JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 6, 20–38 (1989)

Parallel Algorithms for Maximum Bipartite Matchings and Maximum 0–1 Flows

BARUCH SCHIEBER* AND SHLOMO MORAN

Department of Computer Science, Technion–Israel Institute of Technology, Haifa, Israel 32000

Received May 3, 1985

Two parallel algorithms for finding maximum bipartite matchings on a CRCW PRAM are presented. The first algorithm finds a maximum bipartite matching in $O(n \log n)$ time using m processors, where n is the number of vertices and m is the number of edges in the bipartite graph. The second algorithm does the same job in $O(n)$ time using nm processors. Simple modifications of these algorithms induce parallel algorithms for finding maximum 0–1 flows, which are also presented. © 1989 Academic Press, Inc.

1. INTRODUCTION

In this paper we present efficient parallel algorithms for finding maximum bipartite matchings and maximum 0–1 flows.

The input for the maximum bipartite matching problem is a connected bipartite graph $G = (V, E)$ (with no parallel edges), where V (the set of the vertices) can be partitioned into two disjoint subsets X and Y, such that E (the set of the edges) satisfies $E \subseteq X \times Y$. (Denote $n = |V|$, $m = |E|$, $n_x = |X|$, $n_y = |Y|$.) A matching M in G is a subset of edges, no two of which have a common vertex.

The *maximum bipartite matching* problem is: Given G, find in it a matching of maximum possible cardinality.

The input for the maximum 0–1 flow problem is a directed multigraph $G = (V, E)$ (called a *network*) and two specified vertices s (source) and t (sink) in V. (Denote: $n = |V|$ and $m = |E|$.) A 0–1 *flow function* f is an assignment of a number $f(e)$ to each edge $e \in E$, such that: (1) $f(e) \in \{0, 1\}$. (2) For every $v \in V - \{s, t\}$: $\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e)$, where $\alpha(v)$ (resp. $\beta(v)$) is the

* Present affiliation: Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel 69978.

0743-7315/89 $3.00

Copyright © 1989 by Academic Press, Inc.
All rights of reproduction in any form reserved.

set of incoming (resp. outgoing) edges of v. The *flow value* $|f|$ equals $\sum_{e\in\alpha(t)} f(e) - \sum_{e\in\beta(t)} f(e)$.

The *maximum 0–1 flow problem* is: Given a network $G = (V, E)$ and two vertices $s, t \in V$, find a 0–1 flow function with maximum flow value.

The models of parallel computation used in this paper are all members of the parallel random access machine (PRAM) family. A PRAM employs p synchronous processors all having access to a common memory. An exclusive-read exclusive-write (EREW) PRAM does not allow simultaneous access by more than one processor to the same memory location for either read or write purposes. A concurrent-read concurrent-write (CRCW) PRAM allows simultaneous access by more than one processor to the same memory location for both read and write purposes. We use the following concurrent write convention. If several processors attempt to write simultaneously at the same memory location then one of them succeeds but we do not know in advance which one. Vishkin [17] gives an efficient simulation of a CRCW PRAM on an EREW PRAM. From this simulation it follows that given a parallel algorithm which runs $t(n)$ time using $p(n)$ processors on a CRCW PRAM, with the property that for each memory location it is predetermined which processors may access it at each time unit, we can run it on an EREW PRAM in $O(t(n)\log w)$ time using the same number of processors, where w is the maximum number of processors which may access the same memory location simultaneously during the execution on the CRCW PRAM.

Two parallel algorithms for finding maximum bipartite matchings on a CRCW PRAM are presented. The first algorithm finds a maximum bipartite matching in $O(n \log n)$ time using m processors. The second algorithm does the same job in $O(n)$ time using nm processors. (*Remark.* The big Oh is always omitted from the processors bound as we can simulate any algorithm which uses $O(p)$ processors using exactly p processors by increasing the running only by a constant factor.)

By slight modifications to our algorithms we get efficient parallel algorithms for finding maximum 0–1 flows on a CRCW PRAM. The complexity of these algorithms depends on the type of the input network and is summarized in Table I.

The problem of finding maximum bipartite matchings is very important and interesting. Moreover, many combinatorial problems can be reduced to this problem, or can be solved using the solution to this problem as a subroutine. The best-known sequential algorithm for this problem is due to Hopcroft and Karp [7] and it has a time complexity of $O(\sqrt{n}m)$. An algorithm of the same complexity that solves the problem for general graphs is given in [11]. Even and Tarjan [4] noted that the maximum bipartite matching problem can be reduced to the problem of finding maximum 0–1 flows in a certain type of networks. These networks have either one incoming edge or one outgoing edge per node. Using this reduction they derived an algorithm for

TABLE I

Network type	No. of processors	Time complexity
General networks	m	$O(m \log n)$
	nm	$O(m)$
Networks without parallel edges	m	$O(\text{Min}\{n^{1.5}, m \log n\})$
Networks with one incoming or one outgoing edge per node	m	$O(n \log n)$
	nm	$O(n)$

the maximum bipartite matching problem with the same time complexity as in [7].

The only previously known deterministic parallel algorithms for the maximum bipartite matching problem are applications of the parallel algorithms for maximum flows given in Refs. [14, 6]. These algorithms have time complexity of $O(n^{1.5}\log n)$ using $O(m/n)$ processors. Our parallel algorithms for the same problem are faster than these algorithms. Our parallel algorithms for finding maximum 0–1 flows are also superior in their running time to those obtained by specializing the parallel algorithms in [14, 6] to the problem of finding maximum 0–1 flows.

Many parallel algorithms which deal with related problems appear in the literature. In this category we can mention Ref. [9], where a randomized algorithm for finding a maximum matching in general graphs in polylog time, using a polynomial (though quite large) number of processors, is presented. Recently, this algorithm was improved by Galil and Pan [5]. More recently, a simple randomized algorithm for finding maximum matchings was discovered by Mulmuley *et al.* [12]. This algorithm has expected time of $O(\log^2 n)$ using $mn^{3.5}$ processors. Another related parallel algorithm is the parallel algorithm for finding a *maximal* matching in general graphs given in [8]. This algorithm runs in polylog time using m processors. (A matching is *maximal* if it is not contained in any larger matching.)

Some of our algorithms can also be implemented in an asynchronous communication network, using a standard synchronizer described in [2]. The resulting distributed algorithms are superior to previously known distributed algorithms for these problems.

The rest of the paper proceeds as follows. In Section 2 the outline of the maximum bipartite matching algorithms is given. Sections 3 and 4 are devoted to a full description of these algorithms, and in Section 5 the algorithms for maximum 0–1 flows are discussed.

Postscript. Recently, we found, in [13], efficient parallel and distributed algorithms for maximum matchings in *general* graphs. The parallel algorithm for maximum matchings in general graphs runs in $O(n \log n)$ time using m processors.

2. Outline of the Maximum Bipartite Matching Algorithms

We use the following basic definitions relative to a matching M:

An edge in M is a *matching edge.* Every edge not in M is a *free edge.* A vertex is *matched* if it is incident to a matching edge and *free* otherwise. An *alternating path* is a path whose edges are alternately matching and free. The length of an alternating path P, denoted by $|P|$, is the number of edges in it. An alternating path is *augmenting* if it is simple and both of its ends are free vertices. If P is an augmenting path we can augment the matching size by simply interchanging the matching and free edges along this path.

The algorithms presented here, like most other known algorithms for maximum matchings, are based on finding augmenting paths. The reason for this is the following theorem.

THEOREM 2.1 [7]. *Let M be a matching and M^* a maximum matching. Then M has a set of $|M^*| - |M|$ vertex disjoint augmenting paths.*

Theorem 2.1 guarantees that when no augmenting path exists the matching is of maximum cardinality. The basic difference between the algorithms in [7, 4, 14] and ours is that at each phase of our algorithm only one augmenting path is found, and not a maximal set of such paths. The resulting increase in the number of phases is compensated by careful analysis. In this analysis we use the fact that, at the present state of the art (i.e., using the known parallel algorithms), a search for a single augmenting path can be done in parallel faster than a search for a maximal set of such paths. (Note that this is not true in the sequential case.)

At the heart of both algorithms there is a search and update routine which is iterated $k = \text{Min}(n_x, n_y)$ times. (Recall that $n_x = |X|$ and $n_y = |Y|$, where X and Y are two disjoint sets such that $V = X \cup Y$ and $E \subseteq X \times Y$.) In iteration i of this routine an augmenting path whose length is at most $l_i = [2|M_{i-1}|/(k - i + 1)] + 1$ is sought, where M_{i-1} is the matching given at the beginning of iteration i. If an augmenting path is found then it is used to update the matching. Otherwise we are guaranteed that no augmenting path of length $\leq l_i$ exists, and the matching is not changed.

The validity of the algorithms stems from the following theorem.

THEOREM 2.2. *Let M_i denote the matching after the execution of iteration i (where M_0 denotes the empty matching), and let M^* be the maximum matching; then $|M^*| - |M_i| \leq k - i$.*

Proof. By induction.

Base. $i = 0$—obvious as $|M^*| \leq k$.

The inductive step. Let us assume that the theorem is true for $i - 1$ and prove it for i. We must examine two cases.

Case 1. An augmenting path was found in iteration i.

We get $|M^*| - |M_i| = |M^*| - (|M_{i-1}| + 1) \leqslant k - (i-1) - 1 = k - i$.

Case 2. No augmenting path was found in iteration i.

Assume for the contradiction that $|M^*| - |M_i| > k - i$. By Theorem 2.1 this implies that at the end of iteration i there are at least $k - i + 1$ vertex disjoint augmenting paths. In each augmenting path of length l exactly $(l-1)/2$ of the edges are matching edges and so at least one augmenting path P must satisfy the inequality

$$\frac{|P| - 1}{2} \leqslant \frac{|M_i|}{k - i + 1}.$$

But since the matching was not changed in iteration i, in the beginning of that iteration the same path satisfied

$$|P| \leqslant \frac{2|M_{i-1}|}{k - i + 1} + 1 = l_i,$$

contradicting the fact that no such path existed. ■

The above theorem guarantees that after $\text{Min}(n_x, n_y)$ iterations the computed matching is the maximum matching.

The Time Analysis

The total time complexity of the algorithms depends on the complexity of each iteration in a way shown in the next theorem.

THEOREM 2.3. *Let the time complexity of each iteration i be bounded by $O(l_i^\alpha)$.*

(i) *If $0 \leqslant \alpha < 1$, then the time complexity of the whole algorithm is $O(n)$.*

(ii) *If $\alpha = 1$ then the time complexity of the algorithm is $O(n \log n)$.*

Proof. The theorem is proved simply by summing up the times needed for each iteration,

$$\text{(i)} \quad \sum_{i=1}^{k} \left\lfloor \frac{2|M_{i-1}|}{k - i + 1} + 1 \right\rfloor^\alpha \leqslant \int_{x=0}^{k-1} \left(\frac{3k}{k - x}\right)^\alpha dx$$

$$= (3k)^\alpha \left[\frac{1}{1 - \alpha}(k^{1-\alpha} - 1)\right] \leqslant \frac{1}{1 - \alpha} 3^\alpha k = O(n),$$

which proves this part of the theorem.

Note that the constant factor in the bound above is proportional to $1/(1-\alpha)$ and so for small values of α we will get better performance.

To see that this bound is tight, observe that when $n_x = n_y = n/2$ we have

$$\sum_{i=1}^{k} \left\lfloor \frac{2|M_{i-1}|}{k-i+1} + 1 \right\rfloor^{\alpha} \geqslant \sum_{i=1}^{k} \left\lfloor \frac{2|M_{i-1}|}{k-i+1} + 1 \right\rfloor^{0} = \sum_{i=1}^{n/2} 1 = n/2$$

(ii) $$\sum_{i=1}^{k} \left\lfloor \frac{2|M_{i-1}|}{k-i+1} + 1 \right\rfloor \leqslant k + \int_{x=0}^{k-1} \frac{2|M^*|}{k-x}\, dx$$

$$= k + 2|M^*| \log k = O(n \log n).$$

Again, to see that this bound is tight, note that when $n_x = n_y = n/2$ and G contains a perfect matching (i.e., a matching of cardinality $n/2$) then the summation gives

$$\sum_{i=1}^{k} \left\lfloor \frac{2|M_{i-1}|}{k-i+1} + 1 \right\rfloor = \sum_{i=1}^{n/2} \left\lfloor \frac{2(i-1)}{n/2-i+1} + 1 \right\rfloor$$

$$= \sum_{i=1}^{n/2} \left\lfloor \frac{n}{n/2-i+1} - 1 \right\rfloor = \Omega(n \log n). \quad \blacksquare$$

Observe that the number of iterations in the algorithms is linear in n, and so we cannot improve the linear bound using the described method, no matter how fast we search for an augmenting path.

In certain cases the algorithms can be improved a little if a minimum-length augmenting path will be searched in each iteration. Using this improvement and after summing the iterations' complexities more carefully, the bounds we get are $O(|M^*|\log|M^*|)$ and $O(|M^*|)$ instead of $O(n \log n)$ and $O(n)$, respectively. These bounds are the same as the previous ones in the worst case but are better on the average.

In the following sections we describe two algorithms which achieve the time complexities mentioned in Theorem 2.3. The first algorithm is improved to get the $O(|M^*|\log|M^*|)$ time bound using the same number of processors, by implementing the idea mentioned in the previous paragraph. We were not able to implement the same idea on the second algorithm without increasing the number of processors considerably. The description contains mainly the search and update routine which is invoked in each iteration of the algorithms.

3. ALGORITHM 1

The first algorithm can be implemented using exactly m processors, where each processor represents an edge. The search for an augmenting path of

minimum length will be done using a BFS-like scan, while the updating will be done by backtracking through the tree built during the scan.

3.1. *The BFS Scan*

Scanning begins from every free vertex in X, and it is stopped when an augmenting path is found (or after no such path of length $\leqslant l_i$ is found). During the scan the vertices are labeled; the label of a vertex u denotes the length of a shortest alternating path connecting one of the free vertices in X to u. For each vertex we also keep a pointer to the preceding vertex in one of these shortest alternating paths. We start by labeling all the free vertices belonging to the subset X by zero.

At step t all the processors representing as yet unscanned *free* edges emanating from vertices labeled by $2t$ scan the ends of their related edges. (If there are no vertices labeled by $2t$ then the scanning is stopped as no augmenting path of length $\leqslant l_i$ exists.) Every unlabeled end of these edges is labeled by $2t + 1$, and its pointer is initialized to point to the vertex at the other end of the edge (which is labeled $2t$). If one of the newly labeled vertices is a free vertex then an augmenting path is found; else, if $2t + 1 \geqslant l_i$, no augmenting path of length $\leqslant l_i$ exists. In both of these cases the scanning is stopped. Otherwise, all the *matching* edges emanating from the vertices labeled by $2t + 1$ are scanned in a similar way (these edges are as yet unscanned), and their unlabeled ends are labeled by $2t + 2$. This completes the execution of step t, and the execution of step $t + 1$ is started.

3.2. *The Updating*

If an augmenting path was found, the matching has to be updated. The updating is done using the pointers initialized in the scanning process. Starting from a selected non-zero-labeled free vertex we backtrack to the free vertex at the beginning of the path, interchanging the role of each edge in our way. The selection is done using n processors in $O(\log n)$ time (without write conflicts). Note that a single processor may accomplish the backtracking within the stated time bound—a fact that will be used in the analysis of the next algorithm.

3.3. *The Validity and Complexity of the Algorithm*

It can be easily verified that the parallel labeling process really simulates BFS-scan, starting from all the free vertices in X and traversing the free and matching edges alternately. This implies that the label of each vertex u denotes the minimal length of an alternating path from some free vertex in X to u. One of the alternating paths to a free vertex in Y achieving that length can be reconstructed using the pointers.

The complexity computation is also trivial. Each step can be executed in constant time and so the time needed for finding an augmenting path is linear in its length. Using Theorem 2.3 and the fact that the path selection in the

updating step takes $O(\log n)$ time per iteration, we find that the overall time complexity of the algorithm is $O(n \log n)$. The space complexity of the algorithm is easily shown to be $O(m)$. The number of utilized processors is m, as claimed. To improve the time bound to $O(|M^*| \log |M^*|)$, we modify the BFS-scan stage so that the scanning is terminated only if an augmenting path of minimal length is found, or if $|M_{i-1}| + 1$ steps are completed (in iteration i). In this latter case the whole algorithm is terminated, as no augmenting path exists.

Implementation Remark. We assume that the input graph is given by its adjacency lists and that for each vertex we can decide in constant time whether it belongs to X or to Y. To justify this assumption we show how to obtain this input format from an arbitrarily ordered list of the input graph edges within the stated time and processors bounds. (1) Sort the list of edges. This can be done in $O(\log n)$ time with m processors using the parallel sorting algorithm given in [1]. As a result of the sorting we obtain the adjacency lists of the input graph. (2) Compute a (rooted) spanning tree of the input graph. (Recall that the input graph is connected.) This also can be done in $O(\log n)$ time with m processors using the parallel connectivity algorithm given in [15]. (3) For each vertex decide whether it belongs to X or to Y according to its distance from the root of the spanning tree. This can be done using a variation of the list ranking algorithm given in [18] in $O(\log n)$ time with m processors.

The maximum number of processors attempting to read or write in the same location simultaneously in the above algorithm is at most d, where d is the maximum degree of the graph. This situation occurs in the scanning part of the algorithm where some processors attempt to read or to label the same vertex. This fact implies that when the same algorithm is implemented on an EREW PRAM model its time complexity is increased by a factor of $\log d$. In the version where at each phase an augmenting path of minimal length is searched, the maximum number of simultaneous accesses to the same location increases to $O(m)$, as at the end of each step of the BFS-scan all the processors must check whether the scan has terminated already. This increases the time by a factor of $\log n$. However, we can implement this version of the algorithm more efficiently on an EREW PRAM as follows. We set up a binary tree of logarithmic height whose vertices correspond to the m processors. This tree will be used to broadcast the termination message to all the processors. When an augmenting path is found by a processor it sends a message to the root of the tree. This message reaches the root in $O(\log n)$ time. Upon receipt of such a message the root sends termination messages along the tree to all the m processors. This also takes $O(\log n)$ time. Note that some processors will run for $O(\log n)$ steps longer than absolutely necessary; however, this does not affect the validity of the algorithm. This modified version of the algorithm runs in $O(|M^*|(\log n + \log |M^*| \log d))$ time on an EREW PRAM.

4. ALGORITHM 2

The second algorithm has a time complexity of $O(n)$ achieved by using a search and update routine which runs in $O(\sqrt{l_i})$ time in iteration i. Recall that, by Theorem 2.3, the $O(n)$ running time of the algorithm will follow by any algorithm that performs iteration i in $O(l_i^\alpha)$ time for any $0 \leqslant \alpha < 1$. We choose $\alpha = \frac{1}{2}$ since it provides the minimal running time within the stated number of processors. Each iteration of our algorithm consists of two parts—the searching part and the updating part, described below.

4.1. *The Searching*

The search for a path of length at most l is done in two phases.

Phase 1. Find all the pairs of nodes $[x, y]$ satisfying: (a) $x \in X$ and $y \in Y$. (b) There is a simple alternating path from x to y, starting with a free edge, whose length is at most l_1, where l_1 is the least odd number satisfying $l_1 \geqslant \sqrt{l}$.

Each pair $[x, y]$ found at this phase is recorded together with the length of its corresponding alternating path. This corresponding alternating path is also recorded, as a list of its vertices in the order of their appearance in the path. Obviously the size of each list is bounded by $\lceil \sqrt{n} \rceil + 1$.

Implementation. The search is accomplished by activating n_x copies of a parallel BFS-scan algorithm similar to the one described in the previous section. This algorithm differs from the one described previously, in that each copy initially labels only one vertex in X with a zero label, rather than labeling all free vertices in X. The paths are stored by activating $O(n_x n_y)$ copies of the previously described backtracking routine, one copy for each pair. (Recall that this routine uses only one processor.) The time needed for the search and the path recording is proportional to $\sqrt{l}$ and the number of processors needed is nm (nm processors for the search and $n^2 \leqslant nm$ for the recording).

Phase 2. Define a new graph $G' = (V, E')$ where

$$E' = M \cup \{(x, y) \mid x \in X, \; y \in Y \text{ a path from } x \text{ to } y \text{ was found in Phase } 1\}.$$

Obviously the graph G' is bipartite and the edges in M still form a matching in it. Figure 4.1b shows the graph G' defined from the graph G given in Fig. 4.1a. (Note that G' may have parallel edges. However, an edge may occur at most twice, once as a matching edge and once as a free edge.) Search G' for an augmenting path P' which satisfies the following two conditions. (1) Its length is at most $l' = 2\lceil (l - l_1)/(l_1 + 1) \rceil + 1$. (2) The l'th free edge in P' (if

FIG. 4.1. (a, b) Wiggly edges are matching edges. $l = 5$ $l_1 = \lceil \sqrt{l} \rceil = 3$.

such exists) corresponds to a path, found in Phase 1, whose length is at most l_2, where l_2 is the remainder of $l/(l_1 + 1)$. (Note that $\lfloor l/(l_1 + 1) \rfloor = \lceil (l - l_1)/(l_1 + 1) \rceil = (l' - 1)/2$.) Each such augmenting path P' corresponds to an augmenting path of the desired length in the original graph, and vice versa, as will be proved later. Assuming this, all that remains is to update the matching as described in the next subsection.

Implementation. The construction of G' can be done in constant time using $n^2 \leqslant nm$ processors. The search for an augmenting path can be done using the search routine of the previous algorithm with one modification. When we add the l'th free edge to the augmenting path we have to check whether it corresponds to a path whose length is at most l_2. The search routine runs in $O(\sqrt{l})$ time using n^2 processors, as E' contains $O(n^2)$ edges.

The correspondence between augmenting paths in G and G' is proved by the following lemma.

LEMMA 4.1. *G' contains an augmenting path of length at most l'* iff *G contains an augmenting path whose length is at most l.*

Proof. We will prove only one direction, as the other direction can be proved similarly.

Each free edge (x, y) in an augmenting path P' found in G', where $x \in X$ and $y \in Y$, can be replaced by an alternating path from x to y in G, that starts (and ends) with a free edge. The length of this path is at most l_1 if (x, y) is not the l'th edge in P' and at most l_2 otherwise. Each matching edge in G' is also a matching edge in G. It follows that to each augmenting path P' of length l' in G' there corresponds an alternating path P in G whose length is not larger than $l_1(l' - 1)/2 + (l' - 1)/2 + l_2 = l$ and both its ends are free vertices. If P is simple then we are done. Otherwise P contains some cycles, all of which must be of even length (since G is bipartite). This implies that the simple path obtained by removing cycles from P is a simple alternating path with both ends free, and hence is the desired augmenting path.

4.2. *The Updating*

If an augmenting path was found in G', the corresponding augmenting path in G is constructed from it by deleting cycles, as described in the proof of

Lemma 4.1. All we must show is that this can be done within the stated time and processor bounds.

Assume that an augmenting path P' was found in G' (note that we select only one such path and we do it in constant time, permitting write conflicts). This path corresponds to an alternating path P in G, where P can be partitioned into several simple subpaths, each of which corresponds to a certain free edge in P'. The ith subpath will be denoted by P_i. We remove the even cycles from P as follows.

Step 1. Allocate a processor to each occurrence of a vertex in P. This can be done in parallel in constant time using the lists of edges corresponding to every subpath P_i as follows. We allocate $\sqrt{n}$ processors to each subpath P_i. Each one of the processors allocated to P_i accesses a specified entry in the list of vertices belonging to P_i and is allocated to the vertex recorded in this entry (if such exists). For each processor p allocated to an occurrence of a vertex in P we keep in $N(p)$ the identity of the processor allocated to the next vertex in P. (If p is allocated to the last vertex in P than $N(p)$ is assigned *end-of-list*.) This can be done with no overhead during the allocation.

The number of processors in Step 1 is bounded by n since the number of subpaths of P is at most $\sqrt{n}$ and the length of each subpath is also at most $\sqrt{n}$. Note that since P is not necessarily simple, a vertex may occur several times in P, and each such occurrence is represented by a different processor.

Step 2. Find the distance, counting edges, of each (occurrence of a) vertex in P from the end of P. This can be done in $O(\log n)$ time with the allocated processors using the well-known list ranking algorithm given in [18] as follows. Let $R(p)$ denote the distance of the (occurrence of a) vertex allocated to p.

```
for each allocated processor p pardo
  if N(p) = end-of-list
    then R(p) := 0
    else R(p) := 1
odpar
for i := 1 to ⌈log n⌉ do
  for each allocated processor p pardo
    if N(p) ≠ end-of-list
      then R(p) := R(p) + R(N(p))
      N(p) := N(N(p))
  odpar
od
```

Step 3. For each vertex in P find its occurrence in P which is closest to the end of P, among all its other occurrences. This also can be done in $O(\log n)$

time using $n^{1.5}$ processors, even on an EREW PRAM, by setting a binary tournament tree for each vertex. (Note that the number of leaves in each such binary tree is $O(\sqrt{n})$ as the number of occurrences of any vertex in P is $O(\sqrt{n})$, at most one occurrence per subpath.)

Step 4. Suppose a processor p is allocated to an occurrence of the vertex v in P. We update $N(p)$ to point to the processor allocated to the vertex following the *last* occurrence of v in P. This can be done in constant time using the allocated processors.

Observe that in Step 4 we transformed the nonsimple path P into a tree rooted at the end of P. (See Fig. 4.2.) All that is left to be done is to identify the path in this tree from the start of P to its end (the root). This is done in Step 5 by a variation of the list ranking algorithm.

Step 5. In the following algorithm we compute FLAG(p) for each allocated processor. FLAG(p) is assigned *true* only if p is allocated to an occurrence of a vertex which appears in the simple alternating path between the two endpoints of P. We start by setting FLAG(p) = *false* for all processors except the processor allocated to the first vertex in P.

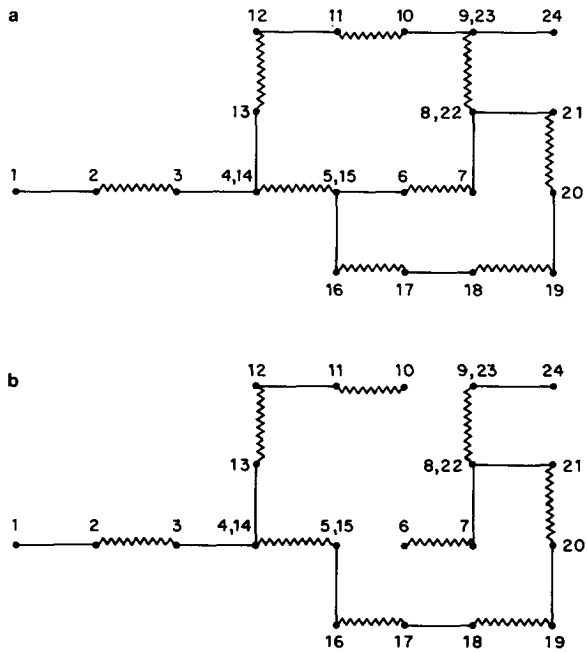

FIG. 4.2. (a, b) The numbers indicate the indices of the vertices in the path.

```
for i := 1 to ⌈log n⌉ do
  for each allocated processor p pardo
    if FLAG(p) = true then FLAG(N(p)) := true
    if N(p) ≠ end-of-list then N(p) := N(N(p))
  odpar
od
```

Clearly, after the execution of the above procedure FLAG(p) will be *true* only for processors allocated to vertices in a simple augmenting path.

We have shown that the updating process can be implemented in $O(\log n)$ time using only $n^{1.5}$ processors, and so the whole algorithm uses nm processors, and accomplishes its task in $O(n)$ time. The space complexity here is $O(nm + n^{2.5})$, as $O(nm)$ space is required for the scan in Phase 1 of the searching process and $O(n^{2.5})$ space is required to record the paths constructed in the same phase.

Note that during the scan of G' and during the path selection in the updating we may have a situation where n processors attempt to access the same location simultaneously. In fact, this is the worst case of such a situation. This implies that the transformation of this algorithm to an EREW PRAM model will increase the complexity by a factor of $\log n$.

We were not able to modify this algorithm so that at each phase an augmenting path of minimal length will be sought in sublinear time, without increasing the number of processors to $O(n^3)$. (Note that using $O(n^3)$ processors we can find an augmenting path of minimal length in polylog time using a transitive closure algorithm.) A similar idea, i.e., to construct augmenting paths in polylog time, appears in [10] in the context of the edge coloring problem.

5. Maximum Flow in 0–1 Networks

In this section we present parallel algorithms for finding maximum 0–1 flows in various types of networks.

5.1. *Preliminaries*

A maximum flow can be found by successive searches for *flow augmenting paths* (abbreviated as *f.a.p.*). An f.a.p. is a simple path from s to t through which a unit-flow can be shifted. (For an exact definition see, e.g., [3].) Each time an f.a.p. is detected it is used to augment the flow by a unit. It can be easily verified that the flow we get when no f.a.p. exists is a maximum integral flow, i.e., a maximum flow function whose values are all integers. Moreover, all the flows during the process are integral flows.

The search for an f.a.p. can be done by a labeling procedure similar to the procedure used for finding an augmenting path in the bipartite matching problem as follows.

DEFINITION. An edge $u \xrightarrow{e} v$ is *useful* iff it satisfies the following conditions: (1) u is labeled and v is unlabeled. (2) $u \xrightarrow{e} v \in E$ and $f(e) = 0$ or $v \xrightarrow{e'} u \in E$ and $f(e') = 1$.

We start by labeling the source s. In each step of the procedure we advance along all the *useful* directed edges at that time and label their head accordingly.

When t is labeled the search is completed and an update routine similar to the update routine in the bipartite matching problem is invoked. This routine uses the f.a.p. found in the search to augment the path by a unit flow.

The close resemblance between the two problems described was noted by Even and Tarjan [4]. They distinguished between three types of 0–1 networks: general networks, networks with no parallel edges, and networks where each vertex has one incoming or one outgoing edge. In the following subsections we analyze each of these types and present parallel algorithms for finding the maximum flow in each one of them. To make the presentation shorter we omit the proofs that appear in [4]. The reader is referred to [4] for further details.

5.2. *General 0–1 Networks*

The following variation of Theorem 2.1 concerning general networks is proved in [4].

THEOREM 5.1 [4]. *Let F be an integral flow in a* 0–1 *network, and let $|F^*|$ be the maximum flow value in that network. Then F has s set of $|F^*| - |F|$* edge *disjoint f.a.p.'s.*

Using this theorem we can develop algorithms for the maximum 0–1 flow problem similar to these presented for the bipartite matching problem. In the iteration i of these algorithms we search for an f.a.p. whose length is bounded by

$$l_i \leqslant \operatorname{Min}\left\{\frac{m}{m - i + 1}, n - 1\right\}$$

if such an f.a.p. is found it is used to augment the flow.

The proof that the stated bound on the length of the f.a.p.'s is sufficient is similar to the proof of Theorem 2.2 with the additional constraint that each f.a.p. must be simple, and is left to the reader.

The time complexity of the two resulting algorithms is given by the following variation of Theorem 2.2.

THEOREM 5.2. *Let the time complexity of each iteration i be bounded by $O(l_i^\alpha)$.*

(i) *If $0 \leqslant \alpha < 1$, then the time complexity of the whole algorithm is $O(m)$.*

(ii) *If $\alpha = 1$ then the time complexity of the algorithm is $O(m \log n)$.* (Note that in this case $O(m \log m) \neq O(m \log n)$, since parallel edges are possible.)

Proof. The proof of (i) is similar to the corresponding part in the proof of Theorem 2.3. The summation in the proof of (ii) is a little different and so is presented.

The total complexity in (ii) is bounded by

$$\sum_{i=1}^{m} \left\lfloor \operatorname{Min}\left\{ \frac{m}{m-i+1}, n-1 \right\} \right\rfloor.$$

Letting $k = [((n-2)/(n-1))m + 1]$, the sum is equal to

$$\sum_{i=1}^{k} \left\lfloor \frac{m}{m-i+1} \right\rfloor + (n-1)(m-k) \leqslant m\left\lceil \log m - \log\left(\frac{m}{n-1}\right) \right\rceil + m$$

$$= O(m \log n).$$

It can be shown that these bounds are also tight. ■

A little more has to be said on the modification of the algorithms. The modification of Algorithm 1 is straightforward using the modified labeling procedure described in Section 5.1. In the modification of Algorithm 2 we have to be more careful as now we do not have special edges like the matching edges. Two minor changes must be pointed out.

(1) In Phase 2 of the search we construct a network $G' = (V', E')$, where

$$E' = \{(x, y) \mid \text{a path from } x \text{ to } y \text{ was found in Phase 1}\}.$$

(2) In the updating process the path from s to t is reconstructed and afterward all its cycles (even and odd) are removed to get an f.a.p.

The modifications of Algorithms 1 and 2 yield two algorithms for finding maximum 0–1 flows. The first one finds the maximum flow using m processors in $O(m \log n)$ time. The second accomplishes the same job in $O(m)$ time, using nm processors.

5.3. *0–1 Networks with No Parallel Edges*

Following [4], we can calculate a tighter bound for the minimum-length f.a.p. in networks with no parallel edges.

THEOREM 5.3 [4]. *Let F be an integral flow in a 0–1 network with no parallel edges, and let $|F^*|$ be the maximum flow value in that network. If $|F^*| > |F|$, then there exists an f.a.p. P which satisfies*

$$|P| \leqslant \frac{\sqrt{2}(n-2)}{\sqrt{|F^*| - |F|}} + 2.$$

(Note that this bound is tighter than the one given in [4].)

Proof. Only a sketch of the proof will be given as it is similar to the proof given in [4].

We have to examine the maximization problem

$$\text{Max } l$$

$$\text{s.t.} \quad \sum_{i=1}^{l-1} n_i \leqslant n-2$$

$$2n_i n_{i+1} \geqslant |F^*| - |F|, \qquad \text{for} \qquad i = 1, \ldots, l-2.$$

Assuming that l is even and using the "arithmetic–geometric mean inequality" (i.e., $4ab \leqslant (a+b)^2$), we get that the maximum value of l is achieved when the cardinalities of all the n_is are equal to $[\sqrt{(|F^*| - |F|)/2}]$, which gives us the bound stated above. ■

This theorem suggests another algorithm for this type of networks. The algorithm is a variation of Algorithm 1 for finding a maximum matching in bipartite graphs. In each iteration of the algorithm we search for the minimum-length f.a.p., and use one of the shortest f.a.p.'s to augment the flow value. The algorithm uses m processors like Algorithm 1, and its time complexity is $O(\text{Min}\{n^{1.5}, m \log n\})$, as proved below.

THEOREM 5.4. *The time complexity of the algorithm described is equal to* $O(\text{Min}\{n^{1.5}, m \log n\})$.

Proof. The upper bound $O(m \log n)$ is the bound derived for general 0–1 networks. The second term in the upper bound is achieved by summing the complexities of each augmenting iteration, plus the complexity of the last iteration when no f.a.p. was found. (Note that in this type of networks $|F^*| \leqslant n$.)

$$\sum_{i=1}^{|F^*|} \left(\frac{\sqrt{2}(n-2)}{\sqrt{|F^*| - (i-1)}} + 2 \right) + n \leqslant n + 2|F^*| + \sqrt{2}n \int_{x=1}^{|F^*|} \frac{1}{\sqrt{x}}\, dx$$

$$\leqslant 2^{1.5} n(\sqrt{|F^*|} + 1) = O(n^{1.5}) \quad ■$$

5.4. *0–1 Networks Where Each Vertex Has One Incoming or One Outgoing Edge*

In [4] the following theorem about this type of networks (abbreviated as type 2) is proved.

THEOREM 5.5. *Let F be an integral flow in a* 0–1 *network of type* 2 *and let* $|F^*|$ *be the maximum flow value in that network. Then F has a set of* $|F^*| - |F|$ vertex *disjoint f.a.p.'s.*

Note that this theorem is essentially identical to Theorem 2.1 and in fact the bipartite matching problem can be viewed as a special case of finding maximum flows in 0–1 networks of this type. The algorithms derived for the general 0–1 networks will give here the same complexity bounds as the maximum bipartite matching algorithms. That is, the first parallel algorithm finds maximum 0–1 flows in $O(n \log n)$ time using m processors, while the second does the same job in $O(n)$ time using nm processors.

5.5. *Other Applications*

We conclude this section by mentioning two combinatorial problems which can be reduced to the maximum 0–1 flow problem, as shown in [4].

(a) Deciding the edge connectivity of a graph.

The network we get here is a general 0–1 network and so the connectivity can be found in $O(m)$ running time.

(b) Deciding the vertex connectivity of a graph.

The network we get here is a 0–1 network of type 2 and so the connectivity can be found in $O(n)$ running time.

Remark. Note that in the special cases of deciding the connectivity and biconnectivity of an undirected graph we can use the faster parallel algorithms given in [15, 16]. These algorithms run in $O(\log n)$ time using $n + m$ processors.

6. CONCLUSIONS AND FURTHER RESEARCH

Two algorithms for constructing maximum matching in bipartite graphs were presented in this paper. The first is easy to implement, uses m processors, and runs in $O(n \log n)$ time. The second algorithm runs in $O(n)$ time but uses nm processors; i.e., reducing the running time by a factor of $O(\log n)$ comes at the cost of increasing the number of processors by a factor of n. It would be interesting to know whether there is an $O(n)$ parallel algorithm for this problem which uses a smaller number of processors. Another interesting question is whether there is a deterministic parallel algorithm for bipartite

matching which uses a polynomial number of processors and runs in $o(n)$ time.

The first algorithm can be implemented in an asynchronous communication network using the synchronizer given in [2]. The resulting distributed algorithm is faster than any published distributed algorithms for the same problem. (*Postscript.* The reader is referred to [13], where this distributed algorithm is presented.)

Another point which should be noted is that the algorithms are parallelizations of sequential algorithms which do not give the best complexity results. One of the reasons for this is that the overhead needed in the parallelization of these sequential algorithms is smaller than the overhead needed for the parallelizations of faster sequential algorithms. This fact suggests that maybe there are other combinatorial problems for which the time optimal parallel algorithms are not parallelizations of the best known sequential algorithms for the same problems.

The techniques used by our maximum bipartite matching algorithms were used to obtain similar parallel algorithms for various types of 0–1 flow problems, which indicates the general nature of these techniques. Some of these algorithms can also be implemented in an asynchronous communication network, and the resulting algorithms are faster than the known distributed algorithms for these problems.

ACKNOWLEDGMENTS

We thank the anonymous referees for many helpful comments.

REFERENCES

1. Ajtai, M., Komlos, J., and Szemeredi, E. An $O(n \log n)$ sorting network. *Combinatorica* **3** (1983), 1–19.
2. Awerbuch, B. An efficient network synchronization protocol. *Proc. 16th ACM Symposium on Theory of Computing,* 1984, pp. 522–525.
3. Even, S. *Graph Algorithms.* Comput. Sci. Press, Rockville, MD, 1979.
4. Even, S., and Tarjan, R. E. Network flow and testing graph connectivity. *Siam J. Comput.* **4** (1975), 507–518.
5. Galil, Z., and Pan, V. Improved processor bounds for algebraic and combinatorial problems in RNC. *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science,* 1985, pp. 490–495.
6. Goldberg, A. V., and Tarjan, R. E. A new approach to the maximum flow problem. *Proc. 18th ACM Symposium on Theory of Computing,* 1986, pp. 136–145.
7. Hopcroft, J. E., and Karp, R. M. An $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *Siam J. Comput.* **2** (1973), 225–231.
8. Israeli, A., and Shiloach, Y. An improved parallel algorithm for maximal matching. *Inform. Process. Lett.* **22** (1986), 57–60.

9. Karp, R. M., Upfal, E., and Wigderson, A. Constructing a perfect matching is in random NC. *Proc. 17th ACM Symposium on Theory of Computing* 1985, pp. 22–32.
10. Lev, G. Size bounds and parallel algorithms for networks. TR CST-8-80, Department of Computer Science, University of Edinburgh, 1980.
11. Micali, S., and Vazirani, V. V. An $O(\sqrt{|V|}\ |E|)$ algorithm for finding maximum matching in general graphs. *Proc. 21st Annual IEEE Symposium on Foundations of Computer Science,* 1980, pp. 17–27.
12. Mulmuley, K., Vazirani, U. V., and Vazirani, V. V. Matching is as easy as matrix inversion. *Proc. 19th ACM Symposium on Theory of Computing* 1987, pp. 345–354.
13. Schieber, B., and Moran, S. Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: Maximum matchings. *Proc. 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* 1986, pp. 282–292.
14. Shiloach, Y., and Vishkin, U. An $O(n^2 \log n)$ parallel max flow algorithm. *J. Algorithms* **3** (1982), 128–146.
15. Shiloach, Y., and Vishkin, U. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms* **3** (1982), 57–67.
16. Tarjan, R. E., and Vishkin, U. An efficient parallel biconnectivity algorithm. *Siam J. Comput.* **14** (1985), 862–874.
17. Vishkin, U. Implementation of simultaneous memory address access in models that forbid it. *J. Algorithms* **4** (1983), 45–50.
18. Wyllie, J. C. The complexity of parallel computation. TR 79-387, Department of Computer Science, Cornell University, 1979.

Statement of ownership, management, and circulation required by the Act of October 23, 1962, Section 4369, Title 39, United States Code: of

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING

Published bimonthly by Academic Press, Inc., 1 East First Street, Duluth, MN 55802. Number of issues published annually: 6. Editors: Dr. Kai Hwang, Departments of Electrical Engineering and Computer Science, University of Southern California, Los Angeles, CA 90089-0781, and Dr. Leonard Uhr, Computer Sciences Department, University of Wisconsin, Madison, WI 53706.
Owned by Academic Press, Inc., 1250 Sixth Avenue, San Diego, CA 92101. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, and other securities: None.
Paragraphs 2 and 3 include, in cases where the stockholder or security holder appears upon the books of the company as trustee or in any other fiduciary relation, the name of the person or corporation for whom such trustee is acting, also the statements in the two paragraphs show the affiant's full knowledge and belief as to the circumstances and conditions under which stockholders and security holders who do not appear upon the books of the company as trustees, hold stock and securities in a capacity other than that of a bona fide owner. Names and addresses of individuals who are stockholders of a corporation which itself is a stockholder or holder of bonds, mortgages, or other securities of the publishing corporation have been included in paragraphs 2 and 3 when the interests of such individuals are equivalent to 1 percent or more of the total amount of the stock or securities of the publishing corporation.
Total no. copies printed: average no. copies each issue during preceding 12 months: 1175; single issue nearest to filing date: 1258. Paid circulation (a) to term subscribers by mail, carrier delivery, or by other means: average no. copies each issue during preceding 12 months: 740; single issue nearest to filing date: 788. (b) Sales through agents, news dealers, or otherwise: average no. copies each issue during preceding 12 months: 0; single issue nearest to filing date: 0. Free distribution by mail, carrier delivery, or by other means: average no. copies each issue during preceding 12 months: 90; single issue nearest to filing date: 90. Total no. of copies distributed: average no. copies each issue during preceding 12 months: 830; single issue nearest to filing date: 878.

(Signed) Roselle Coviello, Senior Vice President