LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-357

SUBLINEAR-TIME PARALLEL ALGORITHMS FOR MATCHING AND RELATED PROBLEMS

Andrew V. Goldberg

Serge A. Plotkin

Pravin Vaidya

June 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Sublinear-Time Parallel Algorithms for Matching and Related Problems

Andrew V. Goldberg[*]
Department of Computer Science
Stanford University
Stanford, CA 94305

Serge A. Plotkin[†]
Laboratory for Computer Science
M.I.T.
Cambridge, MA 02139

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

June 1988

[*]Research partially supported by an NSF Presidential Young Investigator Award.
[†]Supported by DARPA Contract N00014-87-K-825 and by ONR Contract N00014-86-K-0593. Part of the work was done while the author was at AT&T Bell Laboratories, Murray Hill, NJ 07974.

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.

Keywords: Bipartite matching, assignment problem, network flow, parallel algorithms.

1 Introduction

Bipartite matching and related problems are well studied in the contexts of both sequential (see *e.g.*, [ET75,HK73,Tar72]) and parallel (see *e.g.* [AA87,KUW86,MVV87]) computation. Though the latter research produced RNC algorithms, no sublinear-time deterministic parallel algorithms were known. In this paper we describe a number of 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 scaling, so the resulting algorithms run in sublinear time if the weights are polynomially bounded. The assignment problem with unary weights is known to be in RNC [KUW86], but no sublinear deterministic algorithms have been known previously.

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. We also show 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.

To address the above problems, we need the following notation and definitions. 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). 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. A *matching* is a set of edges such that each node in the graph has at most one edge in the matching incident to it. A *perfect* matching is a matching such that each node in the graph has exactly one edge in the matching incident to it, 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.

Our model of parallel computation is CRCW PRAM (concurrent read concurrent write parallel random access machine) [FW78]. Given a directed graph with n nodes and m arcs, $BFS(n,m)$ and $SSP(n,m)$ denote the maximum of $n+m$ and the number of processors required to find a breadth-first search tree and a single-source shortest-path tree in $O((\log n)^2)$ time, respectively. 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 $O(n^{2.5})$ [PR85].

In this paper we address the following problems.

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 $G - \Pi$. 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 $O(\sqrt{n}(\log n)^3)$ 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. The algorithm also solves a slight generalization of the maximal node-disjoint paths problem; this generalization is useful for constructing a depth-first search tree in an undirected graph.

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* iff for all non-tree edges (u,v), u and v lie on the same path starting at the root of the tree. All previous parallel algorithms for the problem use randomization [And87,AA87]. We show how to use our techniques to construct a deterministic algorithm for finding a depth-first search tree in $O(\sqrt{n}(\log n)^5)$ 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 $O(n^{2/3}(\log n)^3)$ time using $BFS(n,m)$ processors. Although the problem is known to be in RNC [KUW86,MVV87], the fastest previously known deterministic algorithm [GT88] runs in $O(n(\log n)^2)$ time.

Assignment problem (Also known as 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 $O(n^{2/3}(\log n)^3(\log nC))$ time if the edge weights are integers in the range $[-C,C]$. Under the as-

sumption that edge weights are given in unary, this problem is known to be in RNC [KUW86,MVV87]; our algorithm is sublinear under this assumption. The fastest previously known deterministic algorithm [GT88] runs in $O(n(\log n)^3 \log(nC))$ time.

Flows in zero-one networks We also study flows in networks with unit capacities. We study two versions of the problem, the maximum flow problem and the minimum-cost flow problem. Note that the bipartite matching problem is a special case of the first problem, and the assignment problem is a special case of the second problem. Our algorithm for the first problem 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 second problem runs in $O(m^{2/3}(\log n)^2 \log(nC))$ time using $SSP(n, m)$ processors.

The problems discussed above are important tools for design of efficient algorithms. 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. The ideas of this paper may lead to improved sequential algorithms as well.

The paper is organized as follows. In Section 2 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 3 we give parallel algorithms for maximum matching and zero-one flow problems; in Section 4 we extend the results to the weighted versions of these problems.

2 Maximal Node-Disjoint Paths

In this section we present 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 undirected graph. We describe the variation of the algorithm that works for undirected graphs. The extension to the directed case is straight-forward.

A natural approach to solve this problem is to find the paths one-by-one. The problem with this approach is a potentially large ($\Omega(n)$) number of paths, which leads to a superlinear running time. Another approach is to maintain the 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 a large number of paths at each iteration because of the interaction among the paths. Our algorithm combines these two approaches.

The algorithm *Maximal-Paths* solves a slight generalization of the node-disjoint paths problem: Given a set of node-disjoint paths connecting sources to intermediate nodes, find a set of node-

disjoint paths from the sources to the sinks such that for any node that is on an input path but on no output path, every path from this node to a sink intersects an output path. The node-disjoint paths problem corresponds to the case when each one of the input paths is a single node. The generalization of the problem is required for the depth-first search algorithm described below.

Figure 1 describes the *Maximal-Paths* algorithm. The algorithm maintains two sets of node-disjoint paths: *Active* paths and *Dead* paths. An active path starts at a source and ends at some intermediate node which is not a sink; a dead path connects a source to a sink. The initial set of active paths is the set of the input paths. 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 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 the status to active. If v' is one of the sinks, this path changes its status to dead. 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 sources 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 there exists a path P' from v_r to some sink t over idle nodes, and for all i, $r < i \leq k$, 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'.

The following lemma is sufficient to show correctness of the algorithm.

Lemma 2.1 *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

```
procedure Maximal-Paths(V, E, P_a);
    P_a   – the set of active paths;
    P_d   – the set of dead paths, connecting a sink to a source;
    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;
        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 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 a sink is reachable via edges in E';
        remove nodes that follow v_r from P, extend P to a sink;
    end;
end.
```

Figure 1: The *Maximal-Paths* procedure

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. ∎

Each iteration of the second phase 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 [SV82b], and $O(\log^2 n)$ time and $BFS(n,m)$ processors for the directed case [PR85]. The following lemma bounds the number of iterations in the first stage.

Lemma 2.2 *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 one. On the other hand, when a path is clipped, its old end-point changes the status from active to dead, again reducing Φ by one. 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 $\Phi \leq 3n$. ∎

Each iteration of the first stage can be implemented in $O(\log^3 n)$ time and with $O(m)$ processors, using the maximal matching algorithm of [IS86]. This leads to the following theorem.

Theorem 2.3

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 a maximal set of node-disjoint paths corresponds to a blocking flow in matching networks (described in detail in the next section). Thus, by using the *Maximal-Paths* procedure to find blocking flow at each iteration of Dinic's maximum-flow algorithm [Din70,ET75], we can compute 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 [Smi86,GB84], and recently Aggarwal and Anderson have found a *randomized* NC algorithm for it [AA87]. However, no deterministic parallel algorithm for the problem with a sublinear running time was known previously.

Although 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. Aggarwal and Anderson reduce this problem to the problem of finding a maximum matching. 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 path problem that is solved by the *Maximal-Paths* algorithm. Therefore, we have the following theorem.

Theorem 2.4 *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.*

3 Bipartite Matching and Zero-One Flows

In this section we describe sublinear-time parallel algorithms for for the bipartite matching problem and for the zero-one network flow problems (both with and without multiple arcs). For the purpose of this section, we assume familiarity with the maximum flow algorithms of [Gol85,GT86].

3.1 Bipartite Matching Algorithm

To solve a bipartite matching problem, we transform it into a zero-one network flow problem in a standard way (see *e.g.* [Law76]). 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 define all arc capacities to be one. The resulting maximum flow problem is equivalent to the original bipartite matching problem. We call a network that can be obtained by the above transformation a *matching network.*

Two possible approaches to 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 [FF62] with a parallel breadth-first search subroutine. Another approach is to use a parallel implementation of the Goldberg-Tarjan method [Gol85,GT86]. 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 Goldberg-Tarjan approach in the first stage and the Ford-Fulkerson approach in the second stage. A proper balancing of the two stages leads to a

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

Relabel(v).
Applicability: Any v.
Action: $d(v) \leftarrow \min\{d(w) + 1 | (v, w) \in E_f\}$.
(If this minimum is over an empty set, $d(v) \leftarrow \infty$.)

Figure 2: Push and relabel operations.

sublinear running time. In the context of sequential algorithms for the problem, similar balancing has been used in [AO87,ET75] to obtain $O(\sqrt{n}m)$ time bounds.

Before describing the algorithm, we need to introduce a few terms. For more detailed definitions, see [GT86]. A *pseudoflow* is a function on arcs of the network that obeys capacity constraints. 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. The *residual capacity* of an arc (v, w) with respect to a pseudoflow f is equal to 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 $G_f = (V, E_f)$. (E_f is the set of arcs in E with positive residual capacity.) A *flow* is a pseudoflow that obeys conservation constraints, in other words excesses at all nodes except the sink and the source are zero. 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). We say that a node $v \in S \bigcup T$ is *active* if $e_f(v) > 0$ and $d(v) \leq k$. (Our algorithm works correctly for any value of $k \leq n$, but the best running time bound is achieved for $k = n^{2/3}$.) Given a pseudoflow f and a distance labeling d, we define an *admissible* graph $G(f, d) = (V, E(f, d))$ by $E(f, d) = \{(v, w) \in E_f | d(v) = d(w) + 1\}$. We also assume familiarity with *push* and *relabel* operations described in Figure 2.

Figure 3 describes the bipartite matching algorithm. The algorithm consists of two phases. The first phase is a variation of the Goldberg-Tarjan method. This phase is executed as long as the number of active nodes is large, and, as a result, many distance labels increase at each (parallel) time step. When the number of active nodes is small, excesses from these nodes are returned to the source, and the second phase begins. In this phase, the algorithm finds augmenting paths from the source to the sink one-by-one. 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 algorithm, distance labels of source and 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 of one. After the above

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

initialization is complete, the *Match-and-Push* procedure is executed until the number of active nodes is less than l. (As we shall see, the best running time is achieved for $l = \lfloor \sqrt{nk} \rfloor$). 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 4, is the key to the first stage of our algorithm. The following lemma states the properties of this procedure that are essential for the analysis of the algorithm.

Lemma 3.1
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 excesses.*

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 the 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 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 3, every node in T has excess of either zero or one, and because of the relabeling done at Step 2 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 2 holds at the end of *Match-and-Push*. ▮

The above lemma implies that the last step of the first phase, namely returning flow from the nodes with excess to the source, is easy. More precisely, Invariants 1 and 2 imply that nodes in T have no excess flow, 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 by the previous part of the algorithm, and therefore for all $v \in S$, the residual capacities of arcs (v, s) are equal to one. Thus, the excess flow 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 our analysis of the algorithm by bounding the running time of the *Match-and-Push* procedure.

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 4: The *Match-and-Push* procedure

Lemma 3.2 *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 with $n + m$ processors [Gol87,SV82b]. The bottleneck is Step 1, which takes $O(\log^3 n)$ time on a CRCW PRAM using $n + m$ processors [IS86]. ∎

Next we bound the number of executions of *Match-and-Push.*

Lemma 3.3 *Procedure* Match-and-Push *is executed at most* $\frac{n(k+1)}{l} + 1$ *times.*

Proof: We call a relabeling of a node v *significant* if the relabeling increases $d(v)$ and before the relabeling $d(v) \leq k$. We show that at all except the last execution of *Match-and-Push*, there are at least l significant relabeling. Since the distance labels never decrease, the total number of significant relabelings is at most $n(k + 1)$, and the desired bound follows.

We claim that a significant 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 all except maybe the last execution of *Match-and-Push*, the number of nodes satisfying the above two conditions is at least l, so establishing this claim completes the proof of the theorem.

Suppose a node $w \in T$ is matched to a node $v \in S$ at Step 1; this implies that $d(v) \leq k$ and $d(w) = d(v) - 1 < k$. We show that $d(w)$ increases either at Step 3 or at Step 5. If $d(w)$ increases at Step 3, we are done. Otherwise, after Step 4, the only residual arc out of w is (w, v). 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; by the above assumption, 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 two.

Now consider an active node $v \in S$ that is not matched at Step 1. By definition of an active node, $d(v) \leq k$; we show that $d(v)$ increases at Step 6. Note that the arc (v, s) cannot be admissible, because $d(v) \leq k \leq n$. Therefore at Step 1 all admissible neighbors of v lie in T. These neighbors are matched at Step 1, 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. ∎

The above two lemmas, 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 3.4 *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 need the following lemma, which is similar to a lemma in [ET75].

Lemma 3.5 *After the first stage of the algorithm, the value of the residual flow is at most $n/k + l$.*

Proof: We show that the amount of flow that can reach the sink after the last execution of *Match-and-Push* is at most $n/k + l$. Since returning excesses to the source does not affect this amount, the above claim implies the theorem.

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. By the properties of matching networks [ET75], 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 $d(v)$. Thus, at most l paths in the first collection have length of k or less. The remaining paths have length of greater then k, and the number of such paths is at most n/k, because the paths are node-disjoint. ∎

In the second phase of the algorithm, we find augmenting paths one-by-one. By Lemma 3.5, we have the following bound on the running time of this stage.

Lemma 3.6 *The second stage of the bipartite matching algorithm runs in $O((\frac{n}{k} + l) \log^3 n)$ time using $BFS(n, m)$ processors.*

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

The running time bound for the algorithm is as follows:

Theorem 3.7 *The bipartite matching algorithm runs in $O(n^{2/3}\log^3 n)$ time using $BFS(n,m)$ processors.*

Proof: Set $k = n^{1/3}$ and $l = n^{2/3}$ and apply Lemmas 3.4 and 3.6. ∎

3.2 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. The bound achieved by the algorithm for general zero-one networks can also be obtained by transforming the input network into a matching network and applying the bipartite matching algorithm described above. However, the method we describe in this section leads to better 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 5). At a high level, this algorithm is similar to the algorithm of the previous section: the algorithm consists of two stages, the first based on the Goldberg-Tarjan method and the second based on the Ford-Fulkerson method. The balancing of work done in the two stages is similar to that of the sequential algorithms of [AO87,ET75]. In addition, the zero-one flow algorithm has a finish-up stage that converts a pseudoflow of maximum value into a flow of maximum value. (By a value of a pseudoflow we mean the amount flowing into the sink.)

The first stage initializes by setting distance label of s to n, setting all other distance labels to zero, setting flow through arcs of the form (s, v) to one, and flow through all other arcs to zero. After that, the *Push-and-Relabel* procedure is iterated until the total amount of excess at active nodes is less then $l = m^{2/3}$.

The *Push-and-Relabel* procedure is describe on Figure 6. The use of parallel prefix computations in steps 2 and 3 is similar to the use of these operations in [Gol87]. (The use of the parallel prefix computations in the design of parallel algorithms is discussed in [Ble86,LM86].)

The second stage of the algorithm keeps finding augmenting paths from a node $v \notin \{s, t\}$ with $e_f(v) > 0$ to the sink. One way to find such a path is to do a breadth-first search backwards from the sink in the residual graph. When no such paths exist, the current pseudoflow f_{end} is of maximum value.

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 capacity one 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 [Gol85,GT86]) that when the zero-one flow

```
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 5: 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, sort residual arcs (v, w) by the distance label of w.

Step 2. For all active nodes v, use a parallel prefix computation on the list of outgoing arcs to distribute $e_f(v)$ among residual neighbors of v, preferring residual neighbors with smaller distance labels.

Step 3. 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 2.

Step 4. Relabel all nodes $v \neq s, t$.

Figure 6: The *Push-and-Relabel* procedure.

algorithm is applied to the modified network, its second stage terminates with a flow (rather then a pseudoflow).

The correctness of the algorithm follows from [GT86]. Performance of the *Push-and-Relabel* procedure is summarized by the following lemma.

Lemma 3.8 *Procedure Push-and-Relabel runs in $O(\log n)$ time using m processors.*

Proof: Cole's sorting algorithm [Col86] implements Step 1 in the desired resource bounds. The bounds on steps 2,3, and 4 follow from [Gol87,SV82a]. ∎

The next lemma bounds the number of times *Push-and-Relabel* is applied.

Lemma 3.9 *Procedure Push-and-Relabel is executed at most $\frac{m(k+1)}{l}$ times.*

Proof: During the first stage of the algorithm, distance label of a node increases at most $k+1$ times. Since each push in a zero-one network is a saturating push, the total number of pushes is at most $m(k+1)$ [GT86]. At each execution of *Push-and-Relabel* there are at least l units of excess at active nodes, and therefore at least l pushes are performed. ∎

The above two lemmas, 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 3.10 *The first stage of the zero-one flow algorithm runs in $O(\frac{mk}{l}\log n)$ time using $n+m$ processors.*

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 3.3.

Lemma 3.11 *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: We show that the amount of flow that can reach the sink after the last execution of *Push-and-Relabel* is at most $m/k+l$. Since returning excesses to the source does not affect this amount, the above claim implies the theorem.

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.

We need to show that the number of paths in the first collection of paths is at most $l+m/k$. 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 $d(v)$. Thus at most l paths in the first collection have length of k or less. The remaining paths have length of greater then k, and the number of such paths is at most m/k, because the paths are arc-disjoint. ∎

Lemma 3.12 *The second stage of the zero-one flow algorithm runs in $O((\frac{m}{k}+l)\log^2 n)$ time using $BFS(n,m)$ processors.*

Proof: The lemma follows from Lemma 3.11. ∎

The running time bound for the algorithm on general zero-one networks is as follows:

Theorem 3.13 *On general zero-one networks, the zero-one flow algorithm runs in $O(m^{2/3}\log^2 n)$ time using $BFS(n,m)$ processors.*

Proof: Set $k = m^{1/3}$ and $l = m^{2/3}$. Lemmas 3.10 and 3.12 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 3.13 for dense graphs (more precisely, for $m > n^{3/2}$). The following lemma, similar to a lemma in [ET75], is a key to the improvement.

Lemma 3.14 *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 $\left(\frac{2n}{k-1}\right)^2 + l$.*

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

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 then 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 a set of these paths, and let $G' = (V, E')$ be a 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:

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

∎

Using lemmas 3.11 and 3.14, one can obtain the following theorem; the proof is similar to the proof of Theorem 3.13.

Theorem 3.15 *On zero-one capacity networks with no multiple arcs, the maximum flow algorithm runs in time* $O(\min(m^{2/3}, (nm)^{2/5}) \log^2 n)$ *on a CRCW PRAM using* $BFS(n,m)$ *processors.*

4 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 zero-one capacities. For the purpose of this section, we assume familiarity with the minimum-cost flow framework of [Gol87,GT87a,GT87b].

4.1 The Assignment Problem

The assignment problem is a weighted version of the bipartite matching problem. Similarly to the unweighted case described in Section 3.1, we transform the assignment problem into the zero-one minimum-cost flow problem in the standard way (see *e.g.*, [Law76]) where the weights on the edges are mapped into costs on the arcs of the transformed network. 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.

In order to describe the algorithm, we need to introduce a few definitions (see [Gol87,GT87a] 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. Define $C = \max\{|c(v,w)| \; : \; (v,w) \in E\}$. We say that a pseudoflow is *ϵ-optimal* if 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,

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

    [Initialization]
    transform the input problem into network flow form;
    V' := S ⋃ T ⋃ {s} ⋃ {t};
    E' := E ⋃ ({s} × S) ⋃ (T × {t});

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

Figure 7: High-level description of the outer (scaling) loop of the assignment algorithm.

i.e., if $u_f(v,w) > 0$ and $c_p(v,w) < 0$. 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-weight bipartite matching algorithm, shown in Figure 7, does generalized cost-scaling [Gol87,GT87a]. Initially $\epsilon = C$. The algorithm iteratively halves ϵ and uses the *Refine* procedure to update the flow to be ϵ-optimal again. It can be shown [Ber86] that ϵ-optimal flow is optimal for $\epsilon < 1/n$, and therefore we have the following lemma.

Lemma 4.1 [Gol87] *The algorithm terminates and produces an optimal flow after* $O(\log nC)$ *calls to the* Refine *procedure.*

The heart of the algorithm is the *Refine* procedure, shown in Figure 8, that converts an 2ϵ-optimal pseudoflow 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 4.3.) 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 by procedure *Refine*, that consists of two stages. The first stage iteratively uses the *Match-and-Push* procedure (see Figure 4) to push the positive excesses towards the negative ones. The *Match-and-Push* procedure used during this

```
procedure Refine(V', E', c, p, 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 ∈ V' do e_f(v) := 0;
    for all (v, w) ∈ E' do f(v, w) := 0;
    for all {(v, w) | (v, w) ∈ E' and c_p(v, w) < −ε} do
        f(v, w) := 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(v, w) + ε;
    while there are active nodes do begin
        let Γ be a shortest path w.r.t. length from an active node to t;
        augment along Γ;
    end;

    return (c, p, f);

end.
```

Figure 8: High-level description of the inner loop of the assignment algorithm. For the algorithm described in this paper, we take $l = n^{2/3}$.

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 $\delta = \min(e_f(v), u_f(v, w))$ units of flow from v to w as follows:
$f(v, w) \leftarrow f(v, w) + \delta$; $f(w, v) \leftarrow f(w, v) - \delta$;
$e_f(v) \leftarrow e_f(v) - \delta$; $e_f(w) \leftarrow e_f(w) + \delta$.

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

stage is exactly the same as for the unweighted case (see Figure 4), except that *Push* and *Relabel* are generalized to the weighted case as described in Figure 9. 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 $k\epsilon$. The first stage is executed as long as the number of active nodes exceeds l. (The best running time is achieved for $l = n^{2/3}$ and $k = n^{1/3}$.)

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 l obtained by adding ϵ to costs. Since we have assumed that the input graph has a perfect matching, *Refine* terminates with a flow. The following lemma shows that the second stage preserves ϵ-optimality.

Lemma 4.2 *Suppose a pseudoflow f is ϵ-optimal, and let $l : E \rightarrow R$ be defined by $l(v, w) = c(v, w) + \epsilon$. Let Γ be a shortest path with respect to l in G_f. Than changing flow along Γ preserves ϵ-optimality.*

Proof: Straight-forward, given the results of [GT87a]. ∎

The following lemma is needed to bound the residual flow after the first stage.

Lemma 4.3 *After the first stage of* Refine, *the value of the residual flow is at most $O(n/k + l)$.*

Proof: Consider nodes other then s and t that have excesses at the end of the first stage. We show that 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 then $k\epsilon$ during the first stage. Each excess has a value of one, and therefore the total amount of excess at the end of the stage is $O(n/k + l)$. This fact gives the desired bound on the value of the residual flow.

The first step decreases the prices of all the nodes in T by 2ϵ. This increases the reduced prices 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 cost. Hence the number of remaining residual arcs with negative reduced cost is at most n. On the other hand, the residual 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.

Without loss of generality, we can assume that at this point all costs are replaced by the residual costs, and all prices are set to zero. We call the resulting costs *transformed.*

Next we give lower and upper bounds on the cost of an ϵ-optimal flow in a network with transformed costs. The transformed arc costs are at least -4ϵ. Moreover, there are at most n negative-cost arcs. Therefore, for any pseudoflow f in this network, we have

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

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. For any saturated arc (v, w) such that $f_\epsilon(v, w) = 1$, there is a residual arc $(w, v) \in E'_{f_\epsilon}$ with cost $c(w, v) = -c(v, w) \geq -\epsilon$, and therefore we have

$$\text{cost}(f_\epsilon) \leq n\epsilon.$$

Consider a pseudoflow f'_ϵ at some point of the execution of the first stage. 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, simple paths from s to t, and simple cycles. Note that $A' \subseteq E_{f'_\epsilon}$, where $E_{f'_\epsilon}$ is the set of residual arcs of the flow f'_ϵ, and therefore the residual cost of any arc in A' is at least $-\epsilon$. Let $|P|$ denote the length of a path P. Then the cost of a path P from node v with excess to node w with deficit is

$$\text{cost}(P) = \sum_{e \in P} c(e) = \sum_{e \in P} c_p(e) + p(w) - p(v) \geq -p(v) - |P|\epsilon$$

The last inequality holds because nodes with deficit are not relabeled. By the properties of matching networks, the sum of the lengths of the paths and the cycles is at most n. The cost of any pseudoflow is at least $-4n\epsilon$ and the cost of f_ϵ is at most $n\epsilon$. Hence, the sum of the costs of the paths is at most $5n\epsilon$. Taking into account that the cost of a cycle is at least the length of the cycle times $-\epsilon$, we have

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

Hence, the number of nodes that were relabeled by $k\epsilon$ is at most $6n/k$, and the bound follows. ∎

The above lemma bounds the number of iterations of the second stage of *Refine*. Next we bound the number of iterations of the first stage.

Lemma 4.4 *There are at most* $\frac{n(k+1)}{l} + l$ *calls to* Match-and-Push *in the first stage of* Refine.

Proof: The proof is similar to the proof of Lemma 3.3. 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 claim then follows because the total amount of relabeling during the first stage of *Refine* is at most $nk\epsilon$.

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 9), w is relabeled by

$$\Delta p(w) = p(w) - c(w, v) - p(v) + \epsilon \geq (p(v) + c(w, v)) - p(v) + c(v, w) = \epsilon.$$

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 that is not matched during the first step. 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 4.5 *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 4.4, the first stage makes $O(\frac{nk}{l} + l)$ calls to *Match-and-Push*. By Lemma 3.2, *Match-and-Push* runs in $O(\log^3 n)$ time on a CRCW PRAM with $n + m$ processors. Hence, by Lemma 4.3, each iteration of *Refine* takes $O(n^{2/3} \log^3 n)$ time on a CRCW PRAM using $SSP(n, m)$ processors.

By Lemma 4.1, after $O(\log(nC))$ iterations of *Refine* the resulting flow is optimal. The claim follows by setting $k = n^{1/3}, l = n^{2/3}$. ∎

4.2 Zero-One Minimum-Cost Flows

In this section we study the minimum-cost flow problems with unit capacities on arcs. To solve such a problem $((V, E), c, s, t)$, we use a classical transformation (see *e.g.* [CSV84,KUW86]) to reduce this problem to a problem of finding a minimum-weight bipartite matching. The transformation is as follows. First, 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 in the line-graph G' of G. Split each node v in G' into two nodes v_1 and v_2, connect them by a zero-weight edge, and connect v_2 to u_1 for each arc (v,u) in G'. Call the resulting graph G''. It is easy to see that minimum-weight bipartite 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 the following theorem:

Theorem 4.6 *The Minimum-Cost Flow algorithm for general zero-one networks runs in time* $O(m^{2/3}\log^3 n \log(nC))$ *on a CRCW PRAM using* $SSP(n,m)$ *processors.*

Proof: The theorem follows from the observation $|V''| = 2m$ and Theorem 4.5. ∎

Remark: We can extend the result of the theorem 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 $c(v,w)$.

The ideas of the previous section can be extended to get a sublinear-time algorithm that finds a maximum flow of minimum cost in graphs with zero-one capacities without using the above reduction. The resulting algorithm runs in $O(m^{2/3}\log^2 n \log(nC))$ time, *i.e.*, improves the running time bound of Theorem 4.6 by a factor of $\log n$. However, we are unable to obtain better bounds, suggested by Theorem 3.15, for the case of graphs with no multiple arcs. Since the algorithm is obtained by a straight-forward combination of the results of this section and Section 3.2, we omit the details.

Remark: Note that our algorithms find an optimal primal solution but not the 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.* [GT88].

Acknowledgments

We would like to thank Alok Aggarwal, 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

[AA87] A. Aggarwal and R. J. Anderson. A random NC algorithm for depth first search. In *Proc. 19th ACM Symposium on Theory of Computing*, pages 325–334, 1987.

[And87] R. J. Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 7, 1987.

[AO87] R. K. Ahuja and J. B. Orlin. *New Distance-Directed Algorithms for Maximum-Flow and Parametric Maximum-Flow Problems.* Technical Report 1908-87, Sloan School of Management, M.I.T., 1987.

[Ber86] 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).

[Ble86] G. Blelloch. *Parallel Prefix vs. Concurrent Memory Access.* Technical Report, Thinking Machines, Inc., 1986.

[Col86] R. Cole. Parallel merge sort. In *Proc. 27th IEEE Annual Symposium on Foundations of Computer Science*, pages 511–516, 1986.

[CSV84] A. K. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, May 1984.

[Din70] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[ET75] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ., 1962.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

[GB84] R. K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24:134–150, 1984.

[Gol85] A. V. Goldberg. *A New Max-Flow Algorithm.* Technical Report MIT/LCS/ TM-291, Laboratory for Computer Science, M.I.T., 1985.

[Gol87] 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).

[GT86] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 136–146, 1986. (To appear in J. Assoc. Comput. Mach.).

[GT87a] A. V. Goldberg and R. E. Tarjan. *Finding Minimum-Cost Circulations by Successive Approximation.* Technical Report MIT/LCS/TM-333, Laboratory for Computer Science, M.I.T., 1987. Also available as Technical Report CS-TR 106-87, Department of Computer Science, Princeton University.

[GT87b] A. V. Goldberg and R. E. Tarjan. Solving minimum-cost flow problems by successive approximation. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 7–18, 1987.

[GT88] H. N. Gabow and R. E. Tarjan. Almost-optimal speed-ups of algorithms for matching and related problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 514–527, 1988.

[HK73] 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.

[IS86] A. Israeli and Y. Shiloach. An improved parallel algorithm for maximal matching. *Information Proc. Lett.*, 22:57–60, 1986.

[KUW86] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a maximum matching is in random NC. *Combinatorica*, 6:35–48, 1986.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[LM86] C. Leiserson and B. Maggs. Communication-efficient parallel graph algorithms. In *Proc. of International Conference on Parallel Processing*, pages 861–868, 1986.

[MVV87] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 345–354, 1987.

[PR85] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proc. 17th ACM Symposium on Theory of Computing*, pages 143–152, 1985.

[Smi86] J. Smith. Parallel algorithms for depth-first searches I. Planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

[SV82a] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[SV82b] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3:128–146, 1982.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput*, 1:146–160, 1972.