This follows immediately from Definition 5.1. If $k = \infty$ then at the end of the iteration we have $\varepsilon = 0$, meaning it is the last iteration. ◀

In the full version, we show how the analysis in [9] is altered to obtain the following bounds.

▶ **Lemma 5.4.** *The number of initially non-abundant arcs that become abundant is* $O(m_c)$.

▶ **Lemma 5.5** (cf. [9, Lemma 6.10])**.** *The total number of essential arcs throughout all iterations is* $O(m_c)$.

▶ **Corollary 5.6.** *The algorithm of [8] with our adjustments performs* $O(m_c/n^{1/3})$ *iterations.*

Proof. By Lemma 5.3, in every iteration, except the last one, we process at least $2n^{1/3}$ essential arcs, and by Lemma 5.5, there are in total $O(m_c)$ essential arcs throughout all iterations. This bounds the number of iterations by $O(m_c/n^{1/3})$. ◀

In [9], the algorithm's correctness (i.e., that the algorithm computes a maximum flow) is shown in [9, Lemma 6.9]. Its proof as well as the proof of a helper lemma [9, Lemma 6.8] is unchanged. In the full version, we also argue the following.

▶ **Lemma 5.7** (cf. [9, Lemma 6.13])**.** *The algorithm is strongly polynomial. For rational input, all values computed remain polynomially bounded in the input encoding length.*

6 Parallelizing Important Parts

In this section we discuss the parts of the algorithm [9] outlined in Section 3, with changes described in Section 5, that are the most challenging to implement in parallel. In the full version we discuss all the other parts; their parallelization does not require much effort.

First, we use our parallel batch-incremental transitive closure data structure (Theorem 1.2, whose construction is given in Section 7) in place of Italiano's data structure [18] that the algorithm of [8] uses.

6.1 ExactFlow

▶ **Lemma 6.1.** *There exists a Monte Carlo randomized parallel implementation of* ExactFlow *with* $\widetilde{O}(n^3 m_c)$ *work and* $\widetilde{O}(m_c)$ *depth. The algorithm's output is correct with high probability.*

This section is devoted to proving Lemma 6.1. To that end, we require an $\widetilde{O}(m_c)$-depth strongly polynomial min-cost flow algorithm. The following theorem follows from Orlin's strongly-polynomial min-cost circulation algorithm [25] (this is also discussed in the full version).

▶ **Theorem 6.2** ([25])**.** *There exists a parallel algorithm computing a minimum-cost circulation in a graph* $G = (V, E)$ *with* m *arcs with capacities in* $\mathbb{R}_{\geq 0} \cup \{\infty\}$ *and costs in* $\mathbb{R}$*, where* m_c *denotes* n *plus the number of finite-capacity arcs, within* $\tilde{O}(m_c \cdot n^3)$ *work and* $\tilde{O}(m_c)$ *depth.*

Moreover, we rely on the following isolation lemma for flows proved in [13].

▶ **Lemma 6.3** ([13, Theorem 8.1]). *Let $\overline{\mathcal{MCF}}$ be an instance of the min-cost flow problem with underlying graph $G = (V, E)$, demand vector $b \in \mathbb{R}^V$, and capacity vector $u \in \mathbb{R}^E_{\geq 0}$. Let the cost vector $\overline{c}_{e \in E}$ be generated by picking each $\overline{c}_e$ independently and uniformly from $\{1, 2, \cdots, C\}$. Then the probability that $\overline{\mathcal{MCF}}$ does not have a unique solution is at most $\frac{2m}{C}$.*

Lemma 6.3 is proven for $C = 4m$ and finite capacities in [13, Theorem 8.1]. We discuss the minor adjustments that need to be made to that proof in the full version.

We are now ready to prove Lemma 6.1. To implement ExactFlow, i.e., compute an acyclic basic exact maximum flow, we perform two min-cost flow computations using Theorem 6.2.

First, we compute the maximum flow value $\nu(G, u)$ only. Note that one can reduce maximum flow on G to minimum-cost circulation on G with an auxiliary infinite-capacity arc from t to s added. The auxiliary ts arc is given cost -1, whereas original arcs are all assigned cost 0. Note that in such a network, the minimum-cost circulation maximizes s, t-flow via original arcs. Given the flow value $\nu(G, u)$, we find an acyclic basic s, t-flow of that value as follows. We apply Lemma 6.3 with $C = 2m^{\gamma+1}$ to G with $b(s) = -\nu(G, u)$, $b(t) = \nu(G, u)$ and $b(v) = 0$ for $v \in V \setminus \{s, t\}$. Here, $\gamma \geq 1$ is an integer constant such that we want the success probability in Lemma 6.1 to be at least $1 - m^{-\gamma}$. Then, we find a minimum-cost flow f with costs $\overline{c}_e$ picked randomly from $\{1, \ldots, 2m^{\gamma+1}\}$.

▶ **Lemma 6.4.** *The obtained minimum-cost flow f is acyclic and basic with high probability.*

6.2 ApproxFlow

▶ **Lemma 6.5.** *There exists a Monte Carlo randomized algorithm that computes* ApproxFlow *with precision $M = \text{poly}(n)$ in $\widetilde{O}(m + n^{1.5} + m'_c n)$ work and $\widetilde{O}(\sqrt{n} + m'_c)$ depth, where m'_c is the number of finite-capacity arcs. The algorithm's output is correct with high probability.*

This lemma introduces and uses the quantity m'_c instead of m_c, because we have defined $m_c = n + m'_c$ and n might be significantly larger than m'_c. In order to construct the algorithm proving Lemma 6.5, we need the following result of [35]:

▶ **Lemma 6.6** ([35]). *There exists a randomized weakly polynomial algorithm solving* minimum-cost *maximum s, t-flow problem with polynomially bounded integer capacities and costs within $\widetilde{O}(m + n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth. The algorithm's output is correct with high probability.*

Our algorithm proceeds exactly as the implementation [9, Algorithm 8] until the function ConvertBasic is called. The algorithm requires a procedure MaxCap which for (G, u) computes an arc $\bar{e}$ such that $mu_{\bar{e}} \leq \nu(G, u)$, estimating the flow left in the network. This procedure is used twice, to estimate the flow in the input instance and, at the end, to compute an arc e^* estimating leftover flow in the residual network of the returned flow. MaxCap is easily implemented in $\widetilde{O}(m + n^{1.5})$ work and $\widetilde{O}(\sqrt{n})$ depth using Lemma 6.6 (see the full version).

The ApproxFlow implementation in [9] computes a feasible flow f that is approximately maximum by running the weakly polynomial algorithm of [7] on an instance with $\text{poly}(n, M)$-bounded integer capacities obtained by scaling the original capacities by a value $\delta = u_{\bar{e}}/(m^2 M)$, rounding them down, and capping them at $m^3 M$. After the integer flow is computed, it is scaled back to use the original (unbounded) range of real capacities.

In order to turn an approximate feasible flow into an acyclic basic flow, the algorithm in [9] uses an $\widetilde{O}(m)$-time procedure ConvertBasic based on [31, 30], which is difficult to parallelize. To circumvent that problem, we first guarantee that f is an acyclic basic integer flow similarly to Section 6.1. That is, instead of computing any max-flow, we assign random

poly(m)-bounded costs to all the arcs (using Lemma 6.3) and find a (unique) minimum-cost maximum flow. Recall from Section 6.1 that the obtained minimum-cost flow is acyclic and basic.

After the integer flow f is scaled back as in [9], it will remain acyclic but might unfortunately no longer be basic. To see this, consider the characterization of a basic flow from item 1 of Definition 2.2. After scaling back, some of the "full" arcs that satisfied $f_e = u'_e$ with respect to *scaled, rounded, capped, and scaled back* capacities $u'_e = \delta \cdot \min\left(\lfloor u_e/\delta \rfloor, m^3 M\right)$, might no longer satisfy $f_e = u_e$. Instead, $f_e < u_e$ may hold. Consequently, the set of *non-basic* undirected arcs with respect to u such that $0 < f_e < u_e$ may not necessarily constitute a forest, even though the non-basic arcs with respect to u' satisfying $0 < f_e < u'_e$ did form a forest.

To deal with that, in the full version we describe a slight change to the integer instance that guarantees the returned flow has at most m'_c non-basic arcs with respect to u that are basic with respect to u'. Our algorithm to convert the flow to a basic one is very similar to the algorithm from [31] but exploits that m'_c bound. This algorithm starts with an empty forest and *adds* arcs one by one. If after adding a new arc we still have a forest we do nothing. However, if it creates a cycle, one can send some flow along the cycle in either direction so that at least one cycle arc satisfies $f_e \in \{0, u_e\}$. This guarantees that at the end of each iteration the set of arcs that satisfy $0 < f_e < u_e$ forms a forest again. To guarantee that the flow is basic, one also needs to check whether there is a path from s to t in the obtained forest. Note that a forest can have at most one such path. If a path from s to t is in the forest, one can pass as much flow as possible from s to t so that at least one of the arcs on this path will no longer satisfy $0 < f_e < u_e$, disconnecting s and t. Observe that the above process cannot introduce new flow cycles, as no flow value turns from 0 to a positive value. Hence, the obtained flow remains acyclic. In the full version, we prove the following.

▶ **Lemma 6.7.** *Adding an arc to the forest of non-basic arcs requires $\widetilde{O}(n)$ work and $\widetilde{O}(1)$ depth.*

Due to the above lemma, we can start from a forest of non-basic arcs with respect to u' and add the $O(m'_c)$ missing non-basic arcs with respect to u one by one. This takes $\widetilde{O}(nm'_c)$ work and $\widetilde{O}(m'_c)$ depth.

6.3 Work and depth analysis

▶ **Theorem 6.8.** *The described parallel max-flow algorithm has $\tilde{O}(nm)$ work and $\tilde{O}(m_c)$ depth.*

Proof. As argued in the full version, the algorithm of [8] with our adjustments has three bottlenecks from the parallel efficiency standpoint: the ApproxFlow calls, ExactFlow calls, and maintaining the incremental transitive closure data structure.

Recall from Corollary 5.6 that the algorithm performs $O(m_c/n^{1/3})$ iterations and thus $O(m_c/n^{1/3})$ batch insertions are issued to the incremental transitive closure data structure. Hence, by Theorem 1.2, that component requires $\widetilde{O}(nm)$ work and $\widetilde{O}(m_c)$ total depth.

Let m_i be the number of essential arcs in iteration i and let $\hat{n}_i, \hat{m}_i$ be the numbers of vertices and arcs in the auxiliary graph in the iteration i. By Lemma 5.5, $\sum_i m_i = O(m_c)$. Recall from Section 3 how an auxiliary network $\bar{G}$ is constructed. We either have $\hat{n}_i = O(m_i)$ and $\hat{m}_i = O(m_i^2)$ if $m_i \leq \sqrt{m}$ or $\hat{n}_i = O(n)$ and $\hat{m}_i = O(m)$ if $m_i > \sqrt{m}$. In both cases, the graph $\bar{G}$ has m_i finite-capacity arcs.

By Lemma 6.5, in $O(m_c/\sqrt{m})$ iterations with $m_i > \sqrt{m}$, the total work of ApproxFlow is

$$\sum_i \widetilde{O}(\hat{m}_i + \hat{n}_i^{1.5} + m_i n) \leq \widetilde{O}(m + n^{1.5}) \cdot \widetilde{O}(m_c/\sqrt{m}) + \widetilde{O}(m_c n) \leq \widetilde{O}(nm).$$

The total depth of these iterations is

$$\sum_i \widetilde{O}(\sqrt{\hat{n}_i} + m_i) \leq \widetilde{O}(\sqrt{n}) \cdot \widetilde{O}(m_c/\sqrt{m}) + \widetilde{O}(m_c) = \widetilde{O}(m_c).$$

For iterations where $m_i \leq \sqrt{m}$, the total work is

$$\sum_i \widetilde{O}(\hat{m}_i + \hat{n}_i^{1.5} + m_i n) = \sum_i \widetilde{O}(m_i^2 + m_i^{1.5}) + \widetilde{O}(m_c n) \leq \widetilde{O}(m_c\sqrt{m} + m_c n) \leq \widetilde{O}(nm),$$

whereas the total depth is $\sum_i \widetilde{O}(\sqrt{\hat{n}_i} + m_i) \leq \sum_i \widetilde{O}(m_i + m_i) = \widetilde{O}(m_c)$.

ExactFlow is used only in iterations where $m_i < 2n^{1/3} < \sqrt{m}$. As in such iterations there are $m_i = \Omega(\hat{n}_i)$ finite-capacity arcs in the auxiliary network, by Lemma 6.1, the total work is

$$\sum_i \widetilde{O}(\hat{n}_i^3 m_i) \leq \sum_i \widetilde{O}((n^{1/3})^3 m_i) \leq \widetilde{O}(nm_c),$$

whereas the total depth of these iterations is $\sum_i \widetilde{O}(m_i) = \widetilde{O}(m_c)$. ◀

7 Parallel Incremental Transitive Closure

In this section we prove Theorem 1.2.

7.1 Data structure for dense graphs

Let us start by noting that a very simple data structure (resembling the incremental approximate APSP data structure of [21]; see the full version) is nevertheless very efficient in the parallel setting for dense directed graphs. We use it as a building block later on.

▶ **Lemma 7.1.** *Let $G = (V, E)$ be a directed graph subject to batches of (distinct) arc insertions. There exists a data structure that maintains a reachability matrix of G within total work $\tilde{O}(n^3)$ such that each batch of insertions is processed in $\tilde{O}(1)$ depth.*

7.2 Data structure for sparse graphs

We define multiple data structures as building blocks for the final data structure.

▶ **Lemma 7.2.** *Let $G = (V, E)$ be a directed graph subject to batches of arc insertions and let $v \in V$. There exists a data structure that maintains a subset $A \subseteq V$ such that:*
(1) *A contains only vertices reachable from v in G,*
(2) *all vertices v' such that the distance between v and v' is at most $n^{1/3}$ are contained in A.*
The total work of the data structure is $\widetilde{O}(m)$. Each batch insertion is processed in $\widetilde{O}(n^{1/3})$ depth.

In the full version, we prove that Lemma 7.2 can be implemented using a parallel version of BFS, truncated to perform $n^{1/3}$ steps after every batch insertion. Using n copies of this data structure for all vertices v of the graph, we immediately get the following data structure:

▶ **Lemma 7.3.** *Let $G = (V, E)$ be a directed graph subject to batches of arc insertions. There exists a data structure that maintains a binary matrix $M \in \{0,1\}^{V \times V}$ such that:*
(1) *$M_{u,v} = 1$ only if there exists a path in G from u to v,*
(2) *if there exists a $u \to v$ path in G of length at most $n^{1/3}$ then $M_{u,v} = 1$.*
The data structure runs in $\widetilde{O}(nm)$ total work. Each batch insertion is processed within $\widetilde{O}(n^{1/3})$ depth.

For the next data structures we will need the following folklore lemma (see, e.g., [32]).

▶ **Lemma 7.4.** *Let S be a set of size n and $\mathcal{S}$ be a family of subsets of S such that $|\mathcal{S}| = \text{poly}(n)$ and $|X| \geq k$ for $X \in \mathcal{S}$. Let $H \subseteq S$ be a set of size $\Theta((n/k)\log n)$ sampled uniformly at random. Then with high probability, for all $X \in \mathcal{S}$, $X \cap H \neq \emptyset$. We call H a* hitting set.

Define a collection of paths $\mathcal{P}$ as follows. Suppose v becomes reachable from u for the first time after the j-th batch insertion, and let G_j be the graph G at that time. Fix some arbitrary simple $u \to v$ path $P_{u,v}$ in G_j. Then let $\mathcal{P}$ include all the $n^{1/3}$-arc subpaths of $P_{u,v}$, if there are any. This way, $\mathcal{P}$ contains at most $n^3 = \text{poly}(n)$ subpaths. Hence, a hitting set H of size $\Theta(n^{2/3}\log n)$ obtained as described in Lemma 7.4 for $S = V$, $k = n^{1/3}$, and $\mathcal{S} = \mathcal{P}$, contains a vertex of each subpath from $\mathcal{P}$ with high probability.

▶ **Lemma 7.5.** *There exists a data structure maintaining (w.h.p.) the reachability matrix among vertices H in G with $\widetilde{O}(nm)$ total work. The depth per batch insertion is $\widetilde{O}(n^{1/3})$.*

Lemma 7.5 is obtained by showing that the transitive closure of a complete graph on H associated with the $H \times H$ submatrix of the matrix maintained by Lemma 7.3 encodes reachability between H with high probability. Hence, that submatrix can be fed into the data structure of Lemma 7.1. For details, see the full version.

The next step is to lift the maintained $H \times H$ reachability to $V \to H$ reachability. This is achieved by considering, for each $h \in H$, an extended reverse graph $\overleftarrow{G}_h$ with *shortcuts* [4] (h, h') added for each $h' \in H$ such that h' can reach h in G (obtained from the previous step). Clearly, in the graph $\overleftarrow{G}_h$, vertex h can reach the same vertices as in $\overleftarrow{G}$. However, one can prove that the added shortcuts reduce the maximum distance from h to $n^{1/3}$.

Note that $\overleftarrow{G}_h$ is also an incremental graph subject to batches of updates to both G and the transitive closure of $H \times H$, maintained in the previous step. Hence, by maintaining the data structure of Lemma 7.2 for each $\overleftarrow{G}_h$ with source h, one obtains the following.

▶ **Lemma 7.6.** *There exists a data structure maintaining the reachability relation for all pairs $V \times H$ in G within $\widetilde{O}(nm)$ total work, and $\widetilde{O}(n^{1/3})$ depth per batch insertion.*

Finally, we can analogously lift the $V \times H$ reachability to the full $V \times V$ reachability by using, for all $v \in V$, graphs G_v with shortcuts (v, h) added for all $h \in H$ that v can reach in G (obtained from the previous step). Again, these incremental graphs constitute the input to a collection of another n data structures of Lemma 7.2. Theorem 1.2 follows.

References

1 Umut A. Acar, Daniel Anderson, Guy E. Blelloch, and Laxman Dhulipala. Parallel batch-dynamic graph connectivity. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2019, Phoenix, AZ, USA, June 22-24, 2019*, pages 381–392. ACM, 2019. doi:10.1145/3323165.3323196.

2 Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025, New Orleans, LA, USA, January 12-15, 2025*, pages 2005–2039. SIAM, 2025. doi:10.1137/1.9781611978322.63.

3 Daniel Anderson, Guy E. Blelloch, and Kanat Tangwongsan. Work-efficient batch-incremental minimum spanning trees with applications to the sliding-window model. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 51–61. ACM, 2020. doi:10.1145/3350755.3400241.

4 Aaron Bernstein. Maintaining shortest paths under deletions in weighted directed graphs. *SIAM J. Comput.*, 45(2):548–574, 2016. doi:10.1137/130938670.

5 Aaron Bernstein, Joakim Blikstad, Jason Li, Thatchaphol Saranurak, and Ta-Wei Tu. Combinatorial maximum flow via weighted push-relabel on shortcut graphs. In *66th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2025, Sydney, Australia, December 14-17, 2025*, pages 464–485. IEEE, 2025. doi:10.1109/FOCS63196.2025.00026.

6 Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996. doi:10.1145/227234.227246.

7 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *J. ACM*, 72(3):19:1–19:103, 2025. doi:10.1145/3728631.

8 Daniel Dadush, James B. Orlin, Aaron Sidford, and László A. Végh. From incremental transitive cover to strongly polynomial maximum flow. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2026, Vancouver, BC, Canada, January 11-14, 2026*, pages 150–163. SIAM, 2026. doi:10.1137/1.9781611978971.7.

9 Daniel Dadush, James B. Orlin, Aaron Sidford, and László A. Végh. From incremental transitive cover to strongly polynomial maximum flow, 2025. doi:10.48550/arXiv.2510.20368.

10 Efim A Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

11 Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.

12 Zvi Galil and Amnon Naamad. An o(evlog^{2}v) algorithm for the maximal flow problem. *J. Comput. Syst. Sci.*, 21(2):203–217, 1980. doi:10.1016/0022-0000(80)90035-5.

13 David Gamarnik, Devavrat Shah, and Yehua Wei. Belief propagation for min-cost network flow: Convergence and correctness. *Oper. Res.*, 60(2):410–428, 2012. doi:10.1287/OPRE.1110.1025.

14 Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998. doi:10.1145/290179.290181.

15 Andrew V. Goldberg and Robert Endre Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.

16 Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. The maximum flow problem is log space complete for P. *Theor. Comput. Sci.*, 21:105–111, 1982. doi:10.1016/0304-3975(82)90092-5.

17 Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, June 1995. doi:10.1093/oso/9780195085914.001.0001.

18 Giuseppe F. Italiano. Amortized efficiency of a path retrieval data structure. *Theor. Comput. Sci.*, 48(3):273–281, 1986. doi:10.1016/0304-3975(86)90098-8.

19 J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.

20 Adam Karczmarz and Jakub Lacki. Reliable hubs for partially-dynamic all-pairs shortest paths in directed graphs. In *27th Annual European Symposium on Algorithms, ESA 2019, Munich/Garching, Germany, September 9-11, 2019*, LIPIcs, pages 65:1–65:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.65.

21 Adam Karczmarz and Jakub Lacki. Simple label-correcting algorithms for partially dynamic approximate shortest paths in directed graphs. In *3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020*, pages 106–120. SIAM, 2020. doi:10.1137/1.9781611976014.15.

22 Adam Karczmarz and Piotr Sankowski. A deterministic parallel APSP algorithm and its applications. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 255–272. SIAM, 2021. doi:10.1137/1.9781611976465.17.

23 Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. *Comb.*, 6(1):35–48, 1986. doi:10.1007/BF02579407.

24 Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. Parallel batch-dynamic algorithms for k-core decomposition and related graph problems. In *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11 - 14, 2022*, pages 191–204. ACM, 2022. doi:10.1145/3490148.3538569.

25 James B. Orlin. A faster strongly polynomial minimum cost flow algorithm. *Oper. Res.*, 41(2):338–350, 1993. doi:10.1287/OPRE.41.2.338.

26 James B Orlin. Max flows in o (nm) time, or better. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 765–774, 2013. doi:10.1145/2488608.2488705.

27 James B. Orlin and Clifford Stein. Parallel algorithms for the assignment and minimum-cost flow problems. *Oper. Res. Lett.*, 14(4):181–186, 1993. doi:10.1016/0167-6377(93)90068-R.

28 Yossi Peretz and Yigal Fischler. A fast parallel max-flow algorithm. *J. Parallel Distributed Comput.*, 169:226–241, 2022. doi:10.1016/J.JPDC.2022.07.003.

29 Yossi Shiloach and Uzi Vishkin. An o(n^2 log n) parallel MAX-FLOW algorithm. *J. Algorithms*, 3(2):128–146, 1982. doi:10.1016/0196-6774(82)90013-X.

30 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

31 Robert E. Tarjan. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Math. Oper. Res.*, 16(2):272–291, 1991. doi:10.1287/MOOR.16.2.272.

32 Jeffrey D. Ullman and Mihalis Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991. doi:10.1137/0220006.

33 Jan van den Brand, Li Chen, Rasmus Kyng, Yang P. Liu, Simon Meierhans, Maximilian Probst Gutenberg, and Sushant Sachdeva. Almost-linear time algorithms for decremental graphs: Min-cost flow and more via duality. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 2010–2032. IEEE, 2024. doi:10.1109/FOCS61266.2024.00120.

34 Jan van den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P. Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 503–514. IEEE, 2023. doi:10.1109/FOCS57990.2023.00037.

35 Jan van den Brand, Hossein Gholizadeh, Yonggang Jiang, and Tijn de Vos. Parallel minimum cost flow in near-linear work and square root depth for dense instances. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2025, Portland, OR, USA, 28 July 2025 - 1 August 2025*, pages 1–16. ACM, 2025. doi:10.1145/3694906.3743356.

36 Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and $_1$-regression in nearly linear time for dense instances. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 859–869. ACM, 2021. doi:10.1145/3406325.3451108.

37 Uzi Vishkin. A parallel blocking flow algorithm for acyclic networks. *J. Algorithms*, 13(3):489–501, 1992. doi:10.1016/0196-6774(92)90051-D.