Sublinear-Time Parallel Algorithms for Matching and Related Problems *

Andrew V. Goldberg†
Department of Computer Science
Stanford University
Stanford, CA 94305

Serge A. Plotkin‡
Department of Computer Science
Stanford University
Stanford, CA 94305

Pravin M. Vaidya§
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*A preliminary version of this paper was presented at the 29th IEEE Symposium on Foundations of Computer Science, White Planes, NY, 1988.

†Research partially supported by NSF Presidential Young Investigator Grant CCR-8858097, IBM Faculty Development Award, and ONR Contract N00014–88–K–0166. Part of the work was done while the author was at AT&T Bell Laboratories.

‡Research supported by NSF Research Initiation Award CCR-900-8226, by U.S. Army Research Office Grant DAAL-03-91-G-0102, by ONR Contract N00014–88–K–0166, and by a grant from Mitsubishi Corporation. Part of the work was done while the author was at M.I.T., Laboratory for Computer Science, Cambridge, MA 02139, supported by DARPA Contract N00014-87-K-825 and by ONR Contract N00014-86-K-0593, and while the author was at AT&T Bell Laboratories.

§Current address: Computer Science Department, University of Illinois, Urbana, IL 61801.

Abstract

This paper presents the first sublinear-time deterministic parallel algorithms for bipartite matching and several related problems, including maximal node-disjoint paths, depth-first search, and flows in zero-one networks. Our results are based on a better understanding of the combinatorial structure of the above problems, which leads to new algorithmic techniques. In particular, we show how to use maximal matching to extend, in parallel, a current set of node-disjoint paths and how to take advantage of the parallelism that arises when a large number of nodes are "active" during an execution of a push-relabel network flow algorithm.

We also show how to apply our techniques to design parallel algorithms for the weighted versions of the above problems. In particular, we present sublinear-time deterministic parallel algorithms for finding a minimum-weight bipartite matching and for finding a minimum-cost flow in a network with zero-one capacities, if the weights are polynomially bounded integers.

1 Introduction

Bipartite matching and related problems have been studied extensively in the contexts of both sequential (*e.g.* [9, 8, 19, 36, 33]) and parallel (*e.g.* [1, 21, 24]) computation. Though recent research produced RNC algorithms for these problems, *i.e.*, randomized parallel algorithms that run in expected polylogarithmic time on a polynomial number of processors, no sublinear-time deterministic parallel algorithms were known. This paper describes several techniques that allow us to construct such algorithms for bipartite matching, flows in zero-one capacity networks, depth-first search, and the problem of finding a maximal set of node-disjoint paths.

Our algorithms for bipartite matching and for zero-one flows generalize to weighted versions of these problems. These generalizations involve the technique of scaling, so the resulting algorithms run in sublinear time if the weights are polynomially bounded. Although Karp, Upfal, and Wigderson [21] have given an RNC algorithm for the assignment problem with unary weights, until now no sublinear-time deterministic algorithms have been discovered.

The results presented in this paper are based on a more complete understanding of the combinatorial structure of the above problems, which leads to new algorithmic techniques. In particular, we show how to use maximal matching to extend, in parallel, a current set of node-disjoint paths. We also show how to take advantage of the parallelism that arises when a large number of nodes are "active" during an execution of Goldberg's parallel push-relabel network flow algorithm [14, 15, 17].

All algorithms of this paper construct paths of a certain type. While the number of paths that need to be constructed is large, the paths are constructed in parallel. When only a small number of paths remain to be constructed, the path are developed one by one using parallel matrix multiplication techniques.

Our model of parallel computation is a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) [11]. Given a directed graph with n nodes and m arcs, $BFS(n, m)$ denotes the maximum of $n + m$ and the number of processors required to find a breadth-first search tree in $O(\log^2 n)$ time; $SSP(n, m)$ denotes the maximum of $n + m$ and the number of processors required to find a single-source shortest-path tree (with nonnegative weights) in $O(\log^2 n)$ time. It is known

that $SSP(n,m) \leq n^3$, and that $BFS(n,m)$ is at most the number of processors required to multiply two $n \times n$ matrices in $O(\log n)$ time, which is $O(n^{2.5})$ [27].

In this paper we address the following problems. (Complexity of the algorithms is given in terms of the number of nodes n, number of edges m, and the largest absolute value of the weights (or costs) C.)

Maximal node-disjoint paths We are given a graph $G = (V, E)$ with a set of sources $S \subset V$ and a set of sinks $T \subset V$, such that $S \bigcap T = \emptyset$. A set Π of node-disjoint paths is said to be *maximal* if each path in Π starts at a distinct source and terminates at a distinct sink, and there is no path from a source to a sink in the graph induced by the nodes in $V - \Pi$. The maximal node-disjoint paths problem is to find a maximal set of node-disjoint paths from S to T. We give an algorithm for the maximal node-disjoint paths problem which runs in $O(\sqrt{n}\log^3 n)$ time, both on directed and on undirected graphs. On undirected graphs our algorithm uses $O(n+m)$ processors, and on directed graphs it uses $BFS(n,m)$ processors. Our algorithm also solves a slight generalization of the maximal node-disjoint paths problem, which is useful for constructing a depth-first search tree in an undirected graph. The fastest previously known technique for solving this problem was to extend the paths sequentially, leading to an $O(n)$-time $O(n+m)$-processors algorithm.

Depth-first search in undirected graphs Given an undirected graph $G = (V, E)$ and a distinguished node, construct a depth-first search tree of the graph rooted at this node. A tree T is a *depth-first search tree* if and only if for all nontree edges (u, v), nodes u and v lie on the same path starting at the root of the tree. The previous parallel algorithms of Anderson [2] and Aggarwal-Anderson [1] for the problem use randomization. Using our techniques for finding node-disjoint paths, we convert the Aggarwal-Anderson randomized depth-first search algorithm into a deterministic algorithm that runs is $O(\sqrt{n}\log^5 n)$ time using $O(n+m)$ processors.

Bipartite matching Given an undirected bipartite graph $G = (S, T, E)$, where $S \bigcup T$ is the set of nodes ($S \bigcap T = \emptyset$) and $E \subset (S \times T)$ is the set of edges, find a maximum cardinality matching in G. We present an algorithm for the bipartite matching problem which runs in $O(n^{2/3}\log^3 n)$ time using $BFS(n,m)$ processors. Although Karp, Upfal, and Wigderson have shown that the problem is in RNC [21], the previous fastest deterministic algorithm, due to Schieber and Moran [28], runs in $\Theta(n)$ time in the worst case. With the parameter values set to optimize the sequential running time, our algorithm runs in $O(\sqrt{n}m)$ time, same as the fastest know sequential bipartite matching algorithm [19, 36].

Assignment problem (Also known as the weighted bipartite matching problem.) Given a weighted undirected bipartite graph $G = (S, T, E)$, find a minimum weight perfect matching.

We present an algorithm for the assignment problem that uses $SSP(n, m)$ processors and runs in $O(n^{2/3} \log^3 n(\log nC))$ time if the edge weights are integers in the range $[-C, C]$. Under the assumption that edge weights are given in unary, this problem is known to be in RNC [21, 24]. Our algorithm is sublinear under this assumption. The previous fastest deterministic algorithm, due to Gabow and Tarjan [12], runs in $\Theta(n \log^3 n \log(nC))$ time in the worst case. With the parameter values set to optimize the sequential running time, our algorithm runs in $O(\sqrt{n}m \log(nC))$ time, same as the fastest know scaling algorithm for the assignment problem [12].

Flows in zero-one networks We study flows in networks with unit capacity arcs (zero-one flow). We consider two versions of the problem, the maximum flow problem and the minimum-cost flow problem. The bipartite matching problem is a special case of the maximum flow problem, and the assignment problem is a special case of the minimum cost flow problem. Our algorithm for finding a maximum flow in a network with unit capacity arcs runs in $O(m^{2/3} \log n)$ time using $BFS(n, m)$ processors. We also show that if the network has no multiple arcs, the algorithm can be modified to run in $O((nm)^{2/5} \log n)$ time. Our algorithm for the minimum-cost flow in a network with unit capacity arcs runs in $O(m^{2/3} \log^2 n \log(nC))$ time using $SSP(n, m)$ processors.

The paper is organized as follows. Section 2 presents definitions and notation used throughout the paper. In Section 3 we describe a parallel algorithm for the maximal node-disjoint paths problem and show how to apply it to depth-first search in undirected graphs. In Section 4 we give parallel algorithms for maximum matching and zero-one flow problems; in Section 5 we extend the results to the weighted versions of these problems.

2 Definitions and Notation

This section presents the notation and definitions used throughout the paper. We assume that the reader is familiar with the standard definitions of maximum flow, minimum cost flow, and maximum matching problems (see, for example, [22, 35] for more details).

Given a graph $G = (V, E)$, let n denote the number of nodes in the graph and let m denote the number of edges (or arcs, if the graph is directed). We assume that G is symmetric, *i.e.*, $(v, w) \in E$ iff $(w, v) \in E$. Throughout the paper we shall deal only with simple paths, and a path will always mean a simple path. The length of a path is defined as follows. If there is a length associated with each edge, then the length of the path is the sum of the lengths of the edges on the path. Otherwise, the length of the path is the number of edges on the path, *i.e.*, each edge is assumed to have length 1.

A *matching* is a set of edges such that each node in the graph is incident to at most one edge in the matching. A *perfect* matching is a matching such that each node in the graph is incident

to exactly one edge in the matching, and a *maximal* matching is a matching such that there is no edge between any two unmatched nodes. The weight of a matching is the sum of the weights of the edges in the matching.

A *pseudoflow* is a function on arcs of the network that obeys capacity constraints and antisymmetry constraints $f(v,w) = -f(w,v)\ \forall (v,w) \in E$. Given a pseudoflow f and a node v, we define the *excess* at v, $e_f(v)$, to be the difference between the incoming and the outgoing flows. A *flow* is a pseudoflow that obeys conservation constraints, *i.e.,* excesses at all nodes except the sink and the source are zero.

The *residual capacity* of an arc (v,w) with respect to a pseudoflow f is defined to be the capacity of (v,w) minus $f(v,w)$, and is denoted by $u_f(v,w)$. Given a pseudoflow f, we denote the corresponding residual graph by $G_f = (V, E_f)$, where E_f is the set of arcs with positive residual capacity.

3 Maximal Node-Disjoint Paths

This section presents an efficient parallel algorithm that finds a maximal set of node-disjoint paths from a set of sources to a set of sinks in a directed or an undirected graph. We describe the variant of the algorithm that works for undirected graphs. The extension to the directed case is straightforward.

A natural approach to solving this problem is to find paths one by one. The problem with this approach is that there can be a large ($\Omega(n)$) number of paths, which leads to a running time that is at least linear. Another approach is to maintain a current set of paths, extending as many paths as possible at each iteration. This approach has two problems. First, it takes time that is proportional to the length of the longest path, and therefore it is slow if the paths are long. Second, it may not be possible to extend many paths at each iteration because of the interaction among the paths. By combining these approaches, however, we can achieve a good running time.

The algorithm *Maximal-Paths* solves a slight generalization of the node-disjoint paths problem: Given a set $\mathcal{P}_{in}$ of node-disjoint paths connecting sources to intermediate nodes, find a set $\mathcal{P}_{out}$ of node-disjoint paths from the sources to the sinks, such that for any node that is on a path in $\mathcal{P}_{in}$ but not on any path in $\mathcal{P}_{out}$, every path from this node to a sink intersects a path in $\mathcal{P}_{out}$. The node-disjoint paths problem corresponds to the case when each one of the input paths is a single source node. The generalization of the problem is required for the depth-first search algorithm which is described at the end of this section.

Figure 1 describes the *Maximal-Paths* algorithm. The algorithm maintains two sets of node-disjoint paths: *active* paths and *completed* paths, denoted by $\mathcal{P}_a$ and $\mathcal{P}_c$, respectively. An active path starts at a source and ends at some intermediate node which is not a sink; a completed path connects a source to a sink. The initial set of active paths $\mathcal{P}_a$ is the set of the input paths $\mathcal{P}_{in}$. The nodes are divided into *idle, active*, and *dead*, denoted by V_I, V_a, and V_d, respectively. A node is *active* if it belongs to a path, *dead* if it was active during the algorithm but currently does not

```
procedure Maximal-Paths(V, E, P_a);
    P_a   – the set of active paths;
    V_I   – the set of idle nodes;
    V_a   – the set of active nodes;
    V_d   – the set of dead nodes;
    T     – the set of sink nodes;

{The first stage}

    V_a ← nodes on paths in P_a;
    V_I ← V − V_a;
    V_d ← ∅;
    while |P_a| ≥ √n do begin
        H ← set of end-points of paths in P_a;
        H' ← {v' : v' ∈ V_I, ∃v ∈ H s.t. (v, v') ∈ E};
        M ← maximal matching on (H × H') ∩ E;
        for all (v, v') ∈ M do begin
            extend the path corresponding to v with v';
            V_I ← V_I − v';
            V_a ← V_a + v';
            if v' ∈ T, remove this path from the set of active paths P_a;
        end;
        for all v ∈ H not matched by an edge in M do begin
            remove v from its path;
            if no nodes left on this path, remove it from P_a;
            V_a ← V_a − v;
            V_d ← V_d + v;
        end;
    end;

{The second stage - number of active paths is below √n}

    for all P ∈ P_a do begin
        E' ← ((V_I × V_I) ∪ (V_a × V_I)) ∩ E;
        v_r ← the node closest to the end of P from which an idle sink is reachable via edges in E';
        remove nodes that follow v_r from P and add them to V_d;
        extend P to a sink via edges in E', add used nodes to V_a;
        remove P from P_a;
    end;
end.
```

Figure 1: The *Maximal-Paths* procedure

belong to any path, and *idle* otherwise. Intuitively, a node becomes *dead* if the current set of active paths can be extended to a maximal set of node-disjoint paths without using this node. Initially, V_d is empty and V_I is the set of nodes not on any input path.

The algorithm consists of two stages. The first stage proceeds in iterations, where at each iteration the algorithm extends some of the active paths by *idle* nodes, changing the status of these nodes to *active*. The algorithm "clips" the other active paths, *i.e.*, removes end-point nodes from these paths, and changes the status of the removed nodes to *dead*. Let H be the set of nodes that are the end-points of the active paths, and H' be the set of *idle* nodes that are neighbors of nodes in H. First, the algorithm finds a maximal matching in the bipartite graph induced by the set of edges $(H \times H') \cap E$, where E is the set of edges in the input graph. If a node $v \in H$ is matched to $v' \in H'$, then the path associated with v is extended, and v' becomes the new end-point of the path, changing its status to *active*. If v' is one of the sinks, this path changes its status to *completed*. If a node $v \in H$ is not matched, the path associated with v is "clipped": the node previous to v on this path becomes the new end-point of the path, and the status of v changes to *dead*. This stage continues as long as the number of active paths is at least $\sqrt{n}$.

During the second stage, the algorithm extends the active paths to sinks one by one. To extend a path $P = (v_1, v_2, \ldots, v_k)$, the algorithm first computes connected components in the graph induced by edges in $(V_I \times V_I) \cup (V_a \times V_I)$. Let v_r be the node on P, such that there exists a path P' from v_r to some sink t via *idle* nodes, and for all $r < i \leq k$, no *idle* sink is reachable from v_i by a path that consists of *idle* nodes only. Then the algorithm clips P, changes the status of the nodes $\{v_i : r < i \leq k\}$ to *dead*, changes the status of nodes on P' to *active*, and extends the path $(v_1, v_2, \ldots, v_r)$ by attaching it to P'. If such a node is not found, *i.e.* it is impossible to extend P using idle nodes, all nodes on P are marked *dead*.

By construction, the algorithm terminates with a set of paths where each path connects a source to a sink, and each source or sink belongs to at most one path. Moreover, a source is marked *dead* if and only if it does not belong to any path and a sink is marked *idle* if and only if it does not belong to any path. Thus, the following lemma is sufficient to show correctness of the algorithm.

Lemma 3.1 At any moment during an execution of the algorithm, there is no path from a *dead* node to an *idle* sink such that all the nodes on this path are either *dead* or *idle*.

Proof: Consider an iteration of the first stage in which a node v becomes *dead*. By construction, v becomes *dead* only if it was not matched during computation of the maximal matching. This means that at the end of this iteration v does not have any *idle* neighbors. On the other hand, if a node changed its status to *dead* during the second stage, then, by construction, there is no path consisting of *idle* nodes only from this node to an *idle* sink. Furthermore, a node cannot change its status to *idle* from any other status, and hence each path from a *dead* node to an *idle* sink must pass through an *active* node. ∎

We analyze the running times of the first and the second stages separately. The following lemma bounds the number of iterations in the first stage.

Lemma 3.2 There are at most $O(\sqrt{n})$ iterations in the first stage.

Proof: The main idea of the proof is that nodes can change status only "in one direction", and that at each iteration a large number of nodes change status. Define a potential function

$$\Phi = |V_A| + 2|V_I|.$$

An extension of a path by one node changes the status of this node from *idle* to *active* and reduces Φ by 1. On the other hand, when a path is clipped, its old end-point changes the status from *active* to *dead*, again reducing Φ by 1. At each iteration of the first stage there are at least $\sqrt{n}$ active paths. At the end of an iteration each one of these paths is either extended or clipped, which causes a total reduction of at least $\sqrt{n}$ in Φ. The claim follows, because $0 \leq \Phi \leq 3n$. ∎

Each iteration of the first stage can be implemented in $O(\log^3 n)$ time with $O(m)$ processors, using the maximal matching algorithm of Israeli and Shiloach [20]. Each iteration of the second stage of the algorithm is essentially a connectivity computation, which can be computed in $O(\log n)$ time and $O(m)$ processors for the undirected case [29], and $O(\log^2 n)$ time and $BFS(n, m)$ processors for the directed case [27]. This leads to the following theorem.

Theorem 3.3

1. On undirected graphs, the *Maximal-Paths* algorithm runs in $O(\sqrt{n}\log^3 n)$ time using $O(n+m)$ processors.
2. On directed graphs, the *Maximal-Paths* algorithm runs in $O(\sqrt{n}\log^3 n)$ time using $BFS(n, m)$ processors.

Observe that maximal sets of node-disjoint paths correspond to blocking flows in matching networks (described in detail in the next section). Thus, by using the *Maximal-Paths* procedure to find a blocking flow at each iteration of Dinitz's maximum flow algorithm [8, 9], we can compute a maximum bipartite matching in sublinear time. In the subsequent sections we will show more efficient algorithms for bipartite matching and related problems; these algorithms do not use the *Maximal-Paths* algorithm.

Depth-First Search Another application of the *Maximal-Paths* algorithm is for constructing a deterministic sublinear-time algorithm for finding a depth-first search tree in an undirected graph. The problem of finding such a tree has been studied before [13, 31], and recently Aggarwal and Anderson have found a *randomized* NC algorithm for it [1]. However, no deterministic sublinear-time parallel algorithm for the problem was known previously.

Although the Aggarwal-Anderson algorithm is randomized, the randomization is used only in order to compute a *maximum* set of node-disjoint paths with the minimum weight [1]. Aggarwal

and Anderson reduce this problem to the assignment problem. A careful examination of their proofs shows that instead of a *maximum* set of paths, it is sufficient to be able to find a *maximal* set of paths. More precisely, it is sufficient to have an algorithm that solves exactly the generalization of the node-disjoint paths problem that is solved by the *Maximal-Paths* algorithm. Therefore, we have the following theorem.

Theorem 3.4 A depth-first search tree in an undirected graph can be found in $O(\sqrt{n}\log^5 n)$ time using $O(n+m)$ processors.

The proof of the theorem involves a straightforward combination of the Aggarwal-Anderson algorithm [1] and the *Maximal-Paths* algorithm.

4 Bipartite Matching and Zero-One Flows

In this section we describe sublinear-time parallel algorithms for the bipartite matching problem and for the zero-one network flow problem (both with and without multiple arcs). This section consists of three parts. In the first part we review Goldberg-Tarjan's generic maximum flow algorithm. In the second part we present our maximum matching algorithm, and in the third part we present our algorithm for finding a maximum flow in networks with unit capacities.

4.1 Goldberg-Tarjan Maximum Flow Algorithm

In this section we review the Goldberg-Tarjan push-relabel framework for solving the maximum flow problem [14, 16, 17]. We present only those ideas which are relevant for finding flows in unit-capacity networks.

Before describing the generic Goldberg-Tarjan algorithm, we need to introduce a few terms. For more detailed definitions, see [17]. A (valid) *distance labeling* is an integer-valued function d on nodes that satisfies $d(v) \leq d(w)+1$ for every residual arc (v,w). Given a pseudoflow f and a distance labeling d, we define

$$E(f,d) = \{(v,w) \in E_f | d(v) = d(w)+1\}$$

to be the the set of *admissible* arcs; the *admissible* graph $G(f,d) = (V, E(f,d))$ is the graph induced by the arcs in $E(f,d)$.

The algorithm is based on *Push* and *Relabel* operations described in Figure 2. *Push* moves one unit of excess through an admissible arc; *Relabel* changes the distance labeling d of a node to create an outgoing admissible arc from this node while maintaining the validity of the distance labeling.

The generic Goldberg-Tarjan maximum flow algorithm is shown in Figure 3. First we set all distance labels to 0, except the label at the source, which is set to n. Then we saturate all arcs

Push(v, w)**.**
Applicability: $e_f(v) > 0$, $u_f(v, w) > 0$ **and** $d(v) = d(w) + 1$**.**
Action: **Send one unit of flow from** v **to** w **as follows:**
$f(v, w) \leftarrow f(v, w) + 1$; $f(w, v) \leftarrow f(w, v) - 1$;
$e_f(v) \leftarrow e_f(v) - 1$; $e_f(w) \leftarrow e_f(w) + 1$.

Relabel(v)**.**
Applicability: **Any** v**.**
Action: $d(v) \leftarrow \min\{d(w) + 1 | (v, w) \in E_f\}$.

Figure 2: Push and Relabel operations for unit capacity networks.

procedure ***Generic-Max-Flow***(V, E, s, t)**;**

[first stage – initialization]
for all $v \in V - \{s\}$ **do** $d(v) \leftarrow 0$;
$d(s) \leftarrow n$;
for all $(v, w) \in E$ **do** $f(v, w) \leftarrow 0$;
for all $v \in V$ **do** $e_f(v) \leftarrow 0$;
for all $v \in V$ **such that** $(s, v) \in E$ **do begin**
$f(s, v) \leftarrow 1$;
$e_f(v) \leftarrow e_f(v) + 1$;
end;

[second stage]
while there exists a node with an excess do
apply *Push* or *Relabel* in arbitrary order.

return the resulting flow f**;**
end.

Figure 3: The generic Goldberg-Tarjan maximum flow algorithm

that emanate from the source. Note that at this point the distance labeling becomes valid. Now we apply *Push* and *Relabel* operations in any order until no more excesses are left.

Goldberg and Tarjan [17] showed that the *Generic-Max-Flow* algorithm terminates with a valid maximum flow. Observe that for the case of zero-one flows, an arc can participate in at most one *Push* operation before one of its ends is relabeled. Since labels can not grow beyond n, this leads to an $O(nm)$ bound on the number of *Push* operations, and an $O(n^2)$ number of *Relabel* operations. Goldberg [14, 15] gave a parallel implementation of the generic algorithm, and showed that it runs in $O(n^2 \log n)$ time.

4.2 Bipartite Matching Algorithm

To solve a bipartite matching problem, we transform it into a zero-one network flow problem in a standard way (see, for example, [22]). Given a bipartite graph with a node set $S \bigcup T$, we direct edges of the graph from nodes in S to nodes in T. We add a source s and arcs (s, v) for all $v \in S$, and a sink t and arcs (w, t) for all $w \in T$; we also add the reverse arcs to make the network symmetric. We define all reverse arc capacities to be zero, and all other arc capacities to be one. The resulting maximum flow problem is equivalent to the original bipartite matching problem. The network which can be obtained by the above transformation is called a *matching network*. We denote vertices and edges of the matching network by V and E, respectively.

Two possible approaches to the design of parallel algorithms for the problem of finding maximum flows in matching networks suggest themselves. One approach is to use the Ford-Fulkerson augmenting path algorithm [10] with a parallel breadth-first search subroutine. Another approach is to use a parallel implementation of the generic Goldberg-Tarjan maximum flow algorithm. Both approaches lead to superlinear-time algorithms, but for different reasons. The bottleneck of the first approach is a potentially large number of augmenting paths; the bottleneck of the second approach is a potentially large number of node relabelings.

Our algorithm works in two stages, using the generic algorithm approach in the first stage and the augmenting path approach in the second stage. A proper balancing of the two stages, which is achieved by adjusting the *activity parameter* and the *distance parameter*, leads to a sublinear running time. In the context of sequential algorithms for the problem, similar balancing, which involves a single parameter that is similar to our distance parameter, was introduced in [9, 36] to obtain $O(\sqrt{n}m)$ time bounds.

Intuitively, the key idea of our bipartite matching algorithm is to keep at most a single unit of excess at any node and to use maximal matching to decide where to push the excesses at each parallel step. Essentially, we replace the "while" loop of the generic Goldberg-Tarjan maximum flow algorithm (see Figure 3) with the *Match-and-Push* procedure (see Figure 5), which, as we will prove below, causes many relabelings to happen at each parallel step.

Figure 4 describes the algorithm *B-Match* which finds a maximum bipartite matching. The algorithm consists of two stages. The first stage is executed as long as the number of *active* nodes is large, where a node is considered active if it has an excess and its distance label d is below k, the *distance parameter*. By "large" we mean that the number of active nodes is above l, the *activity parameter*. (Intuitively, the distance parameter prevents the algorithm from looking for long augmenting paths.) When the number of active nodes is small (below l), excesses from these nodes are returned to the source, and the second stage begins. In this stage, the algorithm finds augmenting paths from the source to the sink one by one. We will prove that the second stage works fast because the residual flow is small.

The first stage of the algorithm starts by initializing the flow to zero, setting distance labels of nodes in $S \bigcup T \bigcup \{t\}$ to zero, and setting the distance label of the source to n. (Throughout the

```
procedure B-Match(S, T, E);

    [initialization]
    transform the input problem into network flow form;

    [first stage]
    for all v ∈ S ⋃ T do d(v) ← 0;
    d(t) ← 0; d(s) ← n;
    for all (v, w) ∈ E do f(v, w) ← 0;
    for all v ∈ S do f(s, v) ← 1;
    for all w ∈ T ⋃ {s, t} do e_f(w) ← 0;
    for all v ∈ S do e_f(v) ← 1;
    while the number of active nodes is at least l do Match-and-Push;
    return all excesses to the source;


    [second stage]
    while there is an augmenting path from s to t do
        find an augmenting path and augment;

    return the matching corresponding to the current (maximum) flow;
end.
```

Figure 4: High-level description of the bipartite matching algorithm. For the algorithm described in this paper, we take $l = n^{2/3}$.

algorithm, distance labels of the source and the sink never change: $d(s) = n, d(t) = 0$.) Then, all arcs going out of the source are saturated. At this point, all nodes of S have excess 1. After the initialization is complete, the *Match-and-Push* procedure is executed until the number of active nodes becomes less than l. Recall that in the context of this algorithm, we have defined an active node to be a node with excess and with label d below the distance parameter k. (As we shall see, the best running time is achieved for distance parameter $k = \lfloor n^{1/3} \rfloor$ and activity parameter $l = \lfloor \sqrt{nk} \rfloor$.) Finally, at the end of the first stage, the flow excesses are returned to the source. Namely, the excess flow is pushed from nodes $v \in S$ such that $e_f(v) = 1$ to s along (v, s).

The *Match-and-Push* procedure, shown in Figure 5, is the key to the first stage of the algorithm. The following lemma states the properties of this procedure that are essential for the analysis of the algorithm.

Lemma 4.1
The *Match-and-Push* procedure maintains the following invariants:

1. The current pseudoflow f is integral.
2. Indegree of a node $v \in S$ in the residual graph G_f is $1 - e_f(v)$.
3. For every node $v \in S \bigcup T$, $e_f(v) \in \{0, 1\}$.
4. On entry to and on exit from *Match-and-Push*, all nodes in T have zero excess.

Proof: Integrality of f follows by induction on the number of the *Push* operations. The second invariant follows from the properties of matching networks.

Invariant 3 holds after initialization by the structure of a matching network. Suppose that the invariant holds before an execution of *Match-and-Push*. Step 1 assures that it holds after Step 2. The relabeling in Steps 3, 5, and 6 cannot affect this invariant. Because of the second invariant, no flow can be pushed to an active node and at most one unit of flow can be pushed to an inactive node at Step 4. Thus Step 4 preserves the invariant.

Invariant 4 holds because after Step 2, every node in T has excess of either zero or one, and because of the relabeling done at Step 3 every node with excess of one has an outgoing admissible arc that can be used to push the excess from the node in Step 4. After Step 4 all nodes in T have zero excesses. The remaining steps do not change the pseudoflow, and therefore Invariant 4 holds at the end of *Match-and-Push*. ∎

From Invariants 3 and 4 in Lemma 4.1 it follows that all matched arcs in Step 2 of *Match-and-Push* are directed from S to T. Lemma 4.1 also implies that the last step of the first stage, namely returning flow from the nodes with excess to the source, is easy. More precisely, Invariants 3 and 4 imply that nodes in T have no excesses, and each node in S has at most one unit of excess. Furthermore, since $k \leq n$, no flow is pushed from a node $v \in S$ to s during the execution of the "while" loop of the first stage of the algorithm, and therefore for all $v \in S$, the residual capacities of arcs (v, s) are equal to one. Thus, at the end of the first stage, the excesses can be pushed from nodes $v \in S$ such that $e_f(v) = 1$ to s along (v, s). Note that these pushes are nonstandard, *i.e.*, they do not preserve the validity of d. This is not a problem, however, because we do not use the distance labels after the last execution of *Match-and-Push*.

We start the analysis of the algorithm by bounding the running time of the *Match-and-Push* procedure.

Lemma 4.2 Procedure *Match-and-Push* runs in $O(\log^3 n)$ time on a CRCW PRAM with $n + m$ processors.

Proof: Steps 2-6 can be implemented so that each step takes $O(\log n)$ time on a CRCW PRAM

Step 1. Find a maximal matching in the subgraph of the admissible graph induced by nodes in T and active nodes in S.
Step 2. For every matched arc (v, w), push excess flow from v to w.
Step 3. Relabel nodes in T.
Step 4. Push flow from active nodes in T along admissible arcs.
Step 5. Relabel nodes in T.
Step 6. Relabel nodes in S.

Figure 5: The *Match-and-Push* procedure

with $n + m$ processors [15, 30]. The bottleneck is Step 1, which takes $O(\log^3 n)$ time on a CRCW PRAM using $n + m$ processors [20]. ∎

The next lemma, which bounds the number of executions of *Match-and-Push*, is the key to the analysis of the algorithm.

Lemma 4.3 Procedure *Match-and-Push* with activity parameter l and distance parameter k is executed at most $\frac{n(k+1)}{l} + 1$ times.

Proof: We show that at every execution of *Match-and-Push* there are at least l relabelings of nodes that have distance label below k at the beginning of the execution. Since the distance labels never decrease, the total number of such relabelings is at most $n(k + 1)$, and the desired bound follows.

We claim that a relabeling of each of the following nodes occurs during an execution of *Match-and-Push*:

1. The nodes in T which are matched in Step 1.
2. The active nodes in S which are not matched in Step 1.

Note that in every execution of *Match-and-Push*, except perhaps the last, the number of nodes satisfying these two conditions is at least l, so establishing this claim completes the proof of the theorem.

Suppose a node $w \in T$ is matched with a node $v \in S$ at Step 1, which implies that $d(v) \leq k$ and $d(w) = d(v) - 1 < k$. We show that $d(w)$ increases at Step 3 or at Step 5. If $d(w)$ increases at Step 3, we are done.

Consider a case in which $d(w)$ did not increase at Step 3. After Step 4, the only residual arc out of w is (w, v). (This follows from the observation that the outdegree of a node $w \in T$ in the residual graph G_f is $1 + e_f(w)$ and that after Step 4 we have $e_f(w) = 0$.) At Step 1, the arc (v, w) is admissible, and therefore $d(v) = d(w) + 1$. Node v has not been relabeled since then, so $d(v)$ did not change, and we have assumed that neither had $d(w)$. By the definition of the relabeling operation, at Step 5 the distance label of w becomes $d(v) + 1$, *i.e.*, the distance label increases by 2.

Now consider an active node $v \in S$ that is not matched at Step 1. Recall that by definition of an active node, $d(v) \leq k$, and thus the arc (v, s) cannot be admissible, because $d(s) = n$ and $d(v) \leq k \leq n$. Hence, in the beginning of Step 1 all admissible neighbors of v lie in T. These neighbors are matched during this step, and therefore by Step 6 their distance labels must increase (by the argument above). Since v has not acquired any new residual neighbors, its distance label must increase at Step 6. ∎

Lemmas 4.2 and 4.3, combined with an observation that the initialization of the first stage can be done in constant time using $n + m$ processors, imply the following result.

Lemma 4.4 The first stage of the bipartite matching algorithm runs in $O(\frac{nk}{l} \log^3 n)$ time using $n+m$ processors.

To complete the analysis of the algorithm, we prove the following lemma, which is similar to lemmas in [9, 36].

Lemma 4.5 After the first stage of the algorithm, the value of the residual flow is at most $n/k + l$, where k and l are the distance and activity parameters, respectively.

Proof: Since returning excesses to the source does not affect the amount of flow that can reach the sink, it suffices to show that the amount of flow that can reach the sink after the last execution of *Match-and-Push* is at most $n/k + l$. The proof uses the fact that for every node v, the label $d(v)$ is a lower bound on the distance in the residual graph from v to the sink [15].

Consider the pseudoflow f and the distance labeling d just after the last execution of *Match-and-Push*, and let $\overline{f}$ be an optimal flow. Consider the set of arcs $A = \{(i, j) | \overline{f}(i, j) > f(i, j)\}$. Note that $A \subseteq E_f$. Arcs in A can be partitioned into a collection of simple paths from nodes with excess to t, a collection of simple paths from nodes with excess to s, and a collection of cycles. As shown in [9, 36], by the properties of matching networks these paths and cycles are node-disjoint.

We need to show that the number of paths in the first collection is at most $l + n/k$. Consider a residual path from a node v to t. Since $d(v)$ is a lower bound on the distance from v to t in G_f, the length of such a path is at least $d(v)$. Since in the second stage at most l nodes are active, at most l paths in the first collection have length of k or less. The remaining paths have length greater than k, and the number of such paths is at most n/k, because the paths are node-disjoint. ∎

In the second stage of the algorithm, we find augmenting paths one by one. This stage terminates when there are no augmenting paths, and therefore produces correct results. The maximum number of augmenting paths found during this stage is bounded by Lemma 4.5, and therefore we have the following claim.

Lemma 4.6 The second stage of the bipartite matching algorithm runs in $O((\frac{n}{k} + l) \log^3 n)$ time using *BFS*(n, m) processors, where k and l are the distance and the activity parameters, respectively.

Remark: In the second stage, the algorithm can use any augmenting path. The fastest current parallel algorithm, however, finds a shortest augmenting path.

Theorem 4.7 The bipartite matching algorithm runs in $O(n^{2/3}\log^3 n)$ time using $BFS(n,m)$ processors, given an appropriate choice for k and l.

Proof: Set $k = \lfloor n^{1/3} \rfloor$ and $l = \lfloor n^{2/3} \rfloor$ and apply Lemmas 4.4 and 4.6. ∎

Sequential version Although the time-processor product of our parallel algorithm is quite large, the sequential version of the algorithm can be rebalanced to yield the $O(\sqrt{n}m)$ time bound, same as that of the fastest algorithms currently known [19, 36]. The rebalancing is achieved by setting the activity parameter l to 1 and the distance parameter k to $\sqrt{n}$. It is straight-forward to see that the resulting algorithm runs in $O(\sqrt{n}m)$ time. Like previous sequential algorithms [3, 25], this algorithm is based on a combination of the push-relabel and augmenting path methods.

4.3 Zero-One Flow Algorithms

In this section we describe algorithms for computing maximum flows in networks with unit arc capacities. We describe two algorithms, one optimized for general zero-one networks and another optimized for networks with no multiple arcs. For general zero-one networks, by transforming the input network into a matching network [6, 21] and applying the bipartite matching algorithm described in the previous section, it is possible to achieve a running time which is only a $\log^2 n$-factor worse than the running time of the algorithm described below. However, the method we describe in this section leads to a simpler algorithm with better running time bounds for networks with no multiple arcs.

First we describe the algorithm that finds a maximum flow in a general zero-one network (V, E, s, t) with unit arc capacities (see Figure 6). At a high level, this algorithm is similar to the algorithm of the previous section. The algorithm consists of two stages, where the first one is based on Goldberg's parallel maximum-flow algorithm, and the second is based on the Ford-Fulkerson method. The balancing of work done in the two stages is similar to that of the sequential algorithms of [9, 36]. In addition, the zero-one flow algorithm has a finish-up stage that converts a pseudoflow of maximum value with only positive excesses into a flow of maximum value, where the value of a pseudoflow is the amount flowing into the sink.

The first stage of the algorithm is essentially Goldberg's parallel maximum flow algorithm [14, 15, 17] but with two important modifications. First, we define an active node as a node that has a distance label of k or less, where k is the *distance parameter*. The second modification is that the stage terminates when the total amount of excess at active nodes is less than $l = m^{2/3}$, where l is the *activity parameter*. The key part of this stage is the *Push-and-Relabel* procedure. This procedure can be implemented by using either techniques of Shiloach and Vishkin [29] or

```
procedure Zero-One(V, E, s, t);

    [first stage]
    for all v ∈ V − {s} do d(v) ← 0;
    d(s) ← n;
    for all (v, w) ∈ E do f(v, w) ← 0;
    for all v ∈ V do e_f(v) ← 0;
    for all v ∈ V such that (s, v) ∈ E do begin
        f(s, v) ← 1;
        e_f(v) ← e_f(v) + 1;
    end;
    while the total amount of excess at active nodes is at least l do Push-and-Relabel;

    [second stage]
    while there is an augmenting path from a node v ∈ V − {s, t} such that e(v) > 0 to t do
        find an augmenting path from v to t and augment;

    [finish-up stage]
    if the current pseudoflow f_end is not a flow,
        convert it into a flow by recursively calling Zero-One;
    return(f̄);
end.
```

Figure 6: High-level description of the zero-one flow algorithm. For general zero-one networks, take $l = m^{2/3}$; for zero-one networks with no multiple arcs, take $l = \min(m^{2/3}, (nm)^{2/5})$.

Step 1. For all active nodes v, use a parallel prefix computation on the list of outgoing arcs to distribute $e_f(v)$ among neighbors of v in the admissible graph.
Step 2. For all nodes v, use parallel prefix computation on the list of incoming arcs to compute new excess $e_f(v)$ by adding up flow pushed to v during Step 1.
Step 3. Using parallel prefix computations, relabel all nodes $v \neq s, t$.

Figure 7: The *Push-and-Relabel* procedure.

by using parallel prefix computations (see *e.g.* [5, 23]) as described in [15]. Figure 7 describes the procedure. (Note that Step 3 of *Push-and-Relabel* can be replaced by a computation of true distances from all vertices to the sink in the residual graph, which can be achieved by doing a breadth-first search backwards from the sink.) The first stage is fast because each execution of *Push-and-Relabel* performs either a large number of pushes or a large number of relabelings.

The second stage of the algorithm repeatedly finds an augmenting path from a node $v \notin \{s, t\}$ with $e_f(v) > 0$ to the sink, and augments along the path. One way to find an augmenting path is to do a breadth-first search backwards from the sink in the residual graph. Observe that when no such paths exist, the current pseudoflow f_{end} is of maximum value. Moreover, since we have not created any negative excesses, all excesses are positive.

The finish-up stage converts f_{end} into a flow $\overline{f}$ by returning excesses from nodes in $V - \{s, t\}$ to s. This conversion is done by running the same algorithm on a modified network. The modified network is obtained from $G_{f_{end}}$ by adding a new source s' and new arcs of unit capacity connecting s' to nodes in $V - \{s, t\}$ that have excesses with respect to f_{end}. If a node v has excess $e_{f_{end}}(v)$, then $e_{f_{end}}(v)$ arcs of the form (s', v) are added. The source of the original network is the sink of the modified network. It can be easily shown (see [14, 17]) that when the zero-one flow algorithm is applied to the modified network, its second stage terminates with a flow (rather than a pseudoflow).

The correctness of the algorithm can be shown in exactly the same way as it was done in [17, 15]. Performance of the *Push-and-Relabel* procedure is summarized by the following lemma, which follows from the results of [5] and [15, 29].

Lemma 4.8 Procedure *Push-and-Relabel* runs in $O(\log n)$ time using m processors.

The next lemma, which is the key to the analysis of this algorithm, bounds the number of times *Push-and-Relabel* is executed.

Lemma 4.9 Procedure *Push-and-Relabel* is executed $O(\frac{mk}{l})$ times, where k and l are the distance and activity parameters, respectively.

Proof: Define the potential function Φ to be equal to the number of pushes plus the sum over all nodes $v \in V$ of the label $d(v)$ times the degree of v, where by "degree" we mean the sum of indegree and outdegree.

The pushes are made only through admissible arcs, which means that if a push was made from v to w, the next push associated with this edge has to be from w to v, which can happen only after the label of w was increased by at least 2. The algorithm executes pushes only from nodes with label of at most k, and hence the total number of pushes is $O(mk)$. Therefore Φ is bounded by $O(mk)$ as well.

Consider a unit of excess at an active node v at the beginning of the *Push-and-Relabel* procedure. If this unit participated in a push at Step 1, it increased Φ by one. Otherwise, the label of v increased by at least 1 during Step 3. This means that Φ was increased by at least the degree of v during this step. Observe that each unit of excess at v can be associated with the arc over which it reached v, and hence the degree of v is at least equal to the number of units of excess at v. During the first stage the total excess at active nodes is at least l, which means that each call to *Push-and-Relabel* increases Φ by at least l, and the bound follows. ∎

Lemmas 4.8 and 4.9, combined with an observation that the initialization of the first stage can be done in constant time using $n + m$ processors, imply the following result.

Lemma 4.10 The first stage of the zero-one flow algorithm runs in $O(\frac{mk}{l} \log n)$ time using $n + m$ processors, where k and l are the distance and activity parameters, respectively.

In the second stage we find augmenting paths one by one. To bound the running time of the stage, we first bound the value of the residual flow after the execution of the first stage. The following lemma is similar to Lemma 4.3.

Lemma 4.11 After the first stage of the algorithm is applied to a general zero-one network, the value of the residual flow is at most $m/k + l$.

Proof: The proof of this lemma is exactly like the proof of Lemma 4.5. The only difference is that, since we are dealing with the zero-one networks instead of the matching networks, we decompose the residual flow into arc-disjoint paths instead of node-disjoint paths. ∎

Lemma 4.12 The second stage of the zero-one flow algorithm runs in $O((\frac{m}{k} + l) \log n)$ time using *BFS*(n, m) processors, where k and l are distance and activity parameters, respectively.

Proof: The lemma follows from Lemma 4.11. ∎

Theorem 4.13 On general zero-one networks, the zero-one flow algorithm runs in $O(m^{2/3} \log n)$ time using *BFS*(n, m) processors, given an appropriate choice of k and l.

Proof: Set $k = \lfloor m^{1/3} \rfloor$ and $l = \lfloor m^{2/3} \rfloor$. Lemmas 4.10 and 4.12 imply that the first two stages of the algorithm run in the desired resource bound. These lemmas also imply that the finish-up stage runs in the same resource bounds. ∎

Now we consider the problem of finding maximum flows in zero-one networks with no multiple arcs. In this case, we can improve the time bound of Theorem 4.13 for dense graphs (more precisely, for $m > n^{3/2}$). The following lemma, which is an adaptation of similar lemmas from [9, 36], is a key to the improvement.

Lemma 4.14 After the first stage of the algorithm is applied to a zero-one network with no multiple arcs, the value of the residual flow is at most $\left(\frac{2n}{k-1}\right)^2 + l$, where k and l are distance and activity parameters, respectively.

Proof: Since returning excesses to the source does not affect the amount of flow that can reach the sink, it suffices to show that the amount of flow that can reach the sink after the last execution of *Push-and-Relabel* is at most $\left(\frac{2n}{k-1}\right)^2 + l$.

Consider the pseudoflow f and the distance labeling d just after the last execution of *Match-and-Push*, and let $\overline{f}$ be an optimal flow. Consider the set of arcs $A = \{(i,j) | \overline{f}(i,j) > f(i,j)\}$. Note that $A \subseteq E_f$. Arcs in A can be partitioned into a collection of simple paths from nodes with excess to t, a collection of simple paths from nodes with excess to s, and a collection of cycles. Since we have a zero-one network, these paths and cycles are arc-disjoint. The number of paths in the first collection that start at a node with a distance label of k or less is at most l.

To complete the proof, we need to show that the number of paths that start at a node with a distance label greater than k and reach the sink is at most $\left(\frac{2n}{k-1}\right)^2$. Suppose for contradiction that this is false. Let P be the set of these paths, and let $G' = (V, E')$ be the graph induced by arcs on paths in P. Let $d'(v)$ be the distance in G' from v to t. Let $V_i = \{v \in V | d'(v) = i\}$. By definition of P, no path in P starts at a node in the set $V_0 \bigcup V_1 \bigcup \ldots \bigcup V_k$. Therefore $|P|$ is bounded, for $0 \leq j \leq k-1$, by the number of arcs in the set $E \bigcap (V_j \times V_{j+1})$, which is at most $|V_j| \times |V_{j+1}|$, since the network has no multiple arcs. Our assumption implies that $|V_j| \times |V_{j+1}| > \left(\frac{2n}{k-1}\right)^2$, and therefore $|V_j| + |V_{j+1}| > \frac{2n}{k-1}$, for $0 \leq j \leq k-1$. We obtain a contradiction as follows:

$$\begin{aligned} & (V_0 + V_1) + (V_2 + V_3) + \ldots + (V_{2\lfloor k/2 \rfloor - 1} + V_{2\lfloor k/2 \rfloor}) \\ > \; & \frac{2n}{k-1} \lfloor \frac{k}{2} \rfloor \\ \geq \; & \frac{2n}{k-1} \frac{k-1}{2} \\ = \; & n. \end{aligned}$$

∎

Using Lemmas 4.11 and 4.14, one can obtain the following theorem; the proof is similar to the proof of Theorem 4.13.

Theorem 4.15 On zero-one capacity networks with no multiple arcs, the maximum flow algorithm runs in time $O(\min(m^{2/3}, (nm)^{2/5}) \log n)$ on a CRCW PRAM using $BFS(n,m)$ processors.

5 The Assignment Problem and Minimum-Cost Flows

In this section we describe parallel algorithms for the weighted versions of the problems studied in the previous section, namely the assignment problem and the minimum-cost flow problem with unit capacities. The section consists of three parts. In the first part we present some definitions and review the generic minimum-cost flow algorithm of Goldberg and Tarjan [15, 18]. In the second part we present our parallel algorithm for weighted bipartite matching, and in the third part we discuss the problem of finding a minimum-cost flow in a general network with unit capacities. For the purpose of this section, we assume familiarity with the minimum-cost flow framework developed by Goldberg and Tarjan [15, 16, 18].

5.1 Goldberg-Tarjan Minimum-Cost Flow Algorithm

In this section we review the main ideas of the minimum-cost flow framework developed by Goldberg and Tarjan [15, 18]. We review only those ideas that are relevant for networks with unit capacities.

In order to describe the algorithm, we need to introduce a few definitions (see [15, 16, 18] for more details). Each node v is assigned a *price* $p(v)$. Given p, the *reduced cost* of an arc (v, w) is defined by

$$c_p(v, w) = p(v) - p(w) + c(v, w),$$

where $c(v, w)$ is the original cost that is part of the input to the problem. We say that a pseudoflow is *ϵ-optimal* [32] if there exists a price function p such that, with respect to this price function, there are no residual arcs with reduced cost below $-\epsilon$. Given a pseudoflow f and a price function p, an arc $(v, w) \in E$ is *admissible* if it is a residual arc with negative reduced cost, *i.e.*, if $u_f(v, w) > 0$ and $c_p(v, w) < 0$. The *admissible graph* is the graph induced by the set of admissible arcs. Define $c(f)$, the *cost* of pseudoflow f, by

$$\sum_{(v,w) \in E : f(v,w) > 0} c(v, w) f(v, w).$$

In the case of zero-one flows, the cost of a pseudoflow is equal to the sum of the costs of the saturated arcs.

The outer loop of the minimum-cost flow algorithm, shown in Figure 8, does generalized cost-scaling [15, 18]. Initially $\epsilon = C$, where C is the maximum absolute value of an input cost. The algorithm iteratively halves ϵ and uses the *Generic-Refine* procedure (see Figure 9) to update the flow to be ϵ-optimal again. Bertsekas [4] noticed that if the costs are integral then an ϵ-optimal flow is optimal for $\epsilon < 1/n$, and therefore we have the following lemma.

Lemma 5.1 (Goldberg and Tarjan [15, 18]) The algorithm terminates and produces an optimal flow after $O(\log nC)$ calls to the *Generic-Refine* procedure.

```
procedure Min-Cost-Flow(V, E, c);

    C ← max{|c(v, w)| : (v, w) ∈ E};
    ε ← C;
    for all v ∈ V do p(v) ← 0;
    p(s) = −2nC;

    while ε ≥ 1/n do
        ε ← ε/2;
        (c, p, f) ← Generic-Refine(V, E, c, p, f, ε);
    end;
    return the current (maximum) flow f;
end.
```

Figure 8: High-level description of the outer (scaling) loop of the generic Goldberg-Tarjan minimum-cost flow algorithm.

The heart of the algorithm is the *Generic-Refine* procedure, shown in Figure 9, that converts a 2ϵ-optimal flow into an ϵ-optimal flow. The procedure is similar to the maximum flow algorithm which we reviewed in Section 4.1. *Generic-Refine* starts with constructing an ϵ-optimal pseudoflow by saturating residual arcs with reduced cost of below $-\epsilon$ and creating appropriate excesses and deficits at the nodes. Then positive excesses are moved towards negative ones using *Push* and *Relabel* procedures, described in Figure 10. Similar to the case of the generic maximum flow, *Push* moves one unit of excess through an admissible arc (*i.e.*, an arc with negative reduced cost); *Relabel* changes the price p of a node to create an outgoing admissible arc from this node while maintaining ϵ-optimality.

Goldberg and Tarjan proved that the *Min-Cost-Flow* algorithm produces a feasible minimum-cost flow of maximum value upon termination. Moreover, they showed that during a single execution of *Generic-Refine* the amount of relabeling of a single node is bounded by $O(n\epsilon)$, which immediately implies an $O(nm)$ bound on the number of times a *Push* is executed during a single execution of *Generic-Refine* for the case of flows in networks with unit capacities.

5.2 The Assignment Problem

The assignment problem is a weighted version of the bipartite matching problem. Similar to the unweighted case described in Section 4.2, we transform the assignment problem into a minimum-cost flow problem with unit capacities in the standard way (see, for example, [22]) where the weights on the edges are mapped into costs on the arcs of the corresponding matching network. As in Section 4.2, the nodes and edges in the resulting matching network are denoted by V and E, respectively. Without loss of generality, we assume that a perfect matching exists. To assure this we can always add a matching with arcs of very high cost.

The outer loop of the minimum-weight matching algorithm (see Figure 11) is the same as the

```
procedure Generic-Refine(V, E, c, p, f, ε);

    [Convert into ε-optimal pseudoflow.]

    for all {(v, w) | (v, w) ∈ E_f and c_p(v, w) < −ε} do
        f(v, w) ← f(v, w) + 1;
        f(w, v) ← f(w, v) − 1;
        e_f(w) ← e_f(w) + 1;
        e_f(v) ← e_f(v) − 1;
    end;

    [convert into ε-optimal flow.]

    while there exists a node with excess do
            apply Push and Relabel in arbitrary order;

    return (c, p, f);

end.
```

Figure 9: High-level description of the inner loop of the generic Goldberg-Tarjan minimum-cost flow algorithm.

outer loop of the Goldberg-Tarjan minimum-cost flow algorithm (see Figure 8.) The heart of the algorithm is the *Refine* procedure, shown in Figure 12, that converts a 2ϵ-optimal flow into an ϵ-optimal flow. The procedure starts by decreasing the prices of all the nodes in T by 2ϵ. (Though somewhat unnatural, this is essential for the proof of Lemma 5.5.) Next it constructs an ϵ-optimal pseudoflow by saturating residual arcs with reduced cost below $-\epsilon$ and creating appropriate excesses and deficits at the nodes.

The resulting pseudoflow is converted into an ϵ-optimal flow in two stages. The first stage iteratively uses the *Match-and-Push* procedure (see Figure 5) to push the positive excesses towards

Push(v, w)**.**
Applicability: $e(v) > 0$, $u_f(v, w) > 0$ **and** $c_p(v, w) = p(v) - p(w) + c(v, w) \leq 0$**.**
Action: **Send one unit of flow from** v **to** w **as follows:**
$f(v, w) \leftarrow f(v, w) + 1$; $f(w, v) \leftarrow f(w, v) - 1$;
$e_f(v) \leftarrow e_f(v) - 1$; $e_f(w) \leftarrow e_f(w) + 1$.

Relabel(v)**.**
Applicability: **Any** v**.**
Action: $p(v) \leftarrow \max\{p(w) - c(v, w) - \epsilon | (v, w) \in E_f\}$.
(If this maximum is over an empty set, $p(v) \leftarrow -\infty$**.)**

Figure 10: Push and relabel operations for minimum-cost flow computation.

```
procedure Assignment(S, T, E, c);

    [Initialization]
    transform the input problem into network flow form;

    C ← max{|c(v, w)| : (v, w) ∈ E};
    ε ← C;
    for all v ∈ V do p(v) ← 0;
    p(s) = −2nC;

    while ε ≥ 1/n do
        ε ← ε/2;
        (c, p, f) ← Refine(V, E, c, p, f, ε);
    end;
    return the matching corresponding to current (maximum) flow f;
end.
```

Figure 11: High-level description of the outer (scaling) loop of the assignment algorithm.

the negative ones. The *Match-and-Push* procedure used during this stage is exactly the same as for the unweighted case (see Figure 5), except that *Push* and *Relabel* are generalized to the weighted case as described in Figure 10. We say that a node v is *active* if $e_f(v) > 0$ and the price change of this node during the current invocation of *Refine* is below $k\epsilon$, where k is the *distance parameter*. (Note the similarity between this definition of the distance parameter and the definition which we have used in the algorithms for the unweighted case.) The first stage is executed as long as the number of active nodes exceeds l. (The best running time is achieved for activity parameter $l = \lfloor n^{2/3} \rfloor$ and distance parameter $k = \lfloor n^{1/3} \rfloor$.)

The second stage of *Refine* finds augmenting paths from nodes with excess to the sink and augments along these paths. The paths used for the augmentations are shortest paths with respect to the distance function *length* obtained by adding ϵ to reduced costs. Since we have assumed that the input graph has a perfect matching, *Refine* terminates with a flow.

Since by construction the algorithm does not terminate as long as there are excesses, in order to prove the correctness of the algorithm it suffices to show that both stages preserve ϵ-optimality.

Lemma 5.2 Procedure *Match-and-Push* preserves ϵ-optimality of the pseudoflow.

Proof: Note that since *Match-and-Push* relabels only vertices with no outgoing admissible arcs, a relabeling can not cause an increase in the price of a node. Assume that after the execution of *Match-and-Push* there exists a residual arc (v, w) such that $c_p(v, w) < -\epsilon$. This means that when v was last relabeled either the price of w was lower, or this arc did not exist. The prices are nonincreasing, and hence the first case is impossible. If (v, w) was not in the residual graph when v was last relabeled, there was a *Push* from w to v since then. During this push the reduced cost of (v, w) was positive, which means that the price of w increased since then, which is impossible. ∎

```
procedure Refine(V, E, c, p, f, ε);
    [Reduce the number of arcs with negative reduced cost.]

    for all v ∈ T do p(v) ← p(v) − 2ε;

    [Convert into ε-optimal pseudoflow.]

    for all {(v, w) | (v, w) ∈ E_f and c_p(v, w) < −ε} do
        f(v, w) ← f(v, w) + 1;
        f(w, v) ← f(w, v) − 1;
        e_f(w) ← e_f(w) + 1;
        e_f(v) ← e_f(v) − 1;
    end;

    [First stage.]

    while the number of active nodes is at least l do Match-and-Push;

    [Second stage]

    for all (v, w) ∈ E do length(v, w) ← c_p(v, w) + ε;
    while there are nodes with excess do begin
        let Γ be a shortest path w.r.t. length in G_f from a node with excess to t;
        augment along Γ;
    end;
    for all v ∈ V do p(v) ← p(v) − the distance from v to the sink in G_f,
        where distance is computed w.r.t. length;

    return (c, p, f);

end.
```

Figure 12: High-level description of the inner loop of the assignment algorithm. For the algorithm described in this paper, we take $l = \lfloor n^{2/3} \rfloor$.

The following lemma shows that the second stage preserves ϵ-optimality.

Lemma 5.3 Suppose a pseudoflow f is ϵ-optimal, and suppose $length : E \to R$ is defined by $length(v, w) = c_p(v, w) + \epsilon$. Let Γ be a shortest path with respect to $length$ in G_f from a node with excess to the sink t. Then augmentation along Γ preserves ϵ-optimality.

Proof: Relabel the graph by setting the price of each node to be equal to the negative of the distance from this node to the sink in the residual graph, where the distance is computed with respect to *length*. Observe that the reduced costs of the residual arcs in this graph are at least $-\epsilon$, and the arcs on Γ have reduced costs of exactly $-\epsilon$. Thus the residual arcs created by the augmentation have reduced costs of $\epsilon > 0$, so the ϵ-optimality is preserved.

Lemma 5.5 bounds the residual flow after the first stage. This lemma is similar to the lemma due to Gabow and Tarjan [12], which they used to construct an $O(m\sqrt{n}\log nC)$-time sequential algorithm for weighted bipartite matching in the Hungarian framework. Our proof is similar to that of Gabow and Tarjan, but is more involved since we do not have a matching in the middle of the execution of the algorithm. The following lemma is the key to the result of Lemma 5.5.

Lemma 5.4 Consider the set of nodes that have excesses at the end of the first stage of the algorithm. The sum of price decreases of these nodes during the stage is $O(n\epsilon)$.

Proof: Intuitively, the main idea of the proof is as follows. Consider an ϵ-optimal pseudoflow f'_ϵ at some point during the execution of *Refine* and the ϵ-optimal flow f_ϵ at the end of the execution of *Refine*. The difference between f'_ϵ and f_ϵ can be decomposed into node-disjoint paths and cycles. The cost of pseudoflow f'_ϵ is less than the cost of flow f_ϵ, and the difference in cost is equal to the sum of all the costs of these paths and cycles. On the other hand, this difference is "small" since both f_ϵ and f'_ϵ are ϵ-optimal. This leads to a bound on the sum of the costs of the paths. Now we observe that the cost of a path depends on the amount of relabeling of its end-points during the execution of *Refine*, which leads to the proof of the claim of the lemma.

The first step decreases the prices of all the nodes in T by 2ϵ. This increases the reduced costs of arcs that go into nodes in T by 2ϵ. The input is 2ϵ-optimal, and therefore after this increase all residual arcs that go into nodes in T have positive reduced cost. Hence the number of remaining residual arcs with negative reduced cost is at most n. (Recall that $n = |S| + |T| + 2$.) On the other hand, the reduced cost of arcs going out of nodes in T is decreased by 2ϵ, and therefore the flow is 4ϵ-optimal with respect to the new prices.

For the purpose of the proof, we assume that at this point all costs are replaced by the reduced costs, and all prices are set to zero. We call the resulting costs *transformed*. After this transformation, the residual network has the following properties:

1. The (transformed) costs of residual arcs are at least -4ϵ.
2. At most n residual arcs have negative (transformed) costs.

Therefore, for any pseudoflow f in the network we have

$$\text{cost}(f) \geq -4n\epsilon. \tag{1}$$

To bound the cost of any ϵ-optimal flow f_ϵ from above, consider the decomposition of this flow into paths from s to t and cycles. The network is a matching network and therefore these paths and cycles are node-disjoint. The prices of both s and t are zero because they are never relabeled, and therefore the cost of f_ϵ is equal to the sum of the reduced costs of the saturated arcs, independent

of the prices of the nodes other than s and t. Let p be the prices (associated with flow f_ϵ) at the end of the execution of *Refine*. For any saturated arc (v,w) (*i.e.*, $f_\epsilon(v,w)=1$), there is a residual arc $(w,v) \in E_{f_\epsilon}$ with reduced cost $c_p(w,v) = -c_p(v,w) \geq -\epsilon$, and therefore we have

$$\begin{aligned} \text{cost}(f_\epsilon) &= \sum_{(v,w):f_\epsilon(v,w)=1} c_p(v,w) \\ &\leq n\epsilon. \end{aligned} \tag{2}$$

Consider an ϵ-optimal pseudoflow f'_ϵ and the associated prices p' at some point of the execution of the first stage. From (1) and (2) we have

$$\text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon) \leq 5n\epsilon. \tag{3}$$

Next we obtain a lower bound on $\text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon)$. Define the set of arcs $A' = \{(i,j) | f_\epsilon(i,j) > f'_\epsilon(i,j)\}$. Arcs in A' can be partitioned into simple paths from nodes with excess to nodes with deficit or to the sink t, simple paths from s to t, and simple cycles. Let S_1 be the set of paths from nodes with excess to nodes with deficit or to t, let S_2 be the set of paths from s to t, and let S_3 be the set of cycles. Note that $A' \subseteq E_{f'_\epsilon}$ where $E_{f'_\epsilon}$ is the set of residual arcs of the pseudoflow f'_ϵ, and therefore the reduced cost (with respect to p') of any arc in A' is at least $-\epsilon$. Let $|P|$ denote the length of P. Then the cost of a path P from node v with excess to node w with deficit or to t is

$$\begin{aligned} \text{cost}(P) &= \sum_{e \in P} c(e) \\ &= \sum_{e \in P} c_{p'}(e) + p'(w) - p'(v) \\ &\geq -p'(v) - |P|\epsilon. \end{aligned}$$

The last inequality holds because neither t nor nodes with deficit are relabeled during execution of *Refine*. If P is either a path in S_2 or a cycle in S_3, then $\text{cost}(P) \geq -|P|\epsilon$. We have

$$\begin{aligned} \text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon) &= \sum_{P \in S_1} \text{cost}(P) + \sum_{P \in S_2} \text{cost}(P) + \sum_{P \in S_3} \text{cost}(P) \\ &\geq -\sum_{e(v)>0} p'(v) - \left(\epsilon \sum_{P \in S_1 \bigcup S_2 \bigcup S_3} |P| \right). \end{aligned}$$

By the properties of matching networks, the sum of the lengths of the paths in $S_1 \bigcup S_2$ and the cycles in S_3 is at most n. Therefore we have

$$\text{cost}(f_\epsilon) - \text{cost}(f'_\epsilon) \geq -\sum_{e(v)>0} p'(v) - n\epsilon. \tag{4}$$

From (3) and (4) we obtain

$$-\sum_{e(v)>0} p'(v) \leq 6n\epsilon.$$

This gives the desired bound on the sum of price decreases at the nodes that have excesses at the end of the first stage. Hence, the number of nodes that were relabeled by $k\epsilon$ during the first stage is at most $6n/k$. ∎

Lemma 5.5 After the first stage of *Refine*, the value of the residual flow is at most $O(n/k + l)$.

Proof: Consider nodes other than s and t that have excesses at the end of the first stage. By Lemma 5.4, the sum of price decreases at these nodes during the first stage is bounded by $O(n\epsilon)$. At the end of the first stage there can be at most l nodes with excesses such that their price was decreased by less than $k\epsilon$ during the first stage. Thus at most $O(n/k + l)$ nodes can have excess at the end of the first stage. Each excess has a value of 1, and therefore the total amount of excess at the end of the stage is $O(n/k + l)$. This gives the desired bound on the value of the residual flow. ∎

Lemma 5.5 bounds the number of iterations of the second stage of *Refine*. Next we bound the number of iterations of the first stage. Note that the *Match-and-Push* procedure maintains the invariants defined in Lemma 4.1.

Lemma 5.6 There are at most $\frac{n(k+1)}{l} + l$ calls to *Match-and-Push* in the first stage of *Refine*, where k and l are the distance and activity parameters, respectively.

Proof: The proof is similar to the proof of Lemma 4.3. The main idea is that as long as there are at least l active nodes, prices of at least l nodes decrease by at least ϵ each during one invocation of *Match-and-Push*. The lemma follows because the total amount of relabeling (price change) during the first stage of *Refine* is bounded by $nk\epsilon$.

Consider an active node $v \in S$ that was matched with $w \in T$ during the first step of *Match-and-Push*. If v pushes to w and w pushes to some node $v' \neq v$, the only residual arc from w after this push is (w, v). But the arc (v, w) was admissible, and therefore in the beginning of the iteration $c_p(v, w) = p(v) - p(w) + c(v, w) < 0$. By definition of Relabel (see Figure 10), w is relabeled by

$$\begin{aligned} \Delta p(w) &= p(w) + c(w, v) - p(v) + \epsilon \\ &\geq -c_p(v, w) + \epsilon \\ &\geq \epsilon. \end{aligned}$$

Similarly, if the push was back to v, it is easy to see that w is also relabeled by at least ϵ.

Consider a node $v \in S$ that is not matched during the first step. Note that the matching obtained in Step 1 of *Match-and-Push* is maximal. From the previous discussion it follows that

each node $w \in T$, such that (v, w) is a residual arc, is relabeled by at least ϵ. Hence, at Step 6, v is also relabeled by at least ϵ. ∎

Given the similarity of the above lemmas with the corresponding lemmas for the unweighted case, it is not surprising that the running times are similar.

Theorem 5.7 The assignment algorithm runs in $O(n^{2/3} \log^3 n \log(nC))$ time using $SSP(n, m)$ processors.

Proof: The initialization of *Refine* can be done in $O(\log n)$ time with $n + m$ processors. By Lemma 5.6, the first stage makes $O(\frac{nk}{l} + l)$ calls to *Match-and-Push*. By Lemma 4.2, *Match-and-Push* runs in $O(\log^3 n)$ time on a CRCW PRAM with $n + m$ processors. Hence, by Lemma 5.5, each execution of *Refine* takes $O((\frac{nk}{l} + l) \log^3 n)$ time on a CRCW PRAM using $SSP(n, m)$ processors.

By Lemma 5.1, after $O(\log(nC))$ iterations of *Refine* the resulting flow is optimal. The claim follows by setting $k = \lfloor n^{1/3} \rfloor$, $l = \lfloor n^{2/3} \rfloor$. ∎

Sequential version As in the bipartite matching case, our algorithm for the assignment problem has an efficient sequential version. This version runs in $O(\sqrt{n} m \log(nC))$ time *i.e.,* as fast as the algorithm of Gabow and Tarjan [12]. The sequential version is obtained by setting l to 1, k to $\sqrt{n}$, and using shortest path algorithms of [7, 34] to solve the shortest path subproblems in linear time, as described in [18]. A somewhat different sequential algorithm that combines the push-relabel and augmenting path methods and achieves the same time bound has been proposed independently by Orlin and Ahuja [26].

5.3 Zero-One Minimum-Cost Flows

The ideas of the previous section can be extended to develop a parallel algorithm that finds a maximum flow of minimum cost in graphs with zero-one capacities. The idea is to change the *Refine* procedure (see Figure 12) to use *Push-and-Relabel* procedure (see Figure 7) instead of *Match-and-Push* in the "while loop". The resulting algorithm runs in $O(m^{2/3} \log^2 n \log(nC))$ time using $O(SSP(n, m))$ processors.

Note that a slightly worse running time can be obtained by reducing the minimum-cost flow problem to weighted bipartite matching (see, for example, the paper of Chandra, Stockmeyer, and Vishkin [6]), and applying the algorithm described in the previous section. The transformation is as follows. First, construct a line digraph $G' = (V', E')$ of G, where

$$\begin{aligned} V' &= E \\ E' &= \{((u, v), (v, w)) | (u, v), (v, w) \in E\} \\ weight((u, v), (v, w)) &= c(u, v) + c(v, w) \end{aligned}$$

Call the nodes of G' that correspond to arcs of G leaving the source or entering the sink "sources" and "sinks", respectively. Observe that there is a one-to-one correspondence between a minimum-cost flow in G and a maximum set of node-disjoint paths of minimum weight from sources to sinks in G'. Split each node v in G', except the sources or sinks, into two nodes v_1 and v_2. Connect these nodes by a zero-weight edge, and connect v_2 to u_1 for each arc (v, u) in G'. Add a complete bipartite graph between sources and sinks with edges of high weight. Call the resulting bipartite graph G''. It is easy to see that the minimum-weight perfect matching in G'' corresponds to the maximum set of node-disjoint paths of minimum weight in G', and therefore solves the minimum-cost flow problem for G. The number of nodes in G'' is $O(m)$, which leads to an $O(m^{2/3} \log^3 n \log(nC))$ running time.

Remark: We can extend our results to networks with arbitrary integral capacities represented in unary by replacing every arc (v, w) of capacity $u(v, w)$ and cost $c(v, w)$ by $u(v, w)$ arcs of capacity one and cost $c(v, w)$.

Remark: Note that our algorithms find an optimal primal solution, but not an optimal dual solution. Our algorithms can be modified to compute an optimal dual solution as well, without increasing the asymptotic running time and processor bounds. See *e.g.* [12].

6 Conclusions

The problems discussed in this paper are important tools for the design of efficient algorithms both in the context of parallel and sequential computation. For example, a linear-time sequential depth-first search algorithm leads to linear-time algorithms for many other problems; NC algorithms for the above problems would result in NC algorithms for many other problems as well. Our results are a step towards the design of efficient parallel algorithms for these problems and we believe that the ideas of this paper may lead to improved sequential algorithms as well.

We would like to note that, since the sequential versions of our algorithms for network flow and matching problems give running time bounds that match the best known bounds for the corresponding problems, improving our parallel time bounds (by more than a few factors of $\log n$) may require improving these sequential time bounds.

Our weighted bipartite matching algorithm can be modified to use a linear number of processors. Recall that the algorithm works in two stages, where the first stage uses only a linear number of processors, and the second stage uses as many processors as needed to compute shortest paths in polylogarithmic time. Using a linear-processor connectivity algorithm in the second stage and rebalancing the distance and the activity parameters, leads to a linear-processor algorithm that solves the weighted bipartite matching problem in $O(n^{4/3} \log^3 n \log nC)$ time. The same transformation can be applied to our algorithm for finding minimum-cost flow in networks with unit capacities, leading to a linear-processor algorithm with $O((mn)^{2/3} \log^2 n \log nC)$ running time. Note that these algorithms work in the same time bounds in a distributed network.

Acknowledgments

We would like to thank Alok Aggarwal, Baruch Awerbuch, Clyde Kruskal, Vijaya Ramachandran, and Bob Tarjan for helpful discussions. The second author would like to thank Charles Leiserson for his enthusiastic support and encouragement.

References

[1] A. Aggarwal and R. J. Anderson. A Random NC Algorithm for Depth First Search. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 325–334, 1987.

[2] R. J. Anderson. A Parallel Algorithm for the Maximal Path Problem. *Combinatorica*, 7, 1987.

[3] B. Awerbuch. Personal Communication. Laboratory for Computer Science, M.I.T., 1985.

[4] D. P. Bertsekas. Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems. Technical Report LIDS-P-1986, Lab. for Decision Systems, M.I.T., September 1986. (Revised November, 1986).

[5] G. Blelloch. Parallel Prefix vs. Concurrent Memory Access. Technical report, Thinking Machines, Inc., 1986.

[6] A. K. Chandra, L. Stockmeyer, and U. Vishkin. Constant Depth Reducibility. *SIAM J. Comput.*, 13:423–439, May 1984.

[7] R. B. Dial. Algorithm 360: Shortest path forest with topological ordering. *Comm. ACM*, 12:632–633, 1969.

[8] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[9] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[10] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[11] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[12] H. N. Gabow and R. E. Tarjan. Almost-Optimal Speed-ups of Algorithms for Matching and Related Problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 514–527, 1988.

[13] R. K. Ghosh and G. P. Bhattacharjee. A Parallel Search Algorithm for Directed Acyclic Graphs. *BIT*, 24:134–150, 1984.

[14] A. V. Goldberg. A New Max-Flow Algorithm. Technical Report MIT/LCS/TM-291, Laboratory for Computer Science, M.I.T., 1985.

[15] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers.* PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

[16] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.

[17] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988. A preliminary version appeared in *Proc. 18th ACM Symp. on Theory of Comp.*, 136-146, 1986.

[18] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Successive Approximation. *Math. of Oper. Res.*, 15:430–466, 1990. A preliminary version appeared in *Proc. 19th ACM Symp. on Theory of Comp.*, 7-18, 1987.

[19] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[20] A. Israeli and Y. Shiloach. An Improved Parallel Algorithm for Maximal Matching. *Information Processing Let.*, 22:57–60, 1986.

[21] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a Maximum Matching is in Random NC. *Combinatorica*, 6:35–48, 1986.

[22] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[23] C. E. Leiserson and B. M. Maggs. Communication-Efficient Parallel Graph Algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

[24] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as Easy as Matrix Inversion. *Combinatorica*, pages 105–131, 1987.

[25] J. B. Orlin and R. K. Ahuja. New Distance-Directed Algorithms for Maximum-Flow and Parametric Maximum-Flow Problems. Sloan Working Paper 1908-87, Sloan School of Management, M.I.T., 1987.

[26] J. B. Orlin and R. K. Ahuja. New Scaling Algorithms for Assignment and Minimum Cycle Mean Problems. Sloan Working Paper 2019-88, Sloan School of Management, M.I.T., 1988.

[27] V. Pan and J. Reif. Efficient Parallel Solution of Linear Systems. In *Proc. 17th Annual ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[28] B. Schieber and S. Moran. Slowing Down Sequential Algorithms for Obtaining Fast Distributed and Parallel Algorithms: Maximum Matchings. In *Proc. 5th Annual ACM Symposium on Principles of Distributed Computing*, 1986.

[29] Y. Shiloach and U. Vishkin. An $O(\log n)$ Parallel Connectivity Algorithm. *J. Algorithms*, 3:57–67, 1982.

[30] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.

[31] J. Smith. Parallel Algorithms for Depth-First Searches I. Planar Graphs. *SIAM Journal on Computing*, 15:814–830, 1986.

[32] É. Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, 5(3):247–255, 1985.

[33] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[34] R. A. Wagner. A Shortest Path Algorithm for Edge-Sparse Graphs. *J. Assoc. Comput. Mach.*, 23:50–57, 1976.

[35] Г. М. Адельсон-Вельский, Е. А. Диниц, и А. В. Карзанов. *Потоковые Алгоритмы.* Наука, Москва, 1985. English transcription: G. M. Adel'son-Vel'ski, E. A. Dinitz, and A. V. Karzanov, *"Potokovye Algoritmy,"* Science, Moscow. Title translation: Flow Algorithms.

[36] А. В. Карзанов. Оценка Алгоритма Нахождения Максимального Потока, Примененного к Задаче о Представителях. Сб. *Вопросы Кибернетики. Труды Семинара по Комбинаторной Математике. (Москва 1971).* Советское Радио, Москва, 1973. English transcription: A. V. Karzanov, "Otsenka Algoritma Nokhozhdeniya Maksimal'nogo Potoka, Primenennogo k Zadache o Predstavitelyakh," *Voprosy Kibernetiki, Trudy Seminara po Kombinatornoi Matematike*, Soviet Radio, Moscow. Title translation: A Bound on a Maximum Flow Algorithm Applied to the Problem of Distinct Representatives.