PARALLEL ALGORITHM FOR FINDING MAXIMUM
BIPARTITE MATCHINGS AND MAXIMUM FLOW
IN 0-1 NETWORKS.

by

E. Schieber and S. Moran

Technical Report #352
January 1985
Parallel Algorithms for Finding
Maximum Bipartite Matchings
and Maximum Flow in 0-1 Networks

by

B. Schieber and S. Moran

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

ABSTRACT

Two parallel algorithms for finding a maximum matching in bipartite graphs are presented. The first finds a maximum matching using \(O(|E|)\) processors with time complexity of \(O(|V| \log |V|)\), while the second does the same job using \(O(|V| |E|)\) processors in \(O(|V|)\) time. Simple modifications of these algorithms induce parallel algorithms for finding maximum 0-1 flows, which are also presented.
1. Introduction

In this paper we present efficient parallel algorithms for finding a maximum matching in bipartite graphs and maximum flow in 0-1 networks. The precise statement of the first problem is as follows:

Let \( G = (X, Y, E) \) be a bipartite graph, where \( X \cup Y = V \) is a set of vertices and \( E \subseteq X \times Y \) is a set of edges; 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 best known sequential algorithm for this problem is due to Hopcroft and Karp [4], and it has a time complexity of \( O(\sqrt{|V|} \cdot |E|) \). An algorithm of a similar complexity that solves the problem for general graphs is given in [8]. Even and Tarjan [3] noted that the maximum bipartite matching problem can be reduced to the problem of finding maximum flow in a certain type of 0-1 networks. These networks have either one incoming edge or one outgoing edge. Using that reduction they derived a similar algorithm for solving the maximum bipartite matching problem, with the same time complexity.

The only published deterministic parallel algorithm for this problem is an application of the parallel algorithm for finding maximum flow in networks presented in [9]. This algorithm has time complexity of \( O(|V|^{1.5} \log |V|) \) using \( O(|E|/\sqrt{|V|}) \) processors.

Many other algorithms, which deal with related problems appear in the literature. In this category we can mention [6] where a randomized algorithm for finding a maximum matching in general graphs in polylog time, using a polynomial (through quite large) number of processors is presented. Also, in [5] an algorithm for finding a maximal matching in general graphs in polylog time and \( O(|E|) \) processors is presented (a maximal matching is a matching that is not contained in any larger matching).

Two new deterministic parallel algorithms for finding a maximum bipartite matching are presented in this paper. The first utilizes \( O(|E|) \) processors and has the time complexity of \( O(|V| \log |V|) \), and the second runs in \( O(|V|) \) time and uses \( O(|V| \cdot |E|) \) processors. Both algorithms are superior to the previous in their running time while
using a moderate number of processors.

By slight modifications to our algorithms we get efficient parallel algorithms for finding maximum 0-1 flow in various types of networks. The complexity of the resulted algorithms is summarized in the following table:

<table>
<thead>
<tr>
<th>Network Type</th>
<th>No. of Processors</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>General networks</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>Networks without parallel edges</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Networks with one incoming or one outgoing edge</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>

These bounds are superior to those obtained by implementing the algorithm in [9] to the problem of finding maximum flow in 0-1 networks.

Some of the presented algorithms can be implemented also in an asynchronous communication network, using a standard synchronizer described in [1]. The resulted distributed algorithms are superior to the previous distributed algorithms known for these problems.

The rest of the paper proceeds as follows. In the next section the computation model is described, and in section 3 the outlines of the algorithms are given. The next two sections are devoted to a closer analysis of the algorithms, and in section 6 the algorithms for maximum 0-1 flow are discussed.

2. Computation Model

There is a variety of models for parallel processing; the model used here is abbreviated as CRCW PRAM, and its exact definition is given in [10]. It is a synchronized model, in which all the processors have an access to a common memory. We assume that the input graph is represented in the common memory by a list of its edges.
case the input graph is bipartite, it is also assumed that the format of each edge is 
\([x, y]\) where \(x \in X\) and \(y \in Y\) (in our case this assumption may be avoided since we can 
preprocess the input \(O(|V|)\) time using \(O(|E|)\) processors).

In a single computation step each processor can perform an atomic operation in 
it local memory, or read (write) a bounded amount of information from (in) the common memory (generally this amount is logarithmic in the size of the problem). Simultaneous reading from the same memory location is allowed, simultaneous writing to 
the same location is also allowed, and in that case it is assumed that one of the processors 
has accomplished the writing successfully. The time complexity (or depth) of an 
exeuction of an algorithm in this model is the number of computation steps executed 
by it. We can implement the same algorithms on a more conservative model, where no 
simultaneous reading or writing is allowed (EREW), in this case the time complexity is 
increased by a factor of at most \(\log w\), where \(w\) is the maximum number of processors 
attempting to access the same location simultaneously. This is true because each time 
such a conflict occurs each processor knows the identities of all the other processors 
competing with it for access the same location. Therefore, the transformation to this 
model is straightforward and not described here.

3. Outlines of the 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 using it, simply by interchanging matching and free 
edges along this path.

The algorithms presented here, like most other known algorithms for maximum 
matching, are based on finding augmenting paths. The reason for this is the following
Theorem 3.1 [4]: Let \( M \) be a matching and \( M^* \) a maximum matching. Then \( M \) has a set of \( |M^*| - |M| \) vertex disjoint augmenting paths.

Theorem 3.1 guarantees that, when no augmenting path exists the matching is of maximum cardinality. The basic difference between the algorithms in ([4], [3], [9]) and ours is that at each phase of our algorithm only one augmenting path is searched, and not a maximal set of such paths. The resulted increase in the number of phases is compensated by careful bookkeeping. In this bookkeeping we use the fact that, at the present state of the art, 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 \( k = \text{Min}(|X|, |Y|) \) times. In the \( i^{th} \) iteration of this routine an augmenting path whose length is bounded by \( l = \frac{2|M_{i-1}|}{k-i+1} + 1 \) 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 \( \leq l \) exists, and the matching is not changed.

The validity of the algorithms stems from the following theorem.

Theorem 3.2: 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 2 cases.

Case 1: An augmenting path was found in iteration \( i \).

We get:

\[
|M^*| - |M_i| = |M^*| - (|M_{i-1}| + 1) \leq k - (i-1) - 1 = k - i.
\]
Case 2: No augmenting path was found in iteration $i$. Assume for the contradiction that $|M'|-|M_i| > k-i$. By theorem 3.1 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 $\frac{l-1}{2}$ of the edges are matching edges and so at least one augmenting path $P$ must satisfy the inequality:

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

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

$$|P| \leq \frac{2|M_{i-1}|}{k-i+1} + 1 = \lambda_i$$

contradicting the fact that no such path existed.

The above theorem guarantees that after $\min(|X|, |Y|)$ iterations the computed matching is the maximum matching.

3.1. The Effect of One Iteration's Complexity on the Overall Complexity

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

**Theorem 3.3:** Let the complexity of each iteration $i$ be bounded by $O(\lambda_i^\alpha)$.

(i) If $0 < \alpha < 1$, then the complexity of the whole algorithm is $O(|V|)$. (ii) If $\alpha = 1$ then the complexity of the algorithm is $O(|V| \log |V|)$.

**Proof:** The theorem is proved simply by summing up the times needed for each iteration:

(i): \[
\sum_{i=1}^{k} \left(\frac{2|M_{i-1}|}{k-i+1} + 1\right)^\alpha \leq \int_{x=0}^{\frac{3k}{k-x}} \left(3k\right)^\alpha \left(\frac{1}{1-\alpha} (x^{1-\alpha} - 1) \right) \, dx \leq \frac{1}{1-\alpha} 3^\alpha k = O(|V|)
\]

Which proves this part of the theorem.

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

To see that this bound is tight, observe that when $|X| = |Y| = |V|/2$ the
summation gives even for $a = 0$:

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

(ii): \[ \sum_{i=1}^{k} \left( \frac{2|M_{i-1}|}{k-i+1} + 1 \right) \leq k + \int_{a=0}^{k} \frac{2|M_{i-1}|}{k-x} \, dx = k + 2|M^{*}| \log k = O(|V| \log |V|) \]

Again, to see that this bound is tight, note that when $|X| = |Y| = \lfloor V/2 \rfloor$ and $G$ contains a perfect matching (i.e., a matching of cardinality $|X|$) then the summation gives:

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

Observe that the number of iterations in the algorithm is linear in $|V|$, and so we can not 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 $O(|V| \log |V|)$ and $O(|V|)$ respectively. These bounds are the same as the previous 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 3.3. 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 considerably the number of processors. The description contains mainly the search and update routine which is invoked in each iteration of the algorithm.

4. Algorithm I

The first algorithm can be implemented using exactly $|E|$ processors, where each processor represents an edge. The search for an augmenting path of minimum length will be done using a BFS-like scanning, while the updating step will be done by back-
tracking the tree built during scanning.

**4.1. The BFS Scanning**

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 $\leq l_i$ is found). During the scan the vertices are labelled, 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 yet unscanned free edges emanating from vertices labelled by $2t$ scan the ends of their related edges. Every unlabelled end of those edges is labelled by $2t + 1$, and its pointer is initialized to point the vertex at the other end of the edge (which is labeled $2t$). If one of the newly labelled vertices is a free vertex then an augmenting path is found; else, if $2t + 1 \geq l_i$, no augmenting path of length $\leq l_i$ exists. In both of these cases the scanning is stopped. Otherwise all the matching edges emanating from the vertices labelled by $2t + 1$ are scanned in a similar way (these edges are yet unscanned), and their unlabelled ends are labelled by $2t + 2$.

This completes the execution of the $t^{th}$ step, and the execution of step $t + 1$ is started.

**4.2. The Updating Step**

In the case that 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 labelled 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 $O(|V|)$ processors in $O(\log |V|)$ 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.
4.3. The Validity and Complexity of the Algorithm

It can easily be verified that the parallel labeling process really simulates BFS-scanning, 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 3.3 and the fact that the path selection in the updating step takes \( O(\log |V|) \) time per iteration, we find that the overall time complexity of the algorithm is \( O(|V| \log |V|) \). The space complexity of the algorithm is easily shown to be \( O(|E|) \). The number of utilized processors is \( |E| \), as claimed. To improve this bound to \( O(|M| \log |M|) \), we modify the BFS-scanning stage so that the scanning is terminated only if an alternating path of minimal length is found, or if \( |M_{i-1}| + 1 \) steps are completed (in the \( i^{th} \) iteration). In this latter case the whole algorithm is terminated, as no augmenting path exists.

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 in 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 \( O(|V|) \), as at the end of each step of the BFS-scanning all the processors must check whether the scanning has already terminated.

5. Algorithm II

The second algorithm has a time complexity of \( O(|V|) \) achieved by using a search and update routine which has a running time of \( O(\sqrt{t}) \) time in iteration \( t \). Recall that,
by Theorem 3.3, the \( O(|V|) \) running time of the algorithm will follow by any algorithm that performs the \( \tau \)th iteration in \( O(4^n) \) time for any \( 0 \leq \alpha < 1 \). 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.

5.1. The Searching Process

The search for a path of length \( l \) 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 \( \sqrt{l} \).

Each pair \([x, y]\) found at this phase is recorded. The corresponding alternating path that was found from \( x \) to \( y \) is also recorded, as a list of its edges in the order of their appearance in the path. Each edge is also labelled by its index in the path. Obviously the size of each list is bounded by \( \sqrt{|V|} \).

The search is accomplished by activating \(|X|\) copies of the parallel BFS-scanning algorithm described in the previous section, one for each vertex in \( X \). The paths are stored by activating \(|X|\times|Y|\) 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 needed processors is \( O(|V| \times |E|) \times O(|V| \times |E|) \) for the recording.

Phase 2: Define a new graph \( \tilde{G} = (V, \tilde{E}) \) where -

\[ \tilde{E} = E \cup \{(x, y) \mid x \in X, y \in Y \text{ a path from } x \text{ to } y \text{ was found in step 1}\} \]

Obviously the graph \( \tilde{G} \) is bipartite and the edges in \( \tilde{M} \) still form a matching in it. The construction can be done in constant time using \( |V|^2 = O(|V| \times |E|) \) processors.

After the construction of \( \tilde{G} \) we search it for an augmenting path whose length is bounded by \( \tilde{l} = 2\sqrt{l} - \frac{3\sqrt{l} - 1}{2\sqrt{l} + 1} \). This search is also done using the search
routine of the previous algorithm with running time \( O(\sqrt{V}) \) and with \( O(|V|^2) \) processors, as \( \tilde{E} \) contains \( O(|V|^2) \) edges. Each such augmenting path 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 section.

The correspondence between augmenting paths in the two graphs is proved by the following lemma.

**Lemma 5.1:** \( \tilde{G} \) contains an augmenting path of length at most \( \tilde{l} \) 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 in an augmenting path found in \( \tilde{G} \), whose ends are \( x \in X \) 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, and whose length is bounded by \( \sqrt{l} \). Each matching edge in \( \tilde{G} \) is also a matching edge in \( G \). It follows that to each augmenting path \( \tilde{P} \) of length \( \tilde{l} \) in \( \tilde{G} \), there corresponds an alternating path \( P \) in \( G \) whose length is not larger than \( \frac{l+1}{2} \sqrt{l} + \frac{l-1}{2} = l \), 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.

### 5.2. The Updating Process

If an augmenting path was found in \( \tilde{G} \), the corresponding augmenting path in \( G \) is constructed from it by deleting cycles, as described in the proof of Lemma 5.1. All we must show is that this can be done within the stated time and processors bounds.

Assume that the augmenting path \( \tilde{P} \) was found in \( \tilde{G} \) (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 sub-paths, each corresponds to a certain free edge in \( \tilde{P} \). The \( i \)th sub-path will be...
abbreviated as $P_i$. Note that every $P_i$ is simple as it is an output of the searching step of algorithm 1, which searches only for simple paths, and that these paths are traversed in $P$ in the same direction they were traversed during their construction.

At first we allocate a processor to each occurrence of an edge in $P$. This can be done parallelly in constant time in the following way:

(a) Allocate a processor to each matching edge in $\tilde{P}$; these edges appear also in $P$.

(b) Allocate $\sqrt{|V|}$ processors to each list of edges corresponding to a sub-path $P_i$.

Each one of those processors checks a specified entry in that list and is allocated to the edge recorded in that entry (if such an edge exists).

The number of processors in step (b) is bounded by $|V|$ since the number of sub-paths of $P$ is at most $\sqrt{|V|}$. Note that since $P$ is not necessarily simple, an edge $e$ may occur several times in $P$, and each such occurrence is represented by a different processor.

Using the allocated processors we construct a table we call $TAB$. This table has $|V|$ rows and $c$ columns, where $c \leq \frac{\lceil \frac{l+1}{2} \rceil}{2}$. Each entry in $TAB$ represents a pair [vertex, sub-path] where $TAB(v,j)$ contains the index of $v$ in $P_j$ (zero, in case $v$ does not appear in $P_j$); i.e., $TAB(v,j) = i$ means that $v$ is the $i^{th}$ vertex in $P_j$. Note that $TAB(v,j)$ was recorded for all $v$ and $j$ in phase 1.

In the beginning $TAB$ is initialized to zero; this can be done in constant time with only $O(|V|^1.5)$ processors, as the size of $TAB$ is bounded by $|V|^{1.5}$. Afterwards, each processor that was allocated to an occurrence of a free edge $e = (u,v)$ in $P_j$ sets the entries $TAB(u,j)$ and $TAB(v,j)$. Note that no write conflicts occur at this stage as each $P_i$ is simple.

We now use $TAB$ to transform $P$ into a simple alternating path (which must be an augmenting path). This is done in $O(\sqrt{V})$ phases. In phase $i$ we delete the first cycle which starts with a vertex in $P_i$, if such a cycle exists, and update the path accordingly. Note that by the fact that each $P_i$ is initially simple, after the deletion of this cycle, the resulted path does not contain any other cycles starting at $P_i$, and by induction, the output path of phase $i$ does not contain any cycles starting with a vertex in $P_j$ for
j \leq i$; hence, after $c$ phases the output path will be simple. $P$ is represented by the table $TAB$ and the processors allocated to each edge in it. The updating of $P$ at each phase is done by zeroing some entries in $TAB$ and by deallocating some of the edge-processors.

Each phase $i$ consists of three steps. In the first step we find the last occurrence in $P$ of each vertex $v$ in $P_i$. This can be done by finding the last nonzero entry of the $v^{th}$ row of $TAB$, for each such $v$. Determining whether $TAB(v,j)$ is the last nonzero entry (for some $j$) can be accomplished by $c - j$ processors $p_{j+1}, \ldots, p_c$ as follows: Processor $p_k$ writes no if either $TAB(v,j) = 0$ or $TAB(v,k) \neq 0$. It is easily verified that $TAB(v,j)$ is the last nonzero entry iff no processor writes no. The total number of processors needed is $O(\sqrt{c^2}) = O(\sqrt{|V|^1})$, as there are at most $\sqrt{c}$ vertices in $P_i$, and for each such vertex we need less than $c^2$ processors.

In the second step we find the first vertex $u$ in $P_i$ whose last occurrence in $P$ is not its (unique) occurrence in $P_i$. If there is no such vertex, no cycle in $P$ starts with a vertex in $P_i$, and the whole phase is terminated. This step can be accomplished in constant time and $O(i)$ processors, using a technique similar to the one used in the previous step.

At the third and last step we use the information gathered in the previous two steps to delete the maximal length cycle starting at $u$. To this end, note that an occurrence of an edge $e$ in $P$ is in this cycle if this occurrence of $e$ comes between the first and last occurrences of $u$. This can be checked by each edge processor in constant time. An edge processor that finds out that its edge is in that cycle, deallocates itself, and in the case that the corresponding edge is also free, is also zeroing the corresponding entries in $TAB$.

After the $c$ phases are terminated we are left with an augmenting path in $G$. The updating of this path (i.e., transforming each edge in it from "matching" to "free" or vice versa) can be done in constant time.

We have shown that the updating process can be implemented in $O(\sqrt{|V|})$ time using only $O(\sqrt{|V|^1})$ processors, and so the whole algorithm uses $O(\sqrt{|V| |E|})$ processors, and accomplishes its task in $O(\sqrt{|V|})$ time. The space complexity here is
\( O(|V| |E| + |V|^2) \), as \( O(|V| |E|) \) space is required for the scanning in phase 1 of the searching process and \( O(|V|^2) \) space is required to record the paths constructed in that phase.

Note that during the scanning of \( G \) and during the path selection in the updating process we may have a situation where \( O(|V|) \) 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 \( O(\log |V|) \).

We were not able to modify this algorithm so that at each phase an augmenting path of minimal length will be sought in sub-linear time, without increasing the number of processors considerably. Note that using \( O(|V|^3) \) processors this can be done in polylog time using a transitive closure algorithm. A similar idea, to construct augmenting paths in the context of the edge coloring problem appears in [7].

6. Maximum Flow in 0-1 Networks

6.1. Preliminaries

Several combinatorial problems can be reduced to the problem of finding maximum flow in 0-1 networks. The precise statement of this problem is as follows:

Let \( N = (G,s,t) \) be a 0-1 network consisting of a directed graph \( G = (V,E) \) and two specified vertices \( s \) (source) and \( t \) (sink).

A flow function \( f \) is an assignment of a number \( f(e) \) to each edge \( e \in E \), such that the following two conditions hold:

1. \( 0 \leq f(e) \leq 1 \)
2. Let \( \alpha(v) \) and \( \beta(v) \) be the sets of edges incoming to and outgoing from \( v \), respectively.

For every \( v \in V - \{s,t\} : \sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e) \)

An integral flow is a flow function whose values are all integers (in our case they must be 0 or 1).
The flow value $|f|$ is defined as: $\sum_{e \in A(t)} f(e) - \sum_{e \in B(t)} f(e)$.

A maximum flow is a flow with the maximum flow value.

The maximum $0-1$ flow problem is: Given a $0-1$ network $N$ find a maximum flow.

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 for example [2]). Each time an f.a.p. is detected it is used to augment the flow by a unit. It can easily be verified that the flow we get when no f.a.p. exists is maximum integral flow. Moreover, all the flows during the process are integral flows.

The search for an f.a.p. can be done by a labelling procedure similar to the procedure used for finding an augmenting path in the bipartite matching problem. Initially the source $s$ is labelled. In each step of the procedure we advance along all the useful directed edges at that time and label their head.

An edge $uv$ is useful if it satisfies the following conditions:

1. $u$ is labelled and $v$ is unlabelled.
2. $uv \in E$ and $f(e) = 0$ or $v'u \in E$ and $f(e') = 1$.

When $t$ is labelled 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 described problems was noted by Even and Tarjan [3]. 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 outgoing edge. In the following subsections we analyse each of these types and present parallel algorithms for finding the maximum flow in them.

### 6.2. General $0-1$ Networks

The following variation of theorem 3.1 concerning these networks is proved in [3].

**Theorem 6.1** [3]: 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 a 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 those presented for the bipartite matching problem. In the \( i \)th iteration of these algorithms we search for an f.a.p. whose length is bounded by:

\[
L_i \leq \min \{ \frac{|E|}{|E|-i+1}, |V|-1 \}
\]

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 3.2 with the additional constraint that each f.a.p. must be simple, and is left to the reader.

The complexity of the two resulted algorithms is given by the following variation of theorem 3.3.

**Theorem 6.2:** Let the complexity of each iteration \( i \) be bounded by \( O(L_i^a) \).

(i) If \( 0 < \alpha < 1 \), then the complexity of the whole algorithm is \( O(|E|) \).

(ii) If \( \alpha = 1 \) then the complexity of the algorithm is \( O(|E| \log |V|) \) (note that in this case \( O(|E| \log |E|) \neq O(|E| \log |V|) \), since parallel edges are possible).

**Proof:** The proof of (i) is similar to the corresponding part in the proof of theorem 3.3. The summation in the proof of (ii) is a little different and so presented.

The total complexity in (ii) is bounded by:

\[
\sum_{i=1}^{k} \left[ \min \{ \frac{|E|}{|E|-i+1}, |V|-1 \} \right]
\]

If we denote \( k = \left[ \frac{|V|-2}{|V|-1}|E| + 1 \right] \) we get that the sum is equal to:

\[
k \sum_{i=1}^{k} \left[ \frac{|E|}{|E|-i+1} \right] + (|V|-1)(|E|-k) \leq |E| \left[ \log |E| - \log \left( \frac{|E|}{|V|-1} \right) \right] + |E| = O(|E| \log |V|)
\]

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 I is straightforward using the modified labelling procedure described in section 6.1. In the modification of algorithm II we have to be more careful, as we do not have now special edges like the matching edges. Two minor changes must be pointed out. First, in phase 1 of the search the constructed network \( \tilde{N} \) consists of the directed graph \( \tilde{G} = (\tilde{V}, \tilde{E}) \), where:
\[ E = \{(x, y) \mid \text{a path from } x \text{ to } y \text{ was found in step 1}\} \]

Second, in the updating process the path from \( s \) to \( t \) is reconstructed and afterwards all its cycles (even and odd) are removed to get an f.a.p.

The modifications of algorithms I and II yield two algorithms for finding maximum 0-1 flow. The first one finds the maximum flow using \( O(|E|) \) processors with time complexity of \( O(|E| \log |V|) \), while the second accomplishes the same job in \( O(|E|) \) time, using \( O(|V| \cdot |E|) \) processors.

### 6.3. 0-1 Networks with no Parallel Edges

In this case we can calculate a more tight bound for the minimum length f.a.p.

**Theorem 6.3** [3]: 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:

\[ |P| \leq \frac{\sqrt{2(|V|-2)}}{\sqrt{|F^*| - |F|}} + 2 \]

(Note that this bound is tighter than the one given in [3].)

**Proof:** Only a sketch of the proof will be given as it is similar to the proof given in [3].

We have to examine the following maximization problem:

Max \( l \)

subject to:

\[ \sum_{i=1}^{l} V_i \leq |V| - 2 \]

\[ 2|V_i| - |V_{i+1}| \geq |F^*| - |F|, \text{ for } i = 1, \ldots, l - 2 \]

Assuming \( l \) is even and using the "arithmetic-geometric mean inequality" (i.e. \( 4ab \leq (a+b)^2 \)), we get that the maximum value of \( l \) is achieved when the cardinalities of all the \( V_i \)’s are equal to \( \left\lceil \frac{\sqrt{|F^*| - |F|}}{2} \right\rceil \), which gives us the bound stated above.

This theorem suggests another algorithm for this type of networks. The algorithm is a variation of algorithm I 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 \( O(|E|) \) processors like algorithm I, and its time complexity is \( O(\min\{\frac{|V|^1.5}{1.5}, |E| \log |V|\}) \), as will be proved.
Theorem 6.4: The complexity of the described algorithm is equal $O(\min\{ |V|^1.5, |E| \log |V| \})$.

Proof: The upper bound $O(|E| \log |V|)$ 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 $|F^*| \leq |V|$).

$$\sum_{i=1}^{k} \left( \frac{\sqrt{2(|V|-2)} + 2}{\sqrt{|F^*| - (i-1)}} \right) + |V| \leq |V| + 2|F^*| + \sqrt{2} |V| \int_{z=1}^{F^*} \frac{1}{\sqrt{z}} dz \leq 2^{1.5} |V| (\sqrt{|F^*|} + 1) = O(|V|^{1.5})$$

6.4. 0-1 Networks where Each Vertex has One Incoming or Outgoing Edge

In [3] the following theorem about this type of networks (abbreviated as type 2) is proved.

Theorem 6.5: 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.

Pay attention that this theorem is identical to theorem 3.1 and in fact the bipartite matching problem can be viewed as a special case of finding maximum flow in 0-1 networks of this type. The derived algorithms for the general 0-1 networks will give here the same complexity bounds as the matching algorithms.

We conclude this section by mentioning two combinatorial problems which can be reduced to the max-flow problem in 0-1 networks.

(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 $O( |E| )$ 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 $O(|V|)$ running time.

7. 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 $|E|$ processors, and runs in $O(|V| \log |V|)$ time. This algorithm can be implemented in an asynchronous communication network using the synchronizer mentioned in [1]. The resulted distributed algorithm has communication complexity of $O(|V|(|E| + k|V| \log |V|))$ and time complexity of $O(|V| \left(\frac{\log |V|}{\log k}\right)^2)$ for all $2 \leq k < |V|$. The time complexity is superior to the time complexity of the algorithms obtained by applying the algorithm in [1] to the problem of finding a maximum bipartite matching, while the communication complexity is superior only in sparse graphs.

The second algorithm runs in $O(|V|)$ time but uses $O(|V| |E|)$ processors; i.e., reducing the running time by a factor of $O(\log |V|)$ is resulted by increasing the number of processors by a factor of $O(|V|)$. It is interesting to know whether there is an $O(|V|)$ parallel algorithm for this problem which uses 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 $o(|V|)$ time.

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 algorithms were used to obtain similar parallel algorithms for various types of 0-1 flow problems, which indicates the general nature of those techniques. Some of these algorithms can also be implemented in an asynchronous communication network, and the resulted algorithms are faster than the known
distributed algorithms for these problems.

REFERENCES


